LBL-25274 c.2

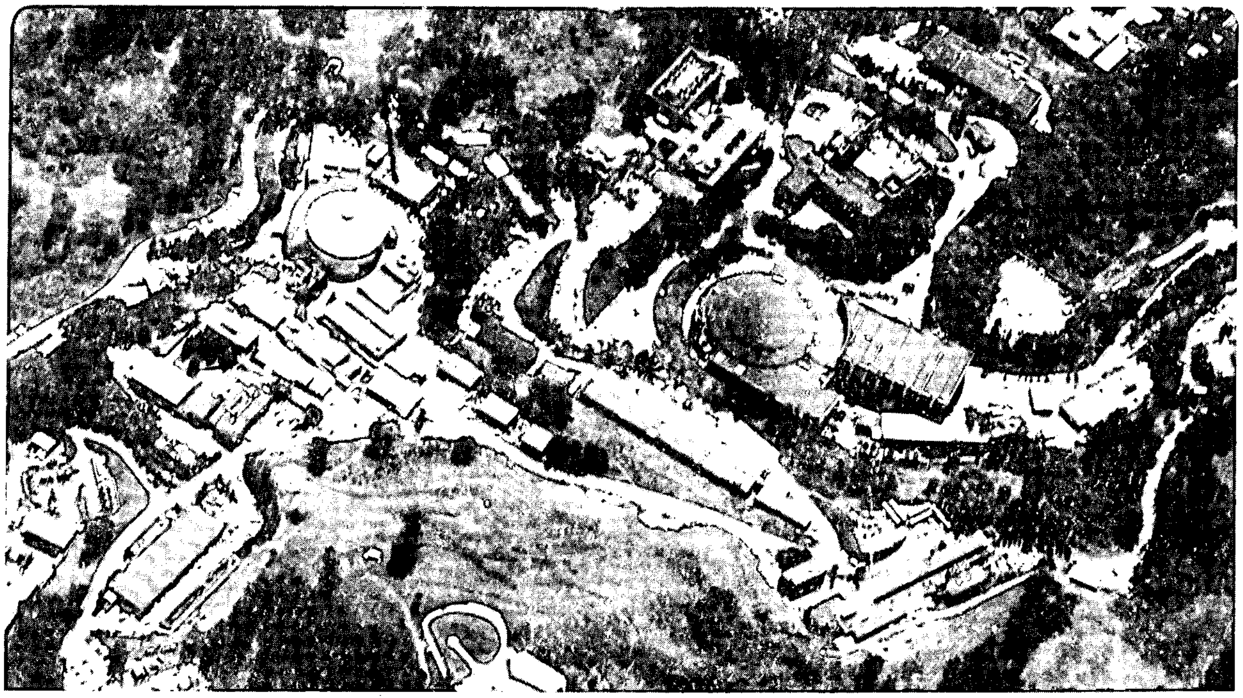
Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA, BERKELEY

Information and Computing Sciences Division

Use of a Distributed Movie-Making System for Presentation of Fluid Flow Data

D.W. Robertson
(M.S. Thesis)

May 1988



LBL-25274 c.2

Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

May 6, 1988 LBL-25274

Use of a Distributed Movie-Making System for Presentation of Fluid Flow Data

David W. Robertson

Advanced Development Projects
Information and Computing Sciences Division
Lawrence Berkeley Laboratory
1 Cyclotron Road
Berkeley, CA 94720

The author can be reached via USMail at: Lawrence Berkeley Laboratory, Bldg. 50B, Rm. 3238, Berkeley, CA 94720. His Email address is: davidr@csam.lbl.gov

The work presented in this paper is supported by the U.S. Department of Energy under contract DE-AC03-76SF00098. Any conclusions or opinions, or implied approval or disapproval of a company or product name are solely those of the authors and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory, or the U.S. Department of Energy.

USE OF A DISTRIBUTED MOVIE-MAKING SYSTEM FOR PRESENTATION OF FLUID FLOW DATA

A thesis submitted to the faculty of
San Francisco State University
in partial fulfillment of the
requirements for the
degree

Master of Science
in
Computer Science

by

DAVID W. ROBERTSON

San Francisco, California

May, 1988

USE OF A DISTRIBUTED MOVIE-MAKING SYSTEM FOR PRESENTATION OF FLUID FLOW DATA

DAVID W. ROBERTSON
San Francisco, California
1988

This thesis describes work done to develop techniques for the graphical presentation of time-dependent 2- or 3-dimensional flow fields generated by fluid modelling together with a distributed computing software architecture to produce the graphics. The software runs on UNIX systems and uses interprocess communication with a TCP/IP based network protocol to implement the architecture. The system components provide for communication between a video workstation consisting of a microcomputer controller, a frame buffer, a video recording device, and a front end system. The use of particle advection to graphically display the flow fields proper was investigated. The associated numerical calculations for transforming the data into graphical form are described, as are the relative merits and problems involved with this type of graphical display. The representation of the object(s) perturbing or constraining the flow is an important topic in itself, and issues involved in the implementation of features of the software which affect both the display of the object(s) and the flow field are discussed. An analysis is given of the use of video recording and playback as the final manner of presentation for a given data set.

ACKNOWLEDGEMENTS

I thank William E. Johnston for his guidance in helping to produce this thesis, and the past and current members of the Advanced Development Group at Lawrence Berkeley Laboratory for their help. I thank Donald and Margaret Robertson for their support.

This work is supported by the U.S. Department of Energy, under Contract No. DE-AC03-76SF00098 with the Lawrence Berkeley Laboratory, University of California.

Table of Contents

List of Tables

List of Figures

List of Appendices

Introduction

Fluid modelling attempts to find a numerical mathematical description of complex flow patterns in a region of interest, such as the flow of air past an airplane wing. One result of such modelling is a flow field. A flow field is a grid laid out over a given area, with the x, y, and in the case of 3-dimensional modelling, z components of the velocities of the flow being given at each grid point. Since the motion of the fluid changes with time, the result of the modelling is not one, but a sequence of such flow fields. The amount of data involved rapidly becomes difficult to comprehend, or even manage, as the number of grid points increases. For example, 300 64x64x64x3 3D flow fields (numbers stored as floats) contain close to 1 gigabyte of information.

The focus of this thesis is gaining the ability to visualize significant features of 2- and 3-dimensional (2D and 3D) flow by converting the data describing the flow fields into a sequence of graphical representations, such as the simulation of the motion of dye tracers being carried along by the fluid. A number of researchers have used computer graphics to better understand large complex data sets [15, 24, 45]. Computer graphics has already been used in the field of fluid modelling [4, 19, 22, 41, 45, 47]. My aim was to further explore its use in this field given the constraints and advantages of a certain class of "video workstation" [20].

The video workstation consists of an PC-type microcomputer, a 16-bit frame buffer, and a video recording device. To be able to study flow visualization, it was necessary to develop much underlying software. The hardware constraints dictated distributed graphics software that relegated the role of the video workstation to that of a graphics/animation server, with the numerical calculations involved in converting flow data to a graphical form and the rendering of the boundary being performed in a client program on a remote UNIX system. The design of a network distributed application was driven by the need to perform the computationally intensive tasks on a powerful-enough CPU, and to minimize the amount of data movement over the network.

A number of problems had to be addressed during the development of the graphics software used to visualize flow fields. These included how to transform the data into graphical form, how to represent the geometry of boundaries constraining the flow, how to display complex data in a clear and unambiguous fashion, and how to generate the results in a timely manner. A considerable part of the effort for this thesis went into developing rendering software that could be used to remove hidden surfaces of, and shade and scan convert the boundary of the region of interest in the 3D case.

Another issue that had to be addressed was the final presentation of the data. The graphical representations of the sequence of flow fields have to be displayed fairly rapidly to gain a better understanding of what is being modelled. The experience gained in using video recording on videotape and videodisk to achieve this rapid display is described.

Chapter 1

Visualizing Flow Dynamics

1.1. Approaches to Graphical Presentation

The information in a flow field has been graphically represented in many ways. It has been converted into a scalar form such as pressure, density, or energy at each grid point and displayed in the two-dimensional (2D) case as contour lines along which the magnitude of the variable is constant. The spacing of the contours easily indicates the values of the gradient [4] [Fig. 1.1]. However, in displaying a sequence of flow fields the gradient changes from frame to frame, giving the spurious effect of movement of contour lines perpendicular to themselves. If a continuous color coding of the magnitude is used [41] [Appendix C, Fig. C1.1][1], this distracting perpendicular movement, which has no physical meaning, is not as noticeable. Another distraction occurs if the difference in magnitude between adjacent levels is too small, in which case lines jump in and out of existence in regions where there is a small gradient [15].

[1]All figure references of the form [Appendix C, Fig. C#.#] are contained in a supplementary videotape to this thesis. Information on availability of the videotape is given in Appendix C.

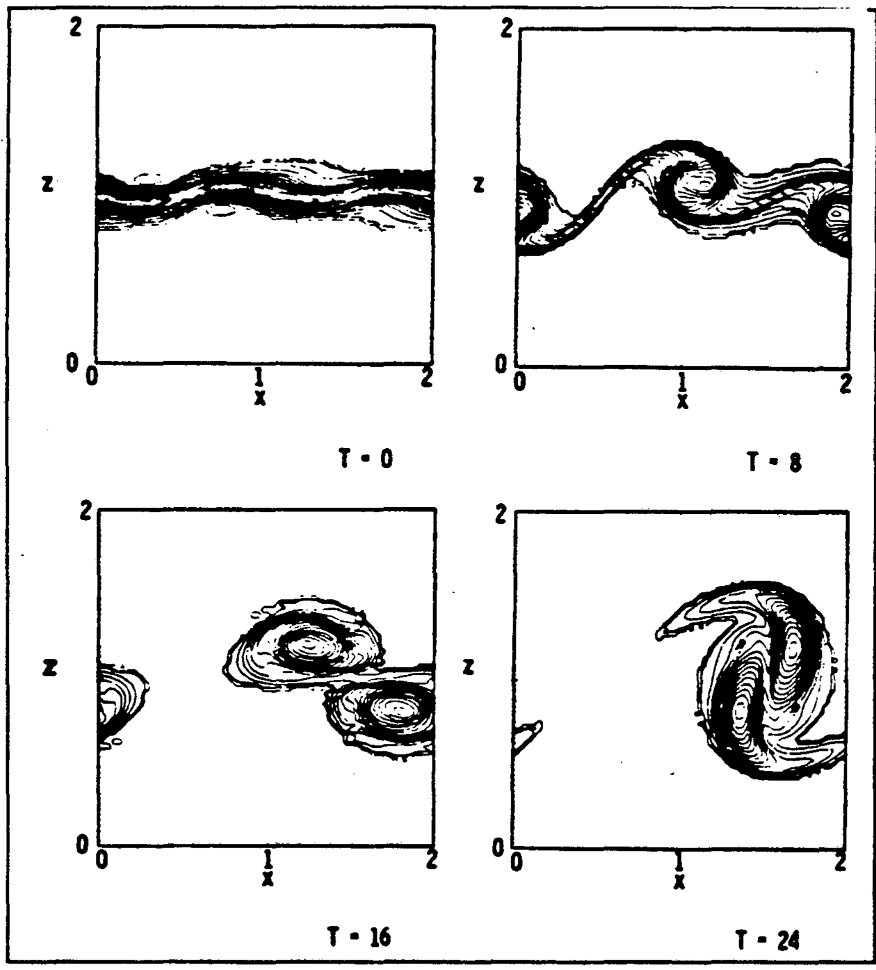


Fig. 1.1. "Simulation of the turbulent behavior of a shear layer. [Furnished by J. Riley and R. Metcalfe]." Reprinted, by permission, from Peyret and Taylor [33], 4.

