Simple, Fast, and Robust Ray Casting of Irregular Grids

Paul Bunyk
Department of Physics
SUNY at Stony Brook
Stony Brook, NY 11794
paul@pbunyk.physics.sunysb.edu

Arie Kaufman
Center for Visual Computing (CVC)
and Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794
ari@cs.sunysb.edu

Cláudio T. Silva
AT&T Shannon Laboratory
180 Park Avenue
Florham Park, NJ 07932-0971
csilva@research.att.com

Abstract

In this paper we describe a simple and efficient ray casting engine that is suitable for the rapid exploration of irregular grids composed of tetrahedra cells, or other cell complexes where cells have been broken up into faces. In our method, in a preprocessing phase, all the cells are broken into their corresponding faces. Visibility determination is performed after all the faces have been transformed into screen space; here we compute for each pixel an ordered list of the stabbing boundary faces. The final phase is the actual ray casting, which is performed independently for each pixel, and is basically a walk in the cell complex inside each component of the stabbing ordered list. For color calculations, a simple analytical lighting model is applied to each intersection of ray and cell. Our algorithm is simple, and our implementation fast and robust.

1 Introduction

Direct volume rendering methods are used to visualize scalar and vector fields by modeling the volume as "cloud-like" cells composed of semi-transparent material that emits its own light, partially transmits light from other cells, and absorbs some incoming light [16, 8, 7]. By varying the lighting model, one can also visualize isosurfaces using direct volume rendering methods.

In the visualization of datasets coming from computational fluid dynamics and partial differential equation solvers, a special and hard-to-render type of grid arises, the so called *irregular grid* (or *mesh*). These grids do not necessarily have uniform cubes, and they have been proposed as an effective means of representing *disparate* field data. Irregular grid data comes in several different formats [12, 14]. One very common format has been *curvilinear grids*, which are *structured* grids in computational space that have been "warped" in physical space, while preserving the same topological structure (connectivity) of a regular grid. However, with the introduction of new methods for generating higher quality adaptive meshes, it is becoming increasingly common to consider more general *unstructured* (non-curvilinear) irregular grids, in which there is no implicit connectivity information. Furthermore, in some applications *disconnected* grids arise.

Rendering of irregular grids has been identified as an especially important research area in visualization [3]. The basic problem consists of evaluating a volume rendering equation [7] for each pixel of the image screen. To do this, it is necessary to have, for each line of sight (ray) through an image pixel, the sorted order of the cells of the mesh along the ray. This information is used to evaluate the overall integral in the rendering equation.

Several works have already presented efficient methods to render irregular grids (*e.g.*, [11, 14, 18, 6, 15, 5, 10]). In this paper, we describe in detail a simple, yet powerful, and robust ray caster for general irregular grids. Our code uses ideas from previous techniques (in particular [1, 13]), and the end product of our work is a minimal set of C++ classes that run on most operating systems and can be used to render general irregular grids. We describe in detail our overall design, the data structures, common robustness problem-

s (and solutions), and the performance, both in rendering time and memory consumption of our code.

Our main goal in this work was to develop a simple to implement, yet fast and robust rendering algorithm. Speed, software portability, and coding simplicity were given the priority over memory efficiency. In a nutshell, we wanted to develop a piece of code that could be easily integrated into current software packages as to bring a state-of-the-art scientific visualization technique closer to the computational scientist.

2 Rendering Algorithm

Here, we describe our technique as it applies to tetrahedral grids. The extension to other cell-types is fairly simple. The basic idea is very simple, and similar to the work of [17, 2, 4]. Instead of rendering cells, we render their faces (or triangulation of their faces, in the case of general cell data). We use ray casting for our rendering. The actual depth-sorting is based on the implicit ordering provided by the connectivity information in the cells (similar to the method proposed in [1]).

2.1 View-Independent Preprocessing

Our method needs minor preprocessing, which basically performs a connectivity reconstruction from the source data. The input dataset is a collection of shared-vertex cells. As the input dataset is read and parsed, all points and tetrahedra are read in and for each input tetrahedron all four of its triangles are enumerated. It is important to avoid duplicate triangles, which represent connectivity between cells.