In the three-dimensional (3D) case multiple parallel planes through the volume of interest have been displayed, with each plane having a continuous color coding of the magnitude [Fig. 1.2]. Another method for representation of scalar quantities is to display surfaces where the variable is constant. When displaying multiple surfaces, transparency has been used to see hidden surfaces [47]. In either case only a small number of slices or surfaces can be usefully displayed, missing 3D aspects of the flow such as large gradient changes [4].

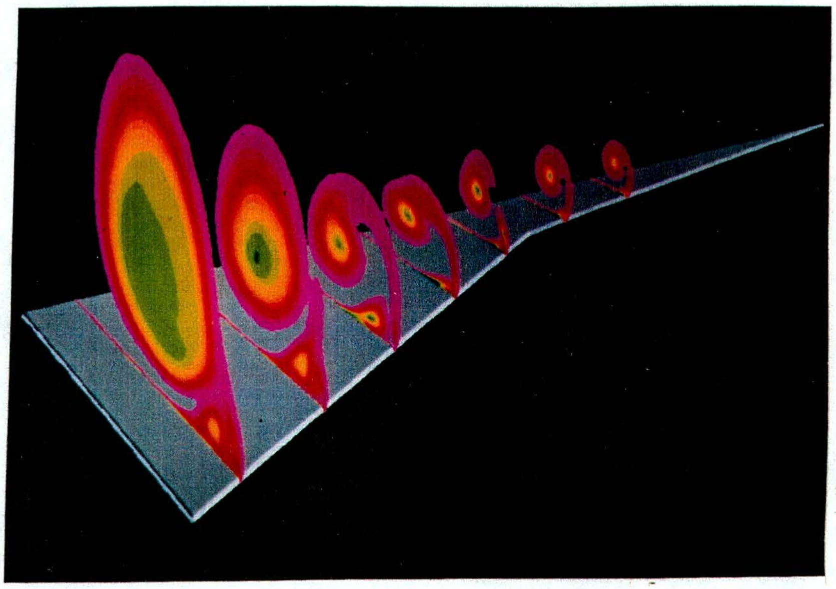


Fig. 1.2. "Vortex breakdown over a strake-delta wing. Contours are constant values of normalized stagnation pressure. (Computer simulations by Fujii)." Courtesy V. Watson, figure from Watson, et al. [47], 10.

BBC 880-11633

Two other approaches to displaying flow fields are direct display of vectors at points in the grid, and particle advection [4, 15, 47]. Both are well suited to animated display (either real time or movie style playback). Displaying vectors requires showing their magnitude (usually color coded) and their direction. In the 2D case, this method has been used to derive features of the flow such as flow separation. Displaying 3D flow vectors is more difficult. If more than one plane through the 3D grid is shown, it is difficult to pick out the separate layers from the jumble of vectors [4].

Particle advection, the approach explored in this thesis, has been widely used by researchers [4, 15, 19, 41, 45, 47]. It simulates the motion of dye tracers dropped into a fluid flow and carried along by it. This simulation reveals features of the flow such as "vortices, shock fronts, boundary layers, and flow separation" [45].

1.2. Particle Advection

Simulating the motion of a tracer particle requires advecting it through the flow fields. In this process the position of a particle at one time step is used to find the flow-field grid cell containing the particle's location. The velocity components at each vertex of the enclosing grid cell are interpolated to that location. The resulting velocity is used to locate the particle position at the next time step. This process is repeated for each time step until the particle leaves the modelled region or until the end of the period for which flow fields were found.

A numerical method is used to approximate the particle's position at the next time step. This involves approximating the solution to the following differential equations:

$$\frac{dx}{dt} = u(x,y,t) \qquad (1.1)$$

$$\frac{dy}{dt} = v(x,y,t)$$

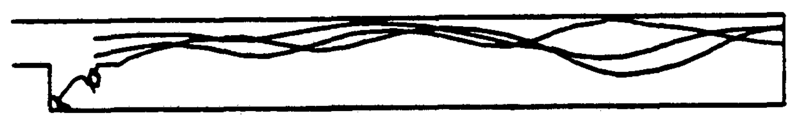
where u(x,y,t) is the x velocity of the particle interpolated to the location (x,y) from the velocities at the enclosing grid points at time t. Similarly, v(x,y,t) is the y velocity of the particle interpolated to the location (x,y) at time t [40].

1.2.1. Numerical Calculation - 2 Dimensional Case

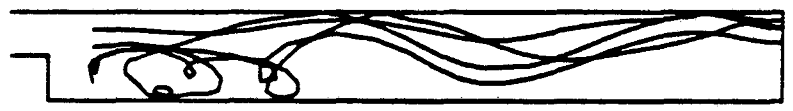
The first study case was a 2D flow field resulting from modelling of flow over a backward facing step [41]. Several numerical methods were tried for approximating the solutions to the above equations, namely Euler's, the modified Euler, and the Runge-Kutta method. The Runge-Kutta method is the most accurate of the three, but also the most complex [5]. Several particles were advected through the flow field using each method, and the resulting series of positions traced by each particle were displayed. [Fig. 1.3].

Fig. 1.3. Four particles injected. Time step = 0.5 s.

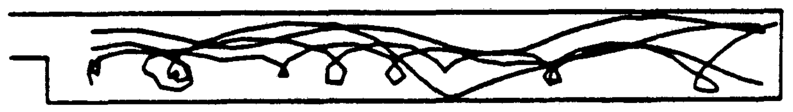
Flow is left to right, entering through the small opening.



(a) Euler's method.



(b) Modified Euler's method.



(c) Runge-Kutta method.

The first step in advecting particles is to bilinearly interpolate between grid points to find the x and y velocity components at a particle location, i.e.,

$$\begin{aligned}
x_\alpha &= x_{par} - x \\
x_{bottom} &= \left(1 - x_\alpha\right) x_{botleft} + x_\alpha x_{botright} \\
x_{top} &= \left(1 - x_\alpha\right) x_{topleft} + x_\alpha x_{topright} \\
y_\alpha &= y_{par} - y \\
x_{interp} &= \left(1 - y_\alpha\right) x_{bottom} + y_\alpha x_{top}
\end{aligned} \tag{1.2}$$

where x and y are the indices of the lower left vertex of the cell, x_{par} is the x coordinate of the particle position, $x_{botleft}$ is the x velocity component at the lower left vertex, $x_{botright}$ at the lower right, $x_{topleft}$ at the top left, and $x_{topright}$ at the top right. x_{bottom} is the intermediate result for the bottom side, x_{top} the intermediate result for the top, and x_{interp} is the final interpolated velocity component. A similar method is used for the y velocity component.

Once the velocity components are found, a numerical method is then used to find the location of the particle at the next step. Euler's method is as follows:

$$\begin{aligned}
x_0 &= \alpha \\
x_{i+1} &= x_i + \delta t \cdot x_{vel}
\end{aligned} \tag{1.3}$$

where α is the x coordinate of the initial injection point, δt is the time step, and x_{vel} is the x component of the velocity as arrived at by bilinear interpolation. The new y position is calculated similarly. The implementation of Euler's and the following methods is slightly different from that given in Burden and Faires [5] because $\frac{dx}{dt} = u(x,y,t)$ is already given at each grid point. Interpolation is all that is necessary to get the velocity x_{vel}.

The less accurate nature of Euler's method is seen by comparing the tracks left using each method [Fig. 1.3]. Using Euler's method, the structure of the tracks is more gross, with less loops traced indicating less capturing of particles by vortices.

The Modified Euler method [5] is as follows:

$$\begin{aligned} x_0 &= \alpha \\ x_{interm} &= x_i + \delta t \cdot x_{vel} \\ x_{i+1} &= x_i + \frac{\delta t}{2} \cdot \left(x_{vel} + x_{vxint} \right) \end{aligned} \tag{1.4}$$

where α is the x coordinate of the initial injection point, δt is the time step, x_{vel} is the interpolated velocity at x_i, x_{interm} is the intermediate x coordinate, and x_{vxint} is the interpolated x velocity at x_{interm}. The first step of this method is identical to Euler's and results in the x position x_{interm}. In the second step the velocity is found at x_{interm}, and averaged in with the x velocity found for the original x position. Again, the results are different from the more accurate Runge-Kutta method [Fig. 1.3].

The Runge-Kutta method of order four is as follows:

$$\begin{aligned} x_0 &= \alpha \\ x_1 &= \text{interpolated x velocity at } x_i \end{aligned} \tag{1.5}$$

$$x_2 = \text{interpolated x velocity at } \left(x_i + \frac{\delta t}{2}\right) \cdot x_1$$

$$x_3 = \text{interpolated x velocity at } \left(x_i + \frac{\delta t}{2}\right) \cdot x_2$$

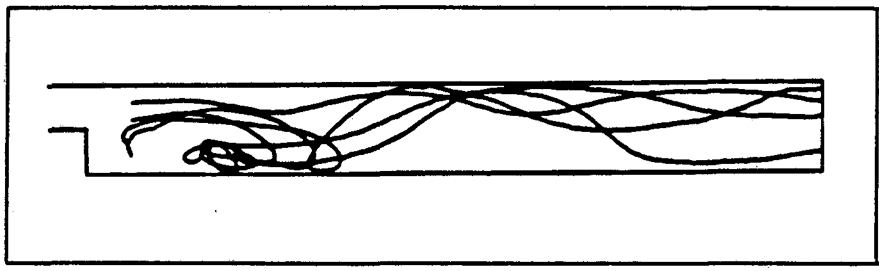
$$x_4 = \text{interpolated x velocity at } \left(x_i + \delta t\right) \cdot x_3$$

$$x_{i+1} = x_i + \frac{\delta t}{6} \cdot \left(x_1 + 2 \cdot \left(x_2 + x_3\right)\right) + x_4\Big)$$

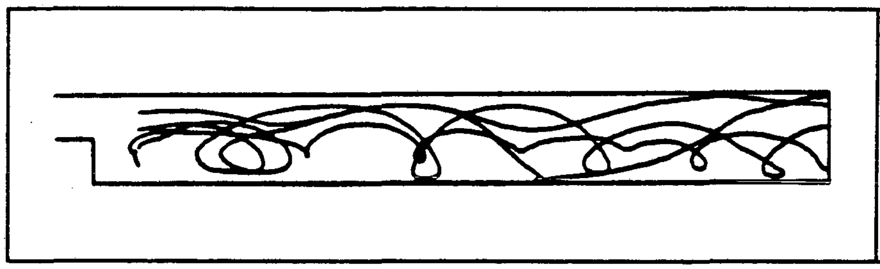
This method requires four more calculations per time step than Euler's, but the accuracy gained makes it worthwhile. The Runge-Kutta method is more accurate using a time step of 4x than the Euler method is using a time step of x [5].

The particle tracks in Fig. 1.3 have a jagged appearance. Using a smaller time step, the simulation of tracer movement is made more visually pleasing owing to the greater accuracy acheived. The smaller the time step, the more accurate the approximation, up to a certain point [5]. Fig. 1.4 shows the result of the same three methods using three interpolated time steps between each modelled one. Interpolation in x and y was performed for two successive time steps, and then interpolation in time between the results. Decreasing the time step further increases the accuracy but would not make much difference visually due to the limited resolution of the display.