This can be performed efficiently by keeping a `referredBy` list of triangles in each vertex, which basically lists all triangles that use that vertex. As each triangle is read, we update the `referredBy` list of its vertices. In order to avoid triangle duplication, we simply search the `referredBy` list of its vertices. When searching for a given triangle, it is only necessary to make pointer comparisons among its vertices. In our implementation, we use a simple linear search in the `referredBy` list, which is quite fast in practice, since the degree of each vertex is usually low.

2.2 World-to-Screen Transformation

Our rendering algorithms performs all its calculations in screen space. For each frame, we rotate the whole scene (transform all points in the scene by the rotation matrix) to make the x and y axes in world coordinate system parallel to the x and y axes in screen coordinate system with z axis pointing away from the viewpoint. This greatly simplifies the transformations between screen and world coordinates, and reduces the problem of determining if a 3D ray intersects a 3D triangle to the problem if a given (x, y) point is within a planar triangle on the xy plane. It is much more time efficient to rotate all points at once at this step than to perform the same coordinate rotation within the main loop, several times for each point.

After the rotation is performed, we calculate plane and interpolation coefficients for each triangle and store them within the triangle's structure. Here, we sacrifice memory, but it greatly improves the overall speed during rendering.

2.3 Visibility Determination

With the scene in screen-space, it is time to determine which triangles belong to the visible side of the scene boundary. These necessarily form a superset of the triangles first hit by rays. It is simple to find these triangles, since, in general, a triangle belongs to the boundary if it has only one tetrahedron in its `referredBy` list. Also, for a given boundary triangle to be visible the fourth point of its base tetrahedron should lay on the other side of triangle plane further along the z axis.

In order to determine all ray-triangle intersections, for all visible boundary triangles we loop over all pixels within this triangle's bounding box in xy plane and determine if a given pixel lays within the triangle boundary. In this case, the given triangle is added to the pixel's list of intersections. This technique lets us efficiently enumerate which boundary triangles intersect a given ray, and in what order. We store a list of these intersection with each pixel on the viewplane. A similar technique is proposed by [2], and this can be seen as a generalization of the z-buffer approach of [13].

2.4 Pixel-by-Pixel Ray Casting

Actual ray-casting is straightforward (and similar to the one originally proposed in [1]). For each ray we take a triangle t_c with minimal z coordinate from the corresponding list of intersections, through which the ray enters the scene volume. The corresponding tetrahedron T is fetched from the t_c's `referredBy` list. Since t_c is a boundary triangle, there is only one such tetrahedron, on next iterations we have a choice of two tetrahedra and we choose the one which is different from the current one. When we have only one tetrahedron in the current triangle's `referredBy` list, this means that the ray is leaving a connected component of the volume, but it may re-enter it later. In our technique, this is easily catched by fetching the next triangle along the ray from the intersections list. Once we determine T, we choose which of its three other triangles is the *next* one. This is done by performing intersection calculations, finding the z coordinates of intersection points, and choosing

the one which gives the minimal z still larger than the current z coordinate along the ray.

Opacity and color integration is determined from the intersection point of the current ray with t_c, and the triangle that follows, let us assume t_n. We keep updating t_c, and t_n, until the ray leaves the volume, and stop after all boundary triangles have been used.

2.5 Lighting Model

A simple lighting model is used: integration of linearly-interpolated color and opacity values along the ray. Scalar values in the input dataset are shifted and scaled to fit the $[0, 255]$ range. A user-specified piecewise-linear transfer function is read from a file, it specifies the mapping from this range to the set of opacity and RGB values. During ray casting, we calculate the z and interpolated scalar field values of the ray intersection points with t_c and t_n (using the coefficients stored on the third step of preprocessing) and pass these values to the transfer function calculation module which updates the RGB values of the current pixel.

The exact integration formulas follow. The following variables are used:

z_c, $z_n - z$ coordinates of intersection with the *current* and *next* triangles

Δz – distance between z_c and z_n

c_c, c_n – linearly-interpolated color component value in z_c and z_n

o_c, o_n – linearly-interpolated opacity in z_c and z_n

C_c, O_c – accumulated on the previous steps color and opacity values, initially 0