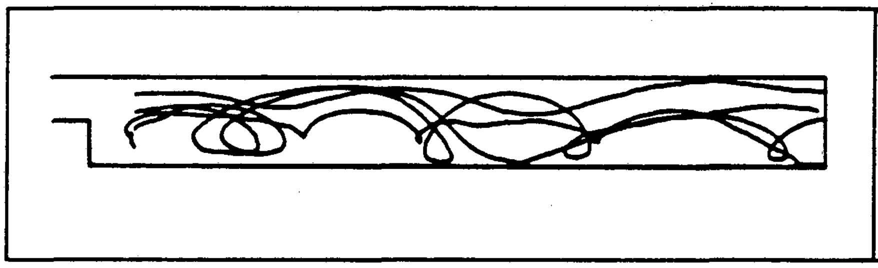
Fig. 1.4. Four particles injected. Time step = 0.125 s.



(a) Euler's method.



(b) Modified Euler's method.



(c) Runge-Kutta method.

With larger time steps and/or less accurate methods, particles overshoot the bounds of the flow field at the top and bottom. Since the modelling assumes solid walls at the top and bottom, this is undesirable. However, with a small enough time step and an accurate method this overshooting cannot occur because (1) a particle moves a smaller distance at a time, encountering the region near the wall where the velocity components are small and pointing towards the interior before it overshoots, and (2) a more accurate method uses more than one calculation per time step and "averages" the results of these calculations, correcting for a too-large displacement.

A logistical problem arises with the modelling of many time steps. One hundred ninety-two steps were modelled for the backward facing step, and three additional time steps are interpolated between each modelled one, resulting in almost 800 frames to be calculated and recorded. To be able to start in the middle of the modelled period without having to re-run the simulation up to that point (a fairly lengthy enterprise), the particles' positions and colors are recorded in a checkpoint file after a given number of frames. This saves a great deal of time and allows movies to be made in several pieces. Checkpoint files are also used in the 3D case.

1.2.2. Advection in the 3-Dimensional Case

The sample modelling data in the 3D case was generated on the University of Minnesota Cray 2. Flow through a torus with three chambers was modelled, with the eventual goal of modelling blood flow through the human heart [26, 32] [Fig. 1.5][2].

[2]See Appendix C for availability of color reproductions of photographic prints.

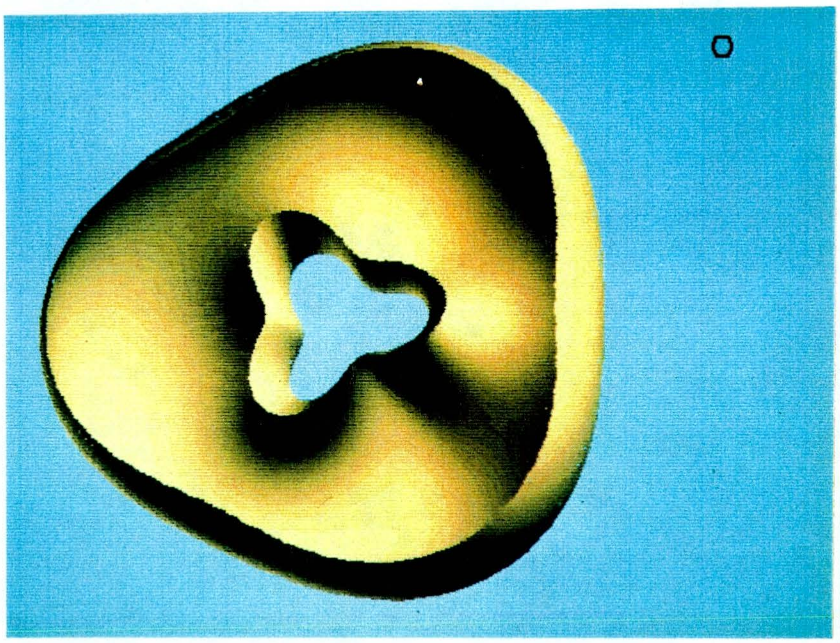


Fig. 1.5. Cut-away of torus with three chambers.

BBC 884-4329

Unfortunately the flow fields consist of 64x64x64x3 or 786,000 floating point numbers per time step, and there are at least several hundred time steps. This large amount of data made transporting the flow fields over a wide-area network with a bandwidth of about 2,000 bytes per second unfeasible. A sample data set consisting of several planes through the 3D flow field, and the boundary data for about 40 time steps, was available at the beginning of the development process.

The flow field data available contained insufficient information for advection. Since the main goal was investigating the graphics issues involved in displaying particle advection through 3-space, a test case was devised. The 2D flow field for the backward facing step [41] is promoted to a 3D field by setting the z velocity components equal to the y components. The rectilinear coordinate space of the flow over the backward-facing step is mapped to the interior of the torus. The geometry of the torus is given by the boundary data for the first time step. As long as particles are injected with the y position sufficiently different from the z position, more than enough 3D, complex motion is generated to test the graphics issues involved.

The Runge-Kutta method is used to advect the particles. The calculations are identical to those of the 2D case, with the addition of the calculation of the z positions at each time step, i.e. an additional step is used with z substituted for y in Equation 1.5. The particles are not transformed into torus space until after advection.

1.2.3. Parameters to Consider in Advecting Particles

Visual as well as numerical issues must be addressed when advecting particles. The goal is to gain insight into the flow through visual means, to present the maximum amount of information without being confusing and showing a "tangled mass of spaghetti" [4]. This "spaghetti" problem was encountered in the case of the backward facing step. The display of even 10 particles leaving their tracks is difficult to understand [Fig. 1.6].

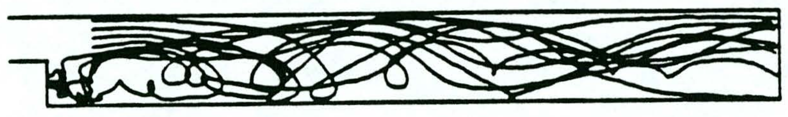


Fig. 1.6. Ten particles injected. Runge-Kutta method. Time step = 0.125 s.

Showing particle tracks has its advantages – the history of the particle over many time steps can be shown, making this device good for static presentations, and a sense of flow is given even when the particles cannot be advected very rapidly.

Unfortunately, unless the tracks are color coded by injection position, only a few intertwining particle trails can be confusing. A partial solution is to erase the portion of a particle track more than a few time steps old. This, in combination with color coding, would gain some leeway in displaying advected particles. If tracks are complex, however, not more than 30 or 40 particles can be usefully displayed [Fig. 1.7].

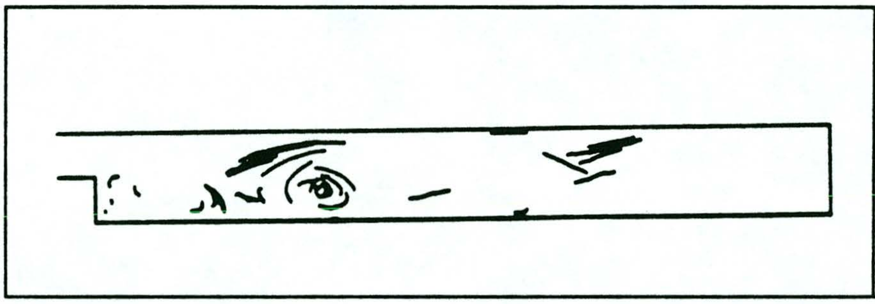
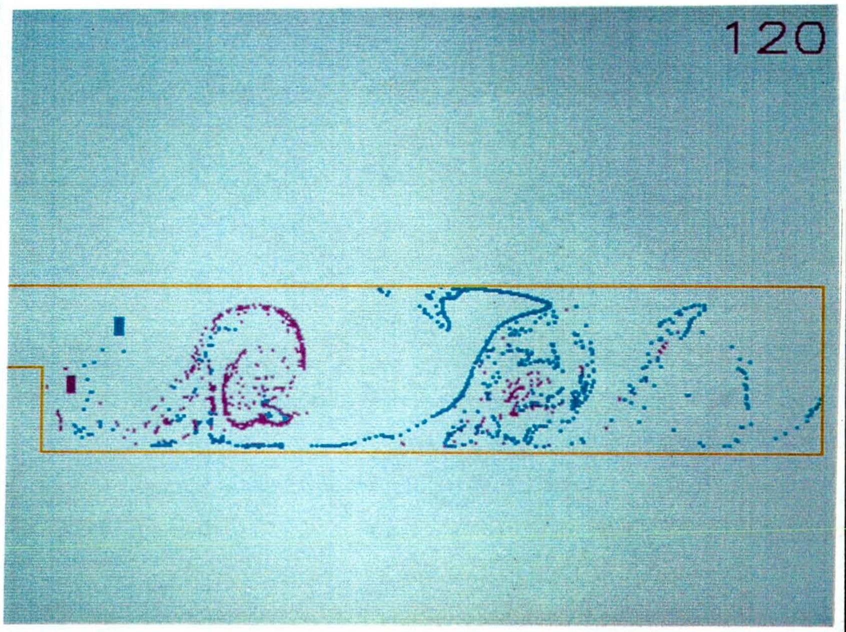


Fig. 1.7. Forty particles injected. Particle tracks are for time = 7.5 s through time = 9.5 s (8 time steps).

If the particles are advected rapidly enough or recorded for video playback, the particle tracks can be eliminated. The result of displaying only the current position of particles at each time step is much like non-soluble colored particles being dropped in an actual flow [45]. A problem with this approach is that the speed of the flow may be too fast to comprehend the advection of the particles properly. In that case each frame can be recorded twice and/or the time step size can be reduced. Reducing the time step size increases the amount of computation since particle positions must be found for each time step.

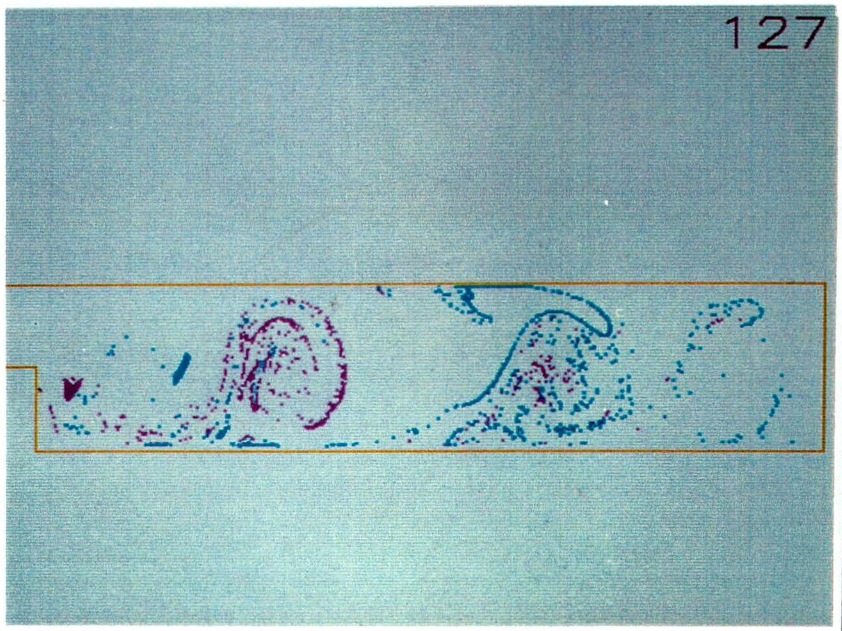
The proper choice of injection point yields much information about the dynamics of the flow. For example, in the case of video of the backward facing step, injection occurs at two points with color coding by location of injection [Fig. 1.8] [Appendix C, Fig. C1.2].

Fig. 1.8. Five hundred particles injected at 2 points.



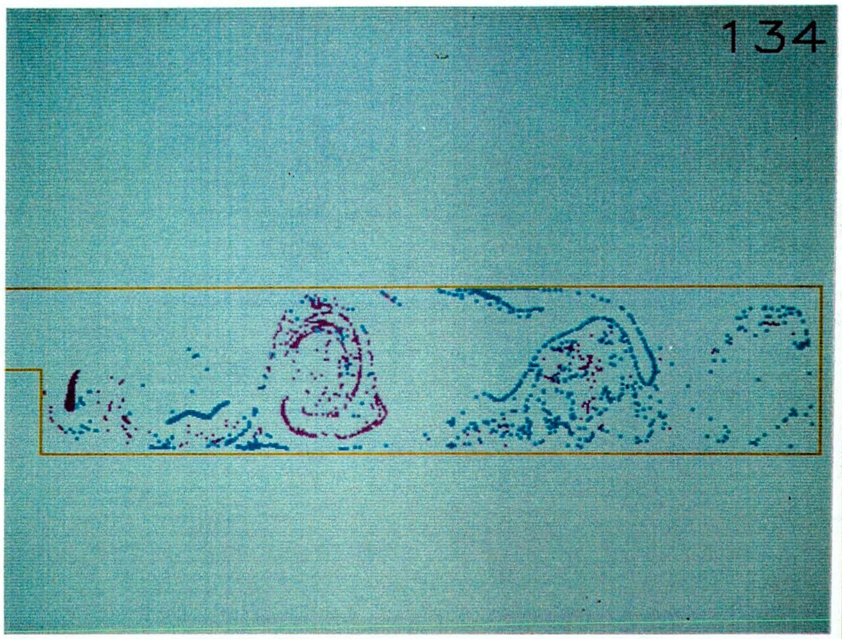
(a) 120th frame.

BBC 884-4301



(b) 127th frame.

BBC 883-2579



(c) 134th frame.

BBC 884-4305

The cyan particles injected in the top region where the flow is more laminar are caught up in turbulence later than the magenta ones.

Four related issues are how often to inject particles, how many to inject, the spacing of the particles at the injection point, and the size of the particles. In the case of the backward facing step, particles continuously exit the region of interest after a quarter of the time steps modelled. If particles are injected only once, too few particles are left in the region of interest to clearly show the structure of the flow in the latter part of the period modelled. Injecting particles at intervals resupplies enough particles to be able to see features of the flow, such as vortices and the stagnation region immediately to the right of the step, that form as quasi steady-state phenomenon [Fig. 1.9] [Appendix C, Fig. C1.2].

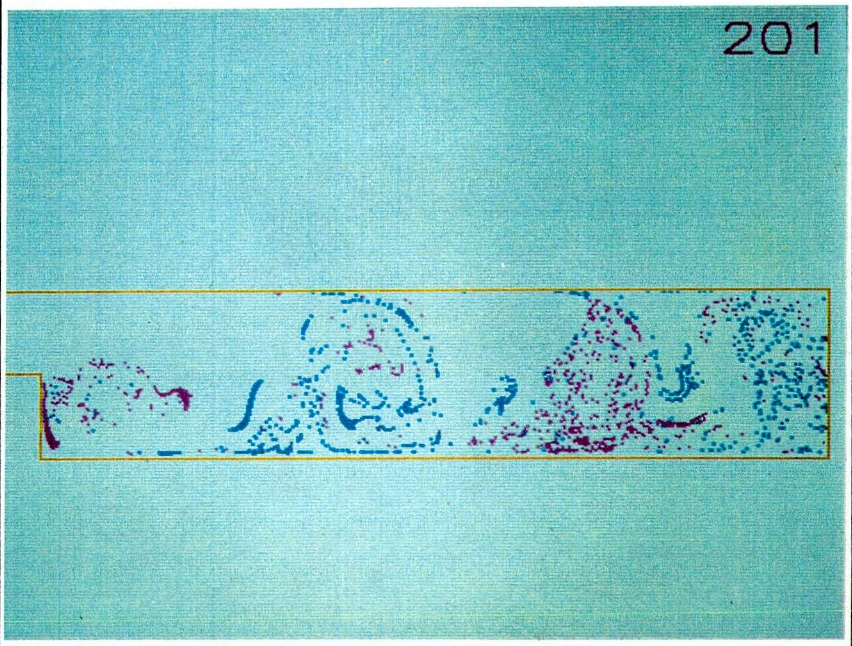


Fig. 1.9. Five hundred particles injected at two injection points. 201st frame. Notice vortices and stagnation region to right of step.

BBC 884-4317

However, if particles are injected too often, the number of particles becomes too large and starts to fill up the modelled region enough to obscure features of the flow. Another reason not to inject too often is that with properly spaced injections, it is possible to see the differences in flow features that arise later in the period modelled even when the injection points are identical. These differences are obscured if injections occur too often.

In the modelling of the backward-facing step, or of flow over objects such as an airplane wing, particles exit the modelled region. In the case of the 3D torus, however, particles are constrained to flow within the interior of the torus. If particles are followed over a period longer than the time for them to complete a circuit of the torus, only a few injections should occur, or there will be too many particles. Another choice, if the focus is on more short-term behavior, is to inject at intervals throughout the period modelled and remove particles after they have made one revolution.

Choosing the number of particles to inject is related to the frequency of injection. If there are too many, features will be obscured. If there are too few, structures will not show up clearly. In both the 2D and 3D cases, 500 to 10000, pixel-sized, colored particles are injected at two locations. An alternative approach in both cases, not used due to time constraints, both programming and computational, is injecting many times more, darkly colored, transparent particles, the result of which looks like smoke in a wind tunnel. With this number of particles, very complex features can be seen [45]. However, a relatively small number of colored particles gives a good idea of what is going on at a fraction of the computational cost. A compromise would be to use the colored particle approach for an exploratory survey of the flow, and the "smoke" approach later for a more detailed visualization.

Choosing the spacing between particles is related to the number of particles injected. Since a small initial difference in position can make a large difference in position later, particles are spaced more closely at the injection point than the resolution of the display. As a result, many particles are displayed at the same pixel location. The more particles, the closer the spacing chosen; otherwise the groups of dye tracers spread out over too great a region of the screen.

The choice of size of the particles is influenced by the fact that frames are eventually recorded for video playback. Video recording uses NTSC encoding for the color, which limits the spatial resolution for many colors [20]. (It should be noted that the photo figures are not NTSC encoded, but are black and white reproductions of a high-resolution RGB signal.) If the particles are too small, the color washes out.

For this reason, particles at least two pixels on a side are used in most cases, making their color clearer.

When thousands of particles are injected near the backward facing step, they are drawn out into long convoluted strings by the flow. In this instance, if particles two pixels on a side are used, detail is lost because the strings are too thick. If interest is more in this region than near the right exit point where the particle strips have broken up, one-pixel sized particles are used [Fig. 1.10] [Appendix C, Fig. C1.3].

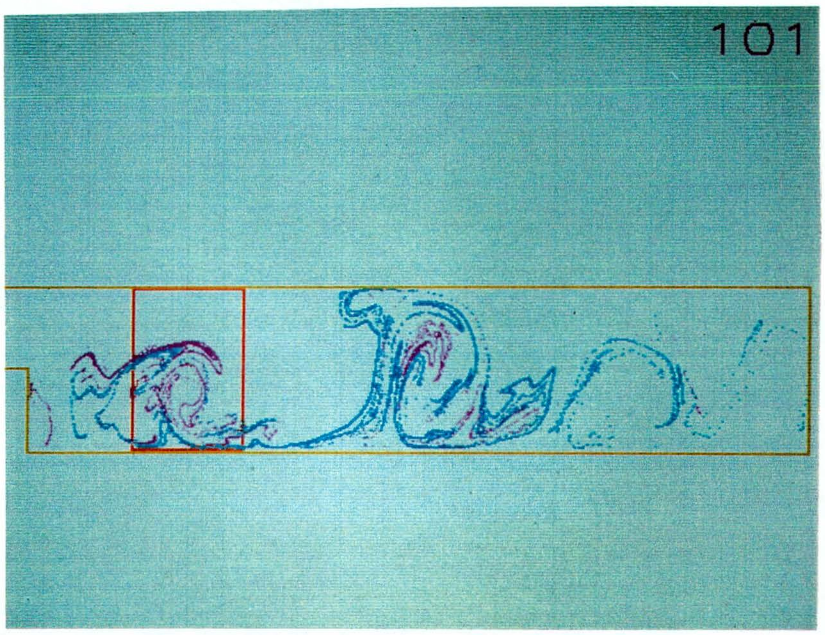


Fig. 1.10. Ten thousand particles injected at two injection points. 101st frame. Outlined region shown in Fig. 1.11.

BBC 884-4311

In addition to problems caused by the low spatial bandwidth of some colors, the use of NTSC encoding necessitates a careful choice of particle and background color combinations. Many choices interact badly, i.e. the color of a group of particles will "bleed" into the background. A particularly bad choice in the 3D case was red for the torus and blue and green for the particles [Appendix C, Fig. C1.4]. The final choice of colors for the 2D and 3D movies was made empirically; there may be other, better, combinations of colors.

Another display problem encountered in the case of the backward facing step involves the aspect ratio of the region modelled, i.e. 10:1, length to height. The resolution of the display is not enough at a tenth of the screen to see clearly what is going on. Even if it were, it would be hard to look at. To solve this problem, the size of the tube is doubled in the y direction, which distorts the flow but at the same time makes it easier to see features of the flow.

To see greater detail in the 2D case, portions of the flow are zoomed in on (enlarged). The aspect ratio of the tube is not as important in this case since the enlarged region can be chosen to have the same aspect ratio as the display device without scaling the flow [Fig. 1.11] [Appendix C, Fig. C1.5].

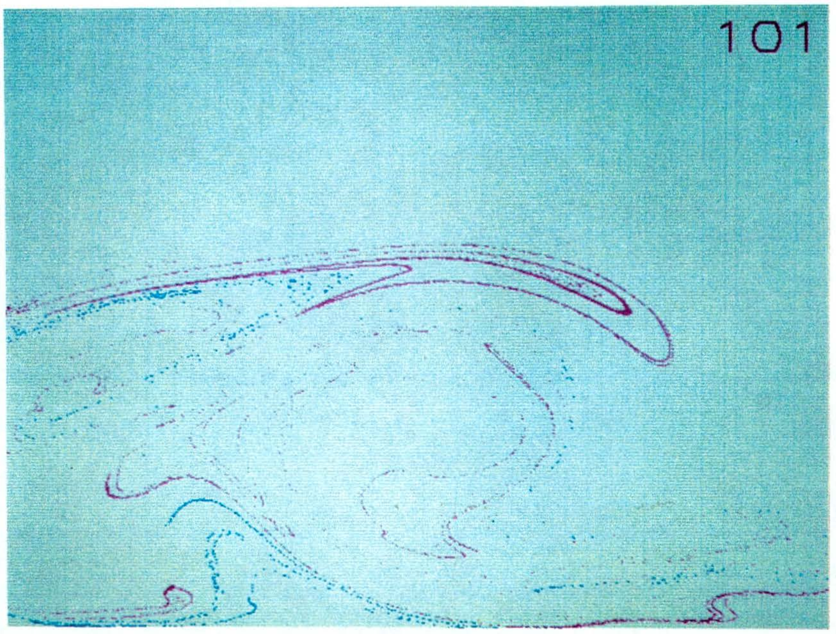


Fig. 1.11. Zoom on outlined region in Fig. 1.10.

BBC 884-4309

If zooming is used, particle parameters such as number injected are modified. Since zooming increases the spatial resolution in the region enlarged, more particles are injected. Five hundred particles are usually injected when using an unenlarged view, but this number results in a very sparse display when zooming is used. Ten thousand particles are injected at two points in Fig. 1.11.