C_n, O_n – updated color and opacity values

Color and opacity are linearly interpolated between their values in z_c and z_n:

$$o(z) = \frac{o_c(z_n - z) + o_n(z - z_c)}{\Delta z}$$

$$c(z) = \frac{c_c(z_n - z) + c_n(z - z_c)}{\Delta z}$$

These linear functions must be integrated from z_c to z_n to obtain O_n, C_n, we also need the opacity value in all intermediate points to use it in color computation:

$$O(z) = O_c + \int_{z_c}^{z} o(z)dz$$

$$C(z) = C_c + \int_{z_c}^{z} c(z)(1 - O(z))dz$$

After taking these integrals analytically we obtain the following values for O_n and C_n:

$$O_n = O_c + \frac{1}{2}(o_c + o_n)\Delta z$$

$$C_n = C_c - \frac{1}{2}(c_c + c_n)(O_c - 1)\Delta z -$$

$$\frac{1}{24}(3c_c o_c + 5c_n o_c + c_c o_n + 3c_n o_n)\Delta z^2$$

2.6 Point-Within-Triangle Algorithm

In order for all ray-casting algorithm to work fast, an efficient, and numerically stable 2D Point-Within-Triangle primitive is necessary. This primitive is used in two critical places. First, when determining the rays intersecting boundary triangles, and also when looking for the next triangle within the current tetrahedron. Our approach to this problem is described here.

Given a triangle ABC in xy plane and a 2D point Q, we have to determine if Q lays within ABC. To do this we first move origin to A, introducing $\vec{p_1} = \vec{B} - \vec{A}$, $\vec{p_2} = \vec{C} - \vec{A}$, $\vec{q} = \vec{Q} - \vec{A}$.

Decompose $\vec{q}$ by $\vec{p_1}$ and $\vec{p_2}$: $\vec{q} = q_1\vec{p_1} + q_2\vec{p_2}$.

The point will be within the triangle $\iff$ $q_1 \geq 0$ and $q_2 \geq 0$ and $q_1 + q_2 \leq 1$.

q_1 and q_2 are determined from the solution of a linear system:

$$\begin{cases} \vec{p_1} \cdot \vec{q} &= q_1\,\vec{p_1} \cdot \vec{p_1} + q_2\,\vec{p_1} \cdot \vec{p_2} \\ \vec{p_2} \cdot \vec{q} &= q_1\,\vec{p_1} \cdot \vec{p_2} + q_2\,\vec{p_2} \cdot \vec{p_2} \end{cases}$$

This system was solved analytically; the solution gives the following formulas to check if the point is within triangle:

$$\begin{array}{rcl} denom & = & p_1^x p_2^y - p_2^x p_1^y \\ q_1 & = & (q^x p_2^y - q^y p_2^x)\,/denom \\ q_2 & = & (q^y p_1^x - q^x p_1^y)\,/denom \\ In? & \iff & q_1 \geq 0 \wedge q_2 \geq 0 \wedge (q_1 + q_2) \leq denom \end{array}$$

This assumes than the denominator $denom$ is positive; it can always be made positive by swapping $p_1 \leftrightarrow p_2$. Since $denom$ does not depend on the point Q, it can be precomputed and stored with the triangle. We also store the values of $p_{1,2}^{x,y}$. Having those actual arithmetics performed for each point Q is five additions/subtractions, four multiplications and three comparisons with no divisions. On most modern microprocessor, both multiplications, additions and subtractions take between one and two clock cycles (sustained), while divisions take anywhere from 20 to 40 clock cycles. These computations use double precision to maintain accuracy.

2.7 Handling Degeneracies

The Point-Within-Triangle algorithm described above behaves well from the numerical accuracy point of view, but when looking for the *next* triangle a rare situation can arise when we can not choose between several triangles or can not find the one which advances z_n. It happens if a ray hits a mesh node. In this case, the ray-caster program tries to find the *next* triangle from a broader set of all faces of all tetrahedra adjacent to the current one. This helps in the vast majority of cases, in our experiments with real datasets not more than about seven points on a 512×512 raster can still have problems. We can detect these degeneracies, and apply custom solutions. The simplest is to apply local averaging of the pixel values in the neighborhood of the degenerate pixel, and simply assign such average to the bad pixel. A more complex (and expensive) solution is to apply supersampling.

3 Performance Results

The test runs were performed on an SGI Power Challenge machine, equipped with 16 R10000 195MHz processors, and 3GB of RAM. Only one of the processors was used during benchmarking. The whole program is written in C++ as a collection of reusable and extendable classes. It was compiled with `-O3` flag using the native SGI C++ compiler in either 32 or 64 bit mode. Here we report the execution time and memory consumption for the raytracing of several well-known datasets (see Table 1). Figures 1–4 show some typical images generated with our algorithm.