If a fixed area is enlarged, as it is in Appendix C, Fig. C1.4 and Fig. C1.5, it is not always possible to have an interesting level of activity in the region of interest, especially if injection does not occur frequently enough. Particles exit the region of interest more rapidly than in the unenlarged case. The area chosen to enlarge should contain some quasi-steady state phenomenon, such as the stagnation region near the step. As can be seen by Appendix C, Fig. C1.6, a vortex does not stay in a fixed area. The solution to this is to track the vortex as is done in Sethian and Ghoniem [41]

[Appendix C, Fig. C1.1].

1.2.4. Depth Cueing

In advecting particles through 3D space, portraying the depth of a particle becomes a problem. An isolated point, unlike larger objects, does not have 3D cues such as shadows cast upon it, another object partially in front of it, or apparent dimunition in size with distance. Three depth cues have been tried for particles: global rotational motion of the entire flow field, intensity, and size. Of the three, rotation is the most effective and the easiest to implement. In general, rotation is very useful in portraying the 3-dimensionality of many types of objects [4, 15, 47].

Care must be taken with using rotation in spite of its effectiveness. In general, in animating 3D scientific results any additional visual features added to enhance the understanding of these results runs the risk of making the display too busy or complex [15]. When the torus and its enclosed particles are rotated too rapidly, it is difficult to tell whether the particles' motion is due to their being advected through the flow field, or due to the rotation. The rotational movement can be slowed considerably and the 3D effect is still retained. It is also important that the rotation be smooth; jerky movement resulting from quick changes in angle is distracting. A cosine function with the frame number as an argument is used to control the amount of rotation. However, if the torus is rotated through too small an angle too slowly, the particles "flatten out", even though the torus still appears three-dimensional [Appendix C, Fig. C1.7].

Another problem with global rotation is that as the torus is rotated away from a full frontal view, its walls obscure part of the flowing particles [Appendix C, Fig.

C1.8]. A solution to this problem is to make the torus walls transparent, though the effect of intensity cueing of the particles would likely be diminished. The best solution to this problem is to interactively rotate the object and advect particles in real time to be able to quickly explore all possible views. However, workstations capable of doing this with more than a small number of particles advected inside the chambered torus are much more expensive than the video workstation [28,43].

The next most effective depth cue is intensity. In this technique the color of a particle is made darker as its distance from the front of the viewing area increases. It is effective because the eye is accustomed to nearer objects being brighter than far-away ones [13].

The HSV (hue, saturation, value) color model [42] was chosen to arrive at colors differing only in their degree of lightness or darkness. A color is given in terms of a hue, saturation, and value component. Hue distinguishes between colors such as red and green. The saturation component gives the degree of whiteness added to a color and ranges from 0.0 (no white added) to 1.0 (fully white). The value (or intensity) component gives a uniform scaling of the underlying color components, and ranges from 0.0 (fully black) to 1.0 (no attentuation) [13]. Thus the intensity component of a particle's color is diminished to make it appear darker as it moves farther away.

The limited intensity resolution of the 15-bit frame buffer-based display (only 5 bits each for the red, green, and blue components) makes intensity cueing more difficult. The problem is aggravated because particles are colored according to their injection location. Using the HSV color scheme, it is difficult to tell two colors with lower intensities apart [9], especially when using the NTSC video encoding. As a result the lower intensities are not used in implementing this type of cueing. To add to the color range, the color saturation of a particle is decreased (white is added to the color) when it moves towards the front of the torus, making it appear brighter.

Since there are a limited number of colors available, ideally the full range of colors, from brightest to darkest, should be used. To accomplish this, the maximum and minimum z values that a particle takes must be found, at which values the brightest and darkest colors respectively are assigned. There is no way to predict a priori this range of values; the advection process must be carried out to find the path of a particle. One way to arrive at the z range is to use the maximum and minimum z values of the region of interest. However, if the region of interest has a much greater z range than that of the advected particles, only a few colors are assigned.

In the case of the torus, the particles can not move out of the interior so the front and back boundary of the torus, with no rotation applied, are used as the front and back z values. Unfortunately this approach has difficulties. The z range of the torus boundary changes as it is rotated. If rotation moves a particle to a location behind the assigned back z value, the particle cannot be made any darker, distorting the intensity cueing.

A more serious problem occurs because of the shape of the torus. As the torus is rotated all the particles in the forward "arm" of the torus become bright and all the particles on the deeper side become dark. This effect would be informative if the particles were in the interior of a sphere, but the shape of the torus makes this cueing redundant [Fig. 1.12] The solution to both of the above difficulties is to lessen a particle's intensity as it moves deeper into the torus, regardless of its z value resulting from rotation [Fig. 1.13].

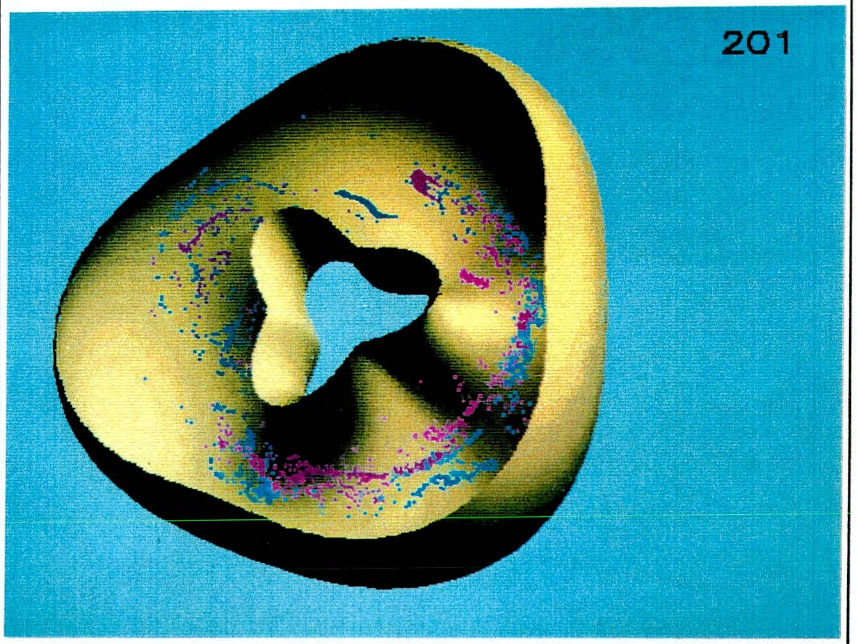


Fig. 1.12. Intensity cueing based on z coordinate. Rotation of torus causes particles at the right to become bright and all those at the left to become dark.

BBC 884-4325

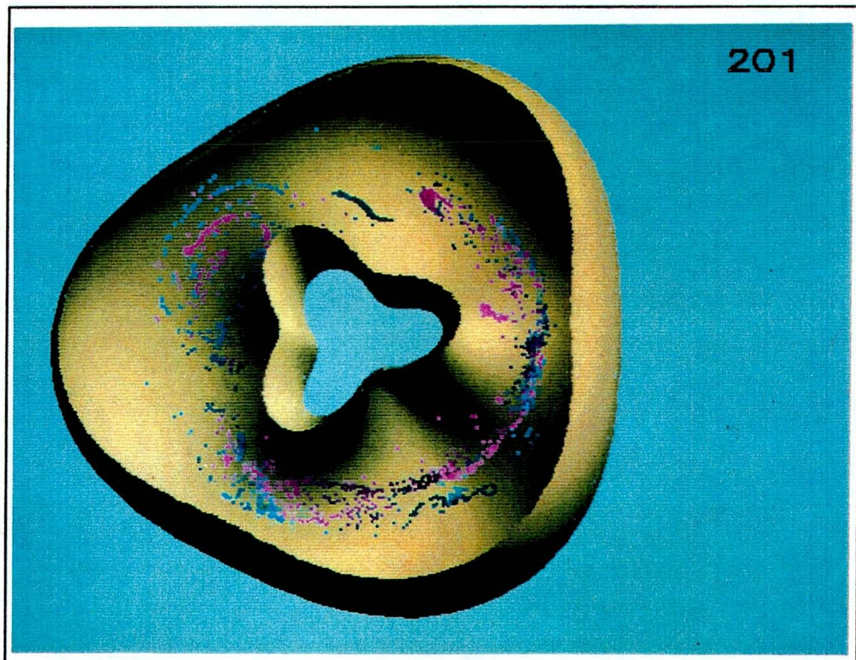


Fig. 1.13. Same as Fig. 1.12 with intensity cueing based on relationship of particle to front of torus.

BBC 884-4323

To accomplish this "depth" cueing, 3 points on the plane containing the front of the unrotated torus are used to find the equation of that plane. The range of z values taken by the unrotated object are also found. As the torus is rotated, the points determining the plane are rotated. The equation:

$$\text{distance} = -\frac{(ax + by + cz + d)}{\sqrt{a^2 + b^2 + c^2}} \tag{1.6}$$

finds the distance of a point from the plane. a, b, c, and d are the components of the plane equation

$$ax + by + cz + d = 0 \tag{1.7}$$

while x, y, and z give the position of the particle [2]. The proportion of the distance found to the total z range determines the intensity of the particle, and when using size cueing, its width also.

Making a particle bigger if it moves toward the front of the torus is the least informative of the three types of depth cueing. The range of tracer particle widths used varied from two pixels to five pixels on a side. With all the activity going on it is hard to tell whether particle(s) are getting bigger, or whether a clump of particles is coming together [Appendix C, Fig. C1.8]. As more particles are advected, this problem becomes worse. Fine detail also becomes harder to see with particles five pixels on a side, since the resolution is effectively reduced by five in that region. Because of these problems, size cueing is not used in the final version of the torus movie [Appendix C, Fig. C1.9].

Chapter 2

Distributed Computing Architecture for Graphics

2.0. Background

Rapidly displaying the graphical representations of a sequence of flow fields requires either the computational power for real-time display or the ability to record the results a frame at a time and then play back the frames in real time. In the past, film recorders were used to record movies of the results of computer modelling on 16 mm or 35 mm film. This involved the expenditure of much time, money, and equipment. The recording process is less time-consuming if videotape instead of film is used. However, the equipment necessary to record a single computer-generated image at a time on videotape is usually expensive. The philosophy of the LBL Video Animation Project has been that scientific movies resulting from computer simulation would be made more frequently if scientists had access to a low-cost video movie making system. Movies produced with this system, while not of broadcast quality, would enable insight into the results of computer modelling. The system put together to reach this goal is described in Johnston, et al. [20].

The goal of this thesis is to display the information in flow fields in graphical form. The achievement of this goal required the use of the "video workstation" described above. The work described in Chapter 1 could not have been done without the color, fairly high-resolution video display, and video recording provided by the workstation.

2.1. Motivation for a Distributed Software Architecture

The main hardware components of the workstation are an IBM PC compatible microcomputer, a 16-bit color frame buffer, a video animation controller, and an Ethernet controller. The Ethernet controller has associated software allowing one to communicate over the network using the TCP/IP [39] protocol. The frame buffer is supplied with software implementing graphics primitives (i.e. for points, lines, etc.). The video animation controller is supplied with software that permits the use of simple commands like "edit in a certain frame at location xxx and record this frame until location yyy". A block diagram of this system is provided in Appendix A.

The first movie made using this system, depicting 2D flow over a backward-facing step [Appendix C, Fig. C1.2], was produced in a convoluted manner. The flow field was generated on a Cray XMP-48 [41]. It was moved by tape to a VAX-VMS system, and then to a VAX-Unix system. Particles were advected with software written and run on the VAX-780 Unix system, since advection was too time-consuming to be done on the PC compatible. The positions and colors of the particles, along with the position of the boundary, were recorded in a graphics metafile. This metafile was then sent to the PC using FTP [8]. The amount of data being sent over the network was so large (around 40 megabytes) that the movie was made in sections. After transfer, a program [20] was started on the PC to read the metafile, display the colored particles, and control the recording process.