We report in Table 1 run times and the amount of allocated memory (in MB) at different resolutions and different number of active pixels. Total run time is further subdivided into first reading/preprocessing stage (Prep. I) performed once per dataset, second preprocessing stage (Prep. II) performed once per frame for several frames corresponding to different view angles and actual ray-casting stage (R.C.). (See Table 2.) We also include results for runs with the 32-bit version, which is more memory-effective, and 64-bit versions, which is much faster.

Our rendering times are very fast, comparable to the ones reported in [18], where special hardware support was used. Our memory consumption is reasonably high, in fact, we use over an order of magnitude more memory as compared to [11]. On the other hand, we are about twice as fast as the Lazy-Sweep Ray Casting algorithm. It was very surprising to see we can achieve a two-fold speedup by only changing the compilation flags from -32 to -64 on the SGI compiler (version 7.1).

Dataset	Points	Tetrahedra	Triangles
Combustion Chamber	47025	215040	437888
Liquid Oxygen Post	109744	513375	1040588
Blunt Fin	40960	187395	381548
Delta Wing	211680	1005675	2032084

Table 1. A list of the datasets used for testing. These were converted into tetrahedra from the original curvilinear datasets provided by NASA.

4 Conclusions

We have presented a simple (only 1700 lines of C++ code, including comments and internal debugging code), reusable, extendable, straightforward and robust implementation of an irregular grid ray-caster. Our algorithm makes no use of hierarchical data structures, or other complicated techniques. Despite its simplicity, this raycaster met all our design goals. It achieves an impressive performance when rendering irregular grid datasets. Our future plans include adding a graphical user interface, with support for progressive rendering for interactive exploration. Another direction is parallelizing our technique, which should be relatively easy for a shared memory machine like SGI Challenge or multiprocessor Linux workstation.

Electronic Information

The full source code of our algorithm, and a PERL script for automatic generation of transfer functions by histograming is available. This, and other information, including images and movies can be obtained from `http://pbunyk.physics.sunysb.edu/~paul/CSE/RayTracer` or by sending mail to the authors.

Acknowledgments

NASA has gracefully provided the Blunt Fin, Liquid Oxygen Post, and Delta Wing datasets. The Combustion Chamber dataset is from the Visualization Toolkit (Vtk). We thank D. Zinoviev and P. Shevchenko for their idea of Point-Within-Triangle algorithm. This project has been partially supported by National Science Foundation grant MIP-9527694, and by grants from NASA Ames, the Office of Naval Research grant N000149710402 , Sandia National Labs and the Dept of Energy Mathematics, Information and Computer Science Office.

References

[1] M. P. Garrity, "Raytracing Irregular Volume Data," *Computer Graphics (San Diego Workshop on Volume Visualization)*, vol. 24, pp. 35–40, Nov. 1990.

[2] L. Hong and A. Kaufman, "Accelerated Ray-Casting for Curvilinear Volumes,". *IEEE Visualization '98*, pp. 247–253, 1998.

[3] A. E. Kaufman, K. H. Höhne, W. Krüger, L. Rosenblum, and P. Schröder, "Research Issues in Volume Visualization," *IEEE Computer Graphics and Applications*, vol. 14, no. 2, pp. 63–67, March 1994.

[4] B. Lucas. "A Scientific Visualization Renderer," *IEEE Visualization '92*, pp. 227–234, 1992.

[5] K-L. Ma, "Parallel Volume Rendering for Unstructured-Grid Data on Distributed Memory Machines," *Proc. IEEE/ACM Parallel Rendering Symposium '95*, pp. 23–30, 1995.

[6] X. Mao, L. Hong, and A. Kaufman, "Splatting of Curvilinear Grids," *IEEE Visualization '95*, pp. 61–68, 1995.

[7] N. Max, "Optical Models for Direct Volume Rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99–108, June 1995.

[8] N. Max, P. Hanrahan, and R. Crawfis, "Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions," *Computer Graphics (San Diego Workshop on Volume Visualization)*, vol. 24, pp. 27–33, Nov. 1990.

[9] S. Ramamoorthy and J. Wilhelms, "An Analysis of Approaches to Ray-Tracing Curvilinear Grids," Tech Report UCSC-CRL-92-07, U. of California, Santa Cruz, 1992.

[10] C. Silva, "Parallel Volume Rendering of Irregular Grids," Ph.D. thesis, Department of Computer Science, State University of New York at Stony Brook, 1996.

[11] C. Silva and J. Mitchell, "The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids," *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, no. 2, June 1997.

[12] D. Speray and S. Kennon, "Volume Probes: Interactive Data Exploration on Arbitrary Grids," *Computer Graphics (San Diego Workshop on Volume Visualization)*, vol. 24, pp. 5–12, Nov. 1990.

[13] S. Uselton, "Volume Rendering for Computational Fluid Dynamics: Initial Results," Tech Report RNR-91-026, Nasa Ames Research Center, 1991.

[14] J. Wilhelms, "Pursuing Interactive Visualization of Irregular Grids," *Visual Computer*, vol. 9, no. 8, 1993.

[15] J. Wilhelms, J. Challinger, N. Alper, S. Ramamoorthy, and A. Vaziri. "Direct Volume Rendering of Curvilinear Volumes," *Computer Graphics (San Diego Workshop on Volume Visualization)*, vol. 24, pp. 41–47, Nov. 1990.

[16] J. Wilhelms and A. Van Gelder, "A Coherent Projection Approach for Direct Volume Rendering," *Computer Graphics (SIGGRAPH '91 Proceedings)*, vol. 25, pp. 275–284, July 1991.

[17] J. Wilhelms, A. Van Gelder, P. Tarantino, and J. Gibbs. "Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids," *IEEE Visualization '96*, pp. 57–64, 1996.

[18] R. Yagel, D. Reed, A. Law, P-W. Shih, and N. Shareef, "Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing," *IEEE-ACM Volume Visualization Symposium*, pp. 55–62, Nov. 1996.

Dataset	Res.	Active	Bits	Prep. I (sec)	Prep. II (sec)	R.C. (sec)	Mem. (MB)
Cobution Chaber	128^2	5032	32	14	1	4	87.7
			64	15	1	2	117.2
	256^2	20234	32	15	2	10	88.8
			64	16	1	6	118.6
	512^2	81544	32	14	1	37	93.1
			64	15	1	18	124.1
	1024^2	327272	32	15	1	141	110.4
			64	15	2	65	146.4
Liquid Oxygen Pot	128^2	6254	32	36	2	5	208.3
			64	36	2	3	278.1
	256^2	25160	32	36	3	19	209.4
			64	36	3	10	279.6
	512^2	101034	32	37	2	72	214.1
			64	38	3	35	285.7
	1024^2	405054	32	35	2	271	232.8
			64	36	3	121	309.8
Blunt Fin	128^2	4453	32	14	1	2	76.5
			64	15	1	2	102.1
	256^2	17858	32	14	1	8	77.5
			64	15	1	4	103.5
	512^2	71508	32	14	1	27	81.7
			64	14	1	15	108.8
	1024^2	286781	32	14	1	104	98.3
			64	15	2	56	130.2
Delta Wing	128^2	4396	32	72	4	4	406.6
			64	76	5	3	542.9
	256^2	17684	32	74	4	13	407.6
			64	78	5	9	544.2
	512^2	71062	32	70	4	43	411.7
			64	76	5	31	549.6
	1024^2	284889	32	71	4	157	428.3
			64	76	5	100	570.9

Table 2. Vital statistics for using our rendering algorithm on a R10000 processor.

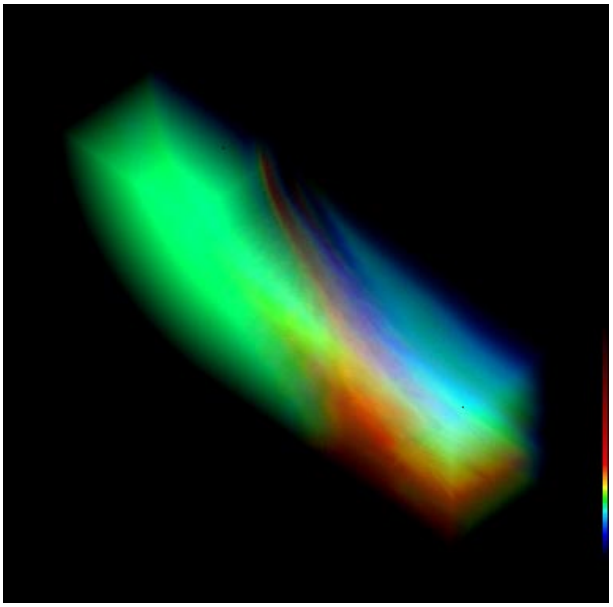

Figure 1. Blunt Fin.