This three-step process requires the user to know something about two operating systems, and the file transfer program that operates between them. Another drawback is that human intervention is required after each step is completed in order to start the next step, making it difficult to leave the process running unattended for long periods. 3D rendering has the same problem in that it is also too time consuming to do on the PC.

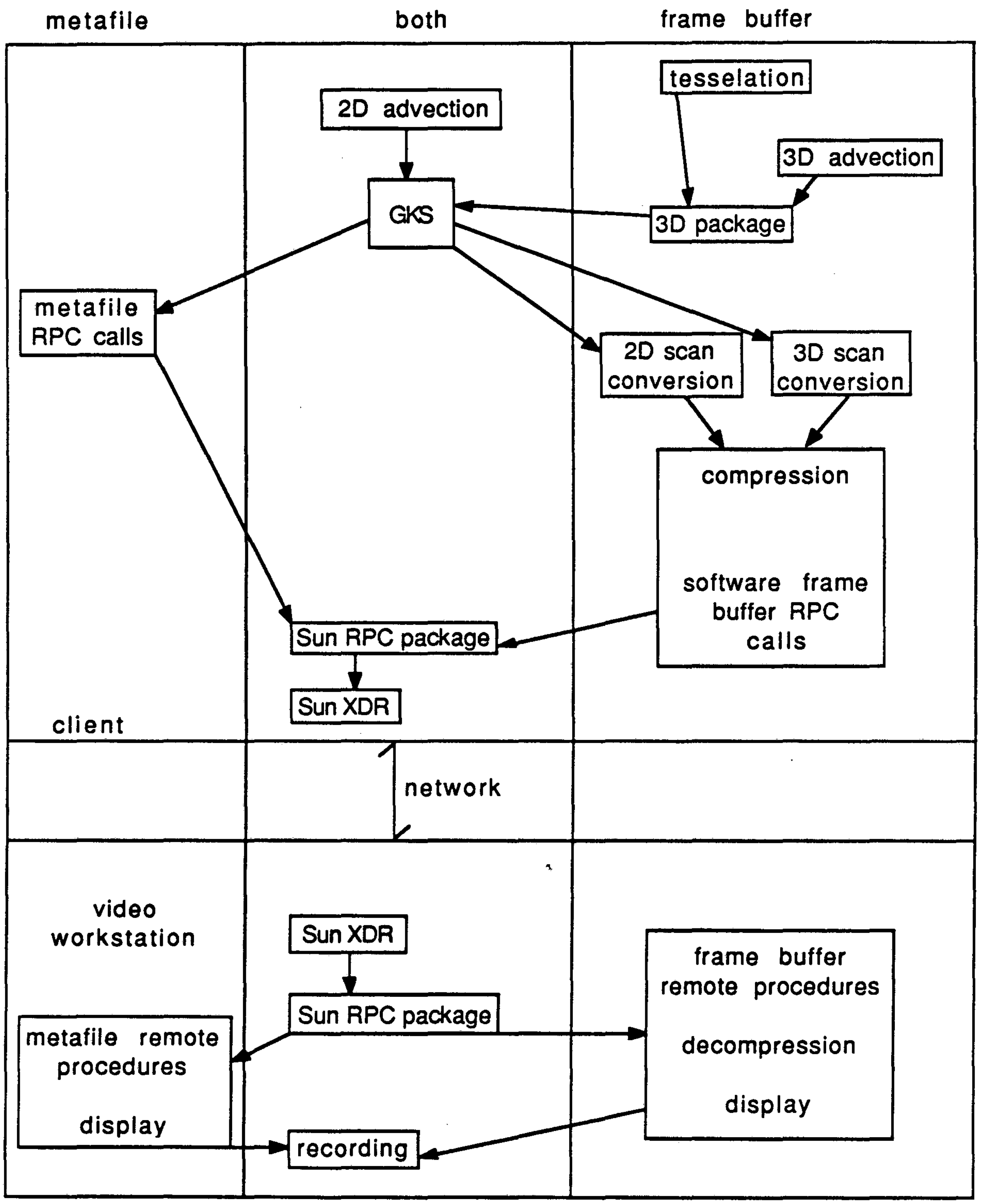
Using a remote procedure call library to control the use of the frame buffer and the video recorder from a front-end system overcomes these difficulties, and has some additional advantages as well. A remote procedure call (hereafter referred to as RPC) is similar to a conventional procedure call, but is made between processes which are potentially on separate machines. An RPC made by one machine (the client) causes the invocation of a procedure on another (the server) through the mediation of the RPC package. The RPC package communicates the arguments across the network, handles data format conversion, and finds the desired procedure on the server. The implementation described here uses the Sun RPC library [35]. This RPC implementation uses the Berkeley Unix socket interface to TCP/IP to provide the underlying network transport [35, 39]. Both are widely available -- thus many systems can use the video workstation for display and recording.

Using RPC's, the three-step process described above is avoided. Following advection, RPC's are used to display graphics primitives and to control the recording process on the video workstation directly from the front-end machine. A user does not need to know anything about the PC with RPC's -- the PC is relegated to the role of a non-interactive animation server. Two approaches are taken in transferring the graphics data. In one, used only in the 2D case, point and line RPC-based graphics primitives are used to invoke the corresponding software routines on the workstation side. In the second approach, graphics primitives, including polygons in the 3D case, are rendered (scan converted) into a software frame buffer located in main memory on the front end machine. This software frame buffer is transferred to the workstation using RPC's and written into the PC frame buffer several scan lines at a time.

2.2. Flow of Graphics Information

The following diagram [Fig. 2.1] shows the flow of graphics information in the two approaches. Each box in the diagram represents a relatively independent module that the information passes through. These modules are described further in Appendix B. In the 2D metafile approach, particles are advected and 2D GKS-like routines invoked. All the steps below are internal and invisible to the user. GKS (the Graphical Kernel Standard) provides a paradigm for a device (and in this case, approach) independent manner of invoking graphics primitives. The user only has to set a flag indicating the metafile approach, and the GKS level automatically calls the low-level metafile routines that in turn invoke RPC's to display the particles and the 2D boundary of the modelled region. The RPC package takes control from here and converts the data into XDR (eXternal Data Representation) form, a standard, machine-independent, representation of data [12]. The information is then sent over the Ethernet using one of two IP protocols, UDP (User Datagram Protocol) or TCP (Transmission Control Protocol), which will be described in the next section. The data received by the PC are converted from XDR format into PC format. The RPC package on the PC finds the desired remote procedure, which then invokes local software providing the graphics primitive to display the data. A separate RPC is made to record the display on frame(s) of videotape.

Fig. 2.1. Flow of graphics information



The other half of the diagram shows the flow of information in the software frame buffer approach. Again, the GKS module is the lowest module visible to the user. In the 3D case, particles are advected and the torus (boundary) data is converted into polygonal form by "tessellation" (described in Section 3.1). The resulting positions of the particles, and the vertices of the polygons are sent through the 3D viewing pipeline, which invokes the appropriate GKS-level calls to display the data. The GKS module invokes 3D routines to scan convert the data into the software frame buffer. The user makes a separate GKS call which invokes a RPC to display the information contained in the software frame buffer.

The software frame buffer requires storage of about 400 kilobytes of data, and the bandwidth observed in the Ethernet connecting the front-end system and the workstation is from 10 to 50 kilobytes/second, depending on the protocol used (see section 2.3). Thus the frame buffer is usually compressed before transmission. (The two types of compression used, and the corresponding decompression on the workstation, are described in Section 2.4.) On the front end system, the RPC package takes the data, converts it into PC internal format, and calls a special XDR routine, xdr_opaque, to take the data and place it in a buffer with no conversion for transferral. Since there are various front-end systems, conditional compilation is used to select the correct means of translating the compressed data into PC format for the mode of compression transferring the data in short integer format. The data is not put in the standard XDR form because conversion from XDR to PC form would take too long on the PC. From here the steps are similar to that of the metafile approach with the addition of the decompression step on the workstation side [Fig. 2.1].

2.3. The RPC Level

Before RPC's could be used, a large part of the Sun RPC library had to be ported to the PC. This work is described in detail in [36]. The two factors on the PC that made this software port possible were the commercially supplied socket library for the Ethernet controller [30], and a run-time library providing a similar environment to Unix [27].

Socket calls are Unix low-level primitives for communication between processes running on the same or different machines. Sun RPC has two flavors -- one in which socket calls are made with the UDP protocol, and the other with TCP. TCP is a stream, or connection oriented protocol, and data transmitted by a single RPC using TCP can be of any length up to $2^{31} - 1$ bytes (in Sun RPC) [12]. UDP is a datagram protocol. The data transferred by a single RPC is limited by the maximum packet length. This length is system dependent, but modern UNIX systems typically allow 8K [35]. UDP transmission is not error-free, while TCP is guaranteed to be reliable, at the cost of error-checking overhead [39].

RPC's were first implemented on the PC using the UDP protocol. Since UDP is not reliable, the Sun RPC package provides some mechanisms for reliability. In particular, the client re-transmits data if an acknowledgement to a RPC is not received within a set time limit. A problem arises in the proper setting for this time limit. If the time limit is too long, excessive delays in displaying the data may result when a packet of data is lost because the client will wait to re-transmit until the time-out period expires. If the time limit is too short, the slowness of the PC server in performing an RPC may cause duplicate calls, because the client could re-transmit several times before the server has a chance to acknowledge. Based on experience over many runs, the retransmission rate eventually chosen was 4 seconds for a local-area network.

At first this setting caused problems when transferring the software frame buffer because one RPC was used to transfer one scan line, resulting in 400 RPC's per image. One or two scanlines per image were sometimes lost, causing a delay of 4 or 8 seconds. This problem is alleviated when compression is used, allowing dozens of scan lines to be sent in a packet via an RPC. The percentage of lost packets might be the same, but with compression there are many fewer packets being sent.

Recording a frame on videotape takes about 15 seconds. This length of time does not cause RPC packet retransmission because the recording RPC is made asynchronously. That is, the PC sends back an acknowledgement to the RPC which initiates the recording before it sends the record command to the animation controller. This allows the front end to be advecting the particles or rendering the next frame while the recording process is going on.

Because the recording RPC is asynchronous, the next RPC made after it is often re-transmitted because the PC is still waiting for the videotape recorder (VTR) to finish and is not able to respond. If the amount of data to be sent is potentially large, a null RPC (sending no data) is made first to verify that the PC is ready before making the next RPC call.

The problems associated with using UDP are eliminated when using the RPC library with the TCP protocol. No packets are lost. The price is a slower rate of transmission over the network. One study showed a maximum bandwidth between

Sun 3/50's of 310 kilobytes per second using RPC's with the UDP protocol but only 140 kilobytes per second using TCP [38]. The bandwidth between a front-end and the PC is considerably less using either protocol. A rough estimate of the data transfer rate between a Sun 3/280 front end and the workstation using RPC's with the UDP protocol is 50 kilobytes per second. With TCP this rate goes down to roughly 10 to 20 kilobytes per second. Over a local-area network, reliability is generally not a problem and UDP is used as the transport protocol. TCP is used over a wide-area network, since reliability is potentially an issue in that case.

Another advantage of TCP became evident when the client side of the package was ported to a Cray X/MP-14 Unicos system. When the Cray is heavily used there is a sizeable delay after each RPC. Even with compression, the amount of data associated with the software frame buffer exceeds the 8K limit imposed by UDP. Since the compressed image of the torus and particles is on the order of 30 to 40K bytes, several RPC's have to be made to transfer the data, with the accompanying delays. With TCP the entire compressed frame buffer is sent with one RPC. There is still a small delay evident after a certain amount of data (estimated at 10K) is sent, but the total transferral time is much smaller.