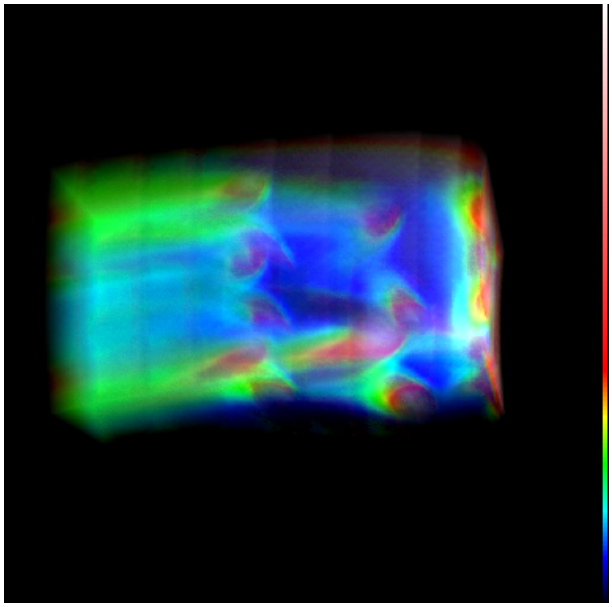

Figure 2. Combustion Chamber.

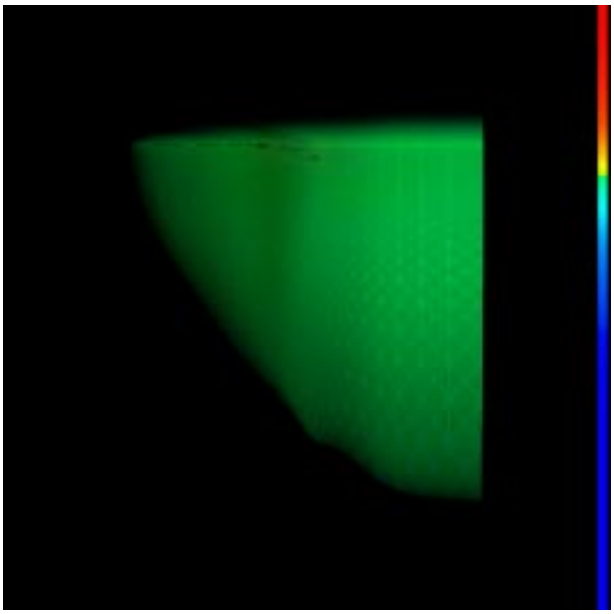

Figure 3. Delta Wing.

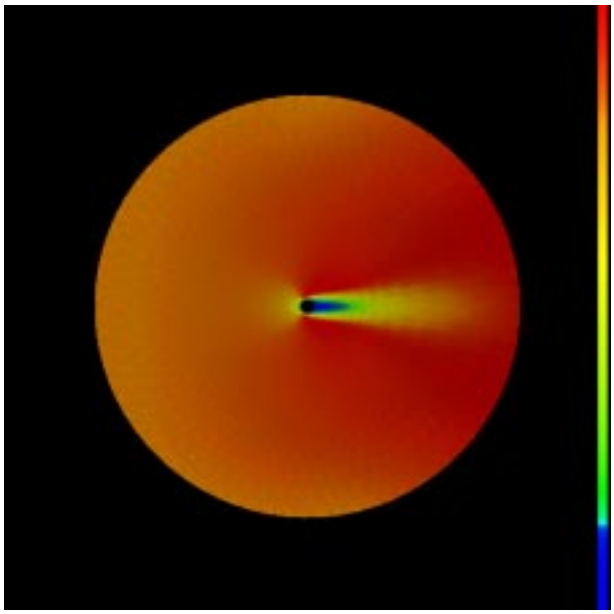

Figure 4. Liquid Oxygen Post.