2.4. Compression Techniques

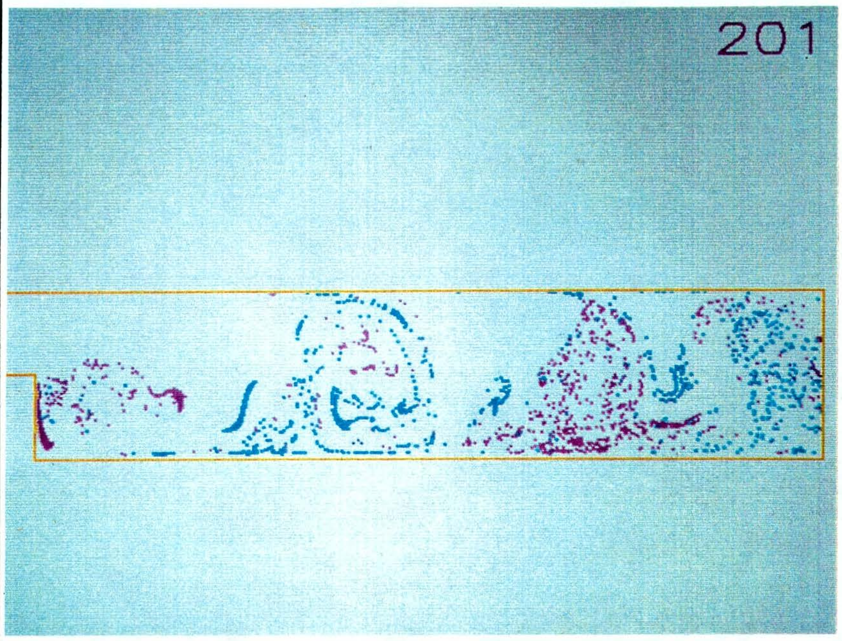
Two suites of programs [44] for compressing and decompressing raster images were integrated into the portion of the graphics package using the software frame buffer approach. One suite uses block truncation coding (BTC) and frame to frame differencing, while the other uses a preliminary BTC step, encoding using a color map, and frame to frame differencing. Both are variations of the encoding described in

Campbell et al. [6].

BTC encoding divides the frame buffer (raster image of 15 bits per pixel) into 4x4 blocks. Two "best" colors are chosen to represent the block. These two colors, along with a bitmap which has 1's for pixels closer to one color and 0's for pixels closer to the other, are the compressed version of the original 4x4 block of pixels. Using the color map approach, 256 colors are selected which are most representative [16] of the colors found in the uncompressed image ($2^{15} - 1$ colors). The two best colors found by BTC encoding are represented by pointers into a lookup table containing 256 colors [6]. Frame to frame differencing finds those blocks of pixels which do not change from one frame to the next. The two colors and the bit map only need to be stored for blocks that change [44].

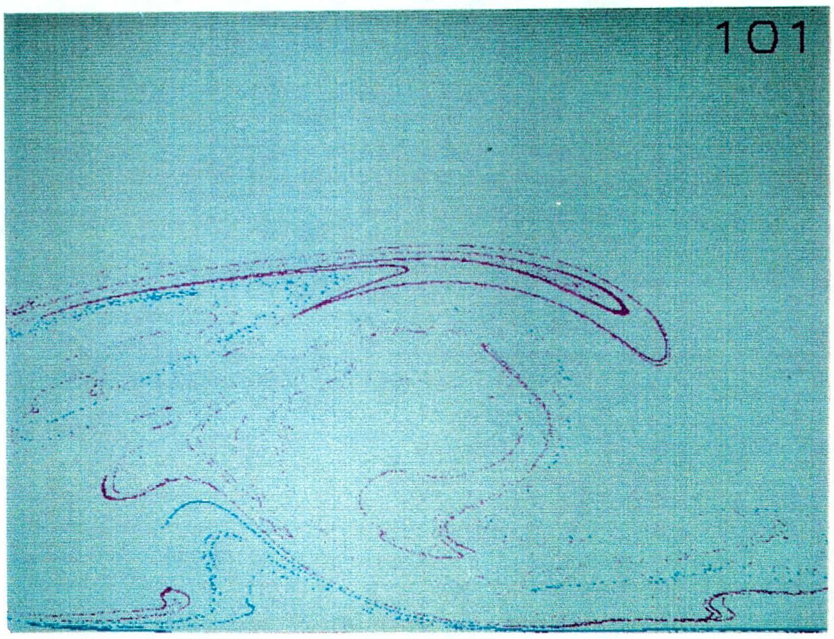
Different types of images result in differing amounts of compression, as illustrated in Table 2.1 for the four images [Fig. 2.2a through Fig. 2.2d]. The minimum amount of compression using BTC is 5.33:1. The maximum size of a 4x4 block using only BTC encoding is 6 bytes, and there are 12,800 blocks, or 76,800 bytes. (The uncompressed software frame buffer is 409,600 bytes.) The minimum amount of compression using BTC and the color map is 8:1, since the maximum size of a 4x4 block is 4 bytes.

Fig. 2.2. Four test images.



(a) Flow over backward facing step.
500 particles injected.
201st frame.

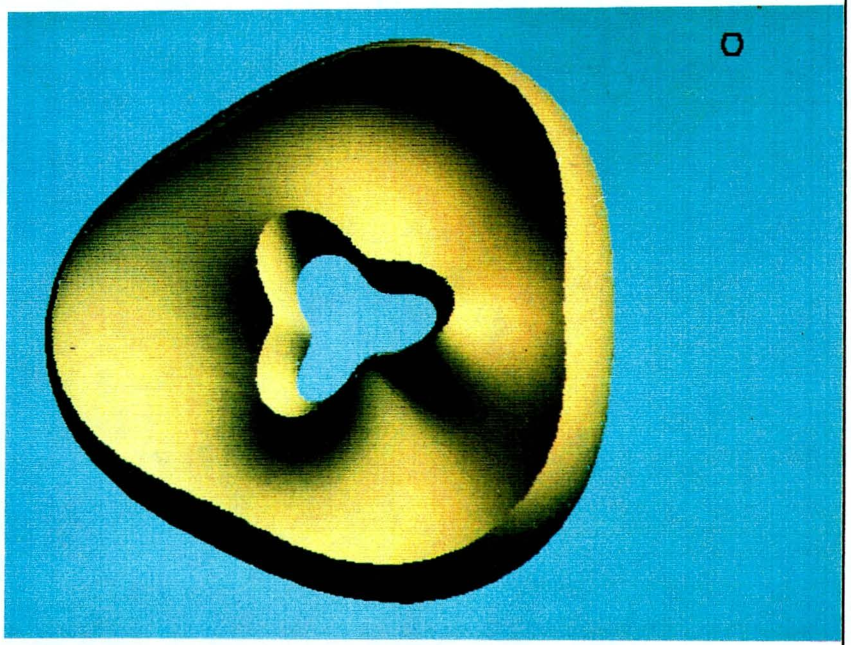
BBC 884-4315



(b) Zoom on region outlined in Fig. 1.10.
10000 particles injected. 101st frame. Image same as Fig. 1.11.

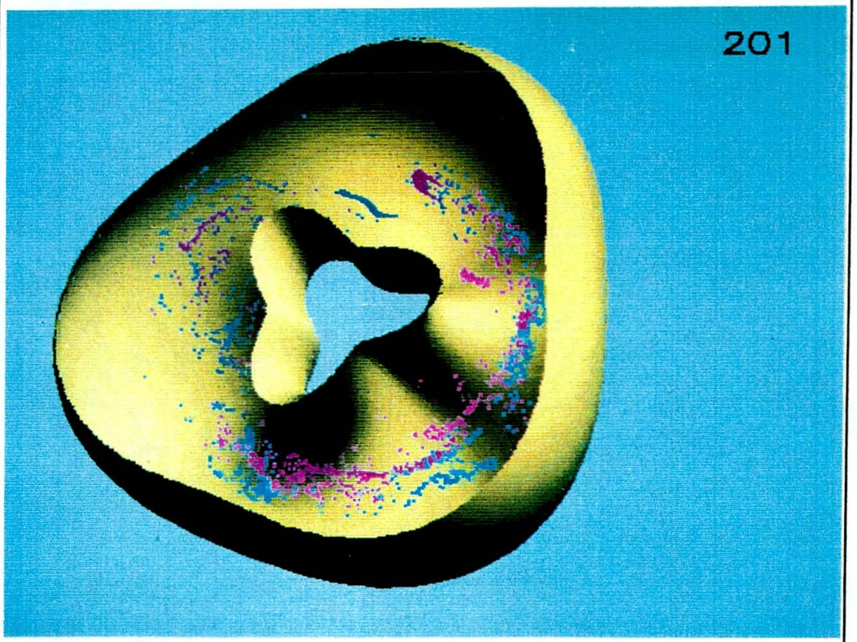
BBC 884-4307

Fig. 2.2. Four test images (continued).



(c) Smooth shaded clipped torus.

BBC 884-4331



(d) Same as (c) with the addition of advected particles.

BBC 884-4327

Table 2.1.--Compression ratios for Figs. 2.2.a through 2.2.d.

compression	(a)	(b)	(c)	(d)
BTC only	7.6:1	7.5:1	6.7:1	6.7:1
BTC and color map only	10.3:1	10.2:1	9.4:1	9.4:1
modification from one frame to the next	advection	advection	rotation	advection, rotation
BTC and frame-frame	43.5:1	27.2:1	12.9:1	12.8:1
BTC, color map, and frame-frame	59.3:1	37.8:1	18.8:1	18.6:1

In each case the colormap does a better job of compression. The frame-to-frame differencing numbers are for a slightly modified version of each image, as indicated in Table 2.1. As can be seen for the numbers for the 2D image, frame-to-frame differencing is very effective for images with a large amount of constant background. In general, scientific graphics synthetic images do not fill up the whole viewing surface unless they are distorted, and the unused portion of the screen compresses very well. However, if a portion of the image is enlarged by zooming in on it as in Fig. 2.2.b, the enlarged portion may indeed fill up the entire screen, in which case the compression ratios are not as good. Shaded 3D images do not compress as well as 2D images, as indicated by the numbers for Fig. 2.2.c and Fig. 2.2.d. The modified (rotated) version of Fig. 2.2.c compresses only slightly better than the modified version of Fig. 2.2.d, in which many particles are changing their positions due to advection in addition to changes due to rotation. However, there is a large amount of difference between frames in both cases, even though the changes in color between Fig. 2.2.c and its modified version are more subtle.

Three factors which must be balanced when choosing the type of compression to use, or whether to use compression at all, are the speed of the decompression at the video recording workstation, the bandwidth of the network, and the speed of the front end system which does the compression. Speedy decompression is essential since the video recording workstation is a PC. Decompression using either BTC by itself or BTC with the addition of a color map requires only a table lookup for each pixel in a block that changes.

A program displaying 7 frames of a rotating torus, one frame of which is Fig. 2.3, was timed three separate times, once using no compression, once using BTC compression only, and once using BTC compression and a color map. The three timings differed by only one second in elapsed cpu time (102, 103, and 103 cpu seconds, respectively). The cpu time spent in compression is justifiable; just as much cpu time is spent in the RPC and transport levels when compression is not used. The amount of real time elapsed in each instance differed more because the time spent in transmitting the data is not added into the cpu time. The color map option is more efficient in compression than BTC by itself, which was reflected in the former option having 30 seconds less elapsed real time than the latter. Using no compression resulted in 45 seconds greater elapsed real time than when using the color map option. Much more data is handled by the network protocol layers on the front-end and on the slower PC side when transporting the entire software frame buffer. These three timings when repeated several times gave much the same picture in real time elapsed.

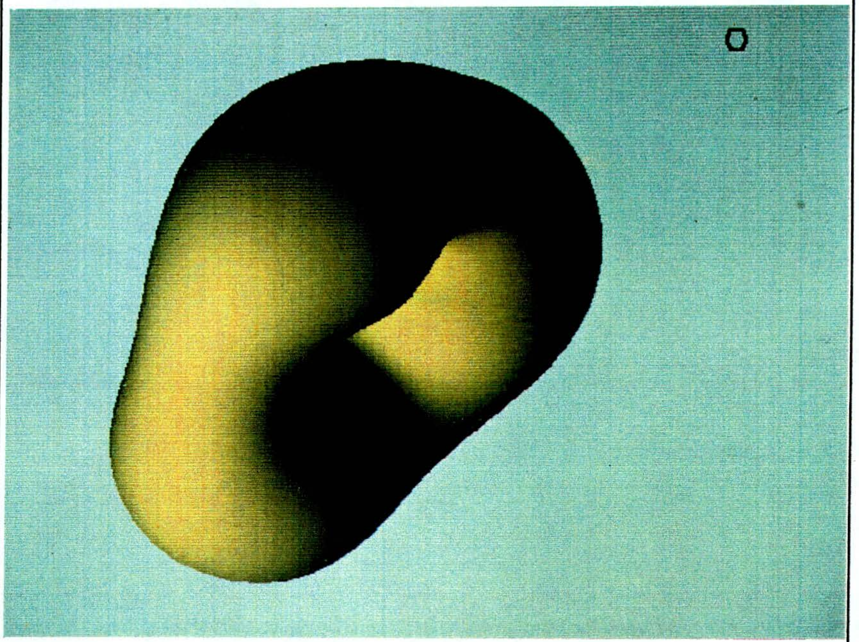


Fig. 2.3. Smooth shaded torus.

BBC 884-4335

The main variable in choosing between no compression, BTC encoding, or color map encoding is the amount of data sent across the network. The color map approach results in the least amount of data sent, which is especially valuable when data is sent over a slower network. When an image is sent over a wide-area network, which has an effective bandwidth of 1 to 2K bytes per second, saving even a few kilobytes is worthwhile. Thus an image compressed using the color map approach is compressed still further using Lempel-Ziv compression when the image is sent over a wide-area network.

The Lempel-Ziv algorithm is an adaptive Huffman compression technique [23]. The distributed graphics package was modified to use an implementation of this algorithm (a variation of the UNIX *compress* function) [18] as an additional compression step after BTC or color map encoding. Decompression takes place on a 68020 coprocessor on the PC, since decoding is as expensive as encoding, unlike the BTC and color map approaches.

Table 2.2 lists compression ratios for the four images in Fig. 2.2 using a color map, frame-to-frame differencing, and Lempel-Ziv encoding. It should be noted that a digitized natural scene has too much noise to be able to achieve these compression ratios; however, synthetic images have a large amount of redundancy in them. As can be seen, even the color-map, frame-to-frame differenced compressed image has redundancy in it that Lempel-Ziv compression is able to exploit.

Table 2.2.--Compression ratios with the addition of Lempel-Ziv.

compression	(a)	(b)	(c)	(d)
BTC, color map, and frame-frame	59.3:1	37.8:1	18.8:1	18.6:1
Addition of Lempel-Ziv	124.1:1	80.8:1	23.9:1	22.2:1

Another reason why compression is preferred to no compression is that compression permits storage of the entire frame buffer on the PC so that maximum execution overlap with the front end can occur. After the compressed image is stored, the PC sends an acknowledgement to the front end, allowing it to continue while the stored image is decompressed and displayed in the hardware frame buffer. When the front end sends over an uncompressed image to the PC, it must do so several scan lines at a time. The PC has only 640K of RAM available in its current configuration and roughly 150K of this is used by the executable version of the server alone. There is not enough space available for the full 409,600 bytes of uncompressed image. In this case the front end must wait until the whole image is displayed before it can continue. The compressed image fits because, as mentioned above, the maximum size of a BTC-compressed frame buffer is 76,800 bytes.

The main drawback of using either the BTC or the colormap approach is that they are not exact algorithms, that is, decompressing a compressed image does not always result in an exact duplicate of the original image. As mentioned before, two "best" colors are chosen to represent the colors of all pixels in a 4x4 block. If there are more than two colors in a block, this results in inaccuracies [44]. For example, Fig. 2.4 from the backward facing step movie with 10,000 particles shows that where the cyan and magenta particles lie close together, some particles are given the wrong color.

These inaccuracies are almost impossible to see when using the NTSC color encoding. Even when compression is not used, colors of intermixed cyan and magenta particles are difficult to distinguish [Appendix C, Fig. C2.1]. To get a clearer picture of the situation in an intermixed region, that region is zoomed in on. In this case cyan and magenta particles are less likely to lie within the same 4x4 block, since the resolution is effectively increased, and the NTSC encoding is less likely to blur the cyan and magenta particles together [Fig. 2.5] [Appendix C, Fig. C2.2]. (As mentioned in section 1.2.3, the photographs are not NTSC encoded. It is necessary to view the videotape to see the effects of NTSC encoding.)

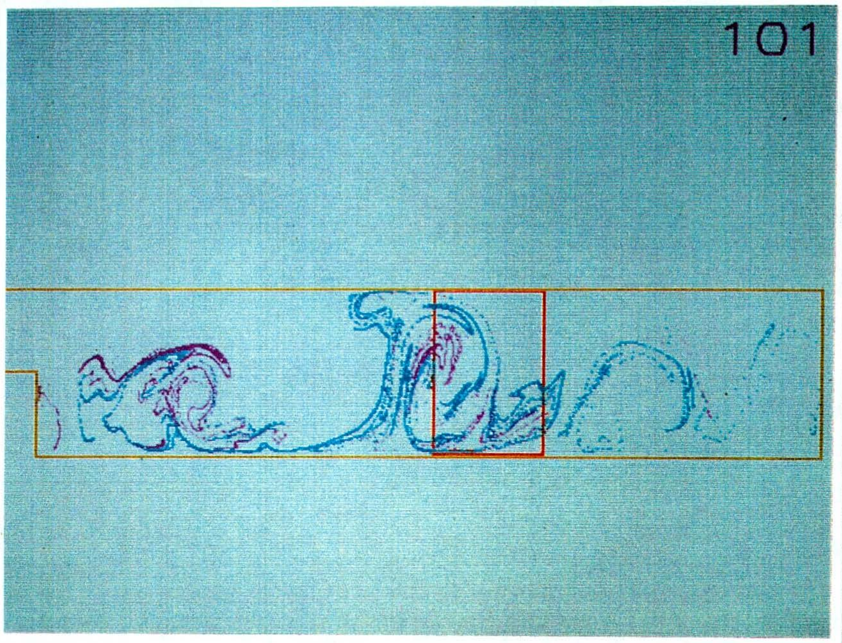


Fig. 2.4. Result of BTC and color map encoding showing inaccuracies.

BBC 884-4313

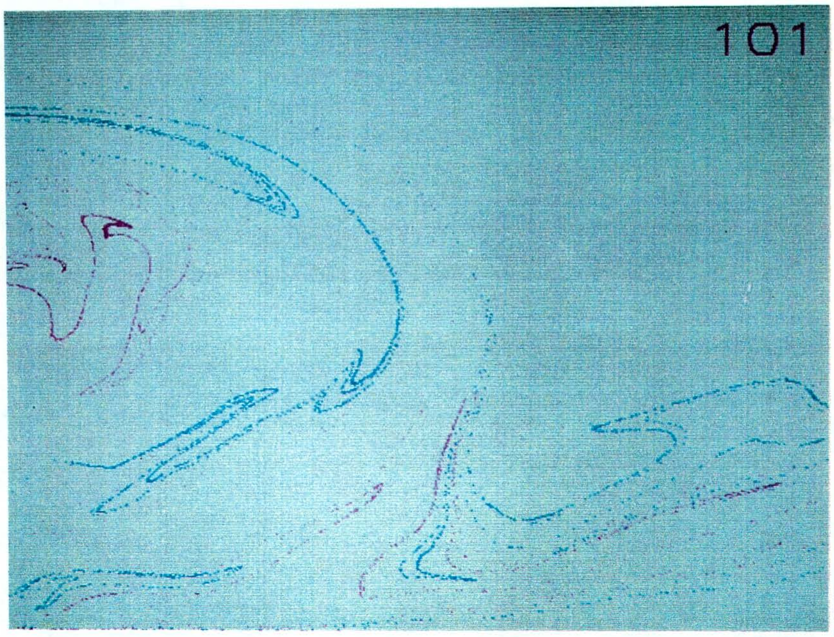


Fig. 2.5. Zoom on outlined region in Fig. 2.4.

BBC 884-4303

A potential problem with the color map approach is that 256 colors are chosen which are most representative of the colors in the entire image, which may have a much greater range of colors. The algorithm chooses the color of a pixel in such a way that it is "close" to the true color. The inaccuracy is unnoticeable with the synthetic images generated in this thesis to visualize fluid modelling. The limited number of colors is much more noticeable in digitized images captured by a camera that have been compressed using a color map [16].

2.5. User-Level Interfaces in the Two Approaches

GKS-style calls are the user-level interface to the routines implementing the metafile and software frame buffer approaches to displaying images on the video workstation. The semantics are different from GKS in that only one way of generating images is accessible at one time, since the metafile and the software frame buffer approach access the same display.

The metafile approach is limited to displaying 2D images for several reasons. The only information sent to the PC with the metafile approach is the position and color of points, lines, and text. They must be scan converted into the hardware frame buffer on the PC side. The process of scan converting 3D polygons is time consuming even on a Sun 3/280; it would be much more so on a PC. The algorithm chosen for hidden-surface removal of 3D objects uses a 0.8 megabyte buffer in addition to the frame buffer. The frame buffer by itself does not fit into PC memory.

A listing of the user-level routines causing invocation of RPC's further illustrates the difference between the two approaches. Following are the calls causing RPC's with the metafile approach:

1. *Dopwk (approach, connection-id)*

   Initializes the hardware frame buffer, the recording device, and the RPC package. *approach* chooses the metafile or software frame buffer approach. *connection-id* has two values or'd together, the *protocol* value and a number identifying which video workstation (#1 or #2) to connect to. Sets up the RPC package to use UDP or TCP based on the *protocol* argument.

2. *Dclwk (approach)*

   Closes the devices and the RPC package.

3. *Dscr (approach,index,hue,sat,val)*

   Sets an entry in the color table resident on the PC via an RPC if the metafile approach is used. The current point, line, or text color (set by other calls) is identified by an index into this table. *approach* identifies whether the metafile or software approach is to be used. *index* is the location of the entry set in the color lookup table. *hue, sat,* and *val* are the hue, saturation, and value components of the color specified.

4. *dtxt (x, y, str)*

   Writes the text *str* at location *x, y* via an RPC. Frequently used to include the frame identifier as text in the graphics frame.

5. *Dpl (numpts, x_array, y_array)*

   Draws a connected line made up of *numpts* number of points given by *x_array* and *y_array*.

6. *Dpt (x, y)*

   Plots a point at location *(x, y)*. The point will not appear on the monitor immediately. All points to be plotted are stored until the *Duwk* call. Thousands of particles are advected after several injections have been made. One RPC per point plotted would incur too much overhead.

7. *Duwk (approach)*

   Sends all points stored as a result of the *point* command to the workstation to be displayed if *approach* indicates the metafile approach is being used. If not in preview mode, records what is stored in the frame buffer on a videotape or videodisk frame(s). When preview mode has been set, the images are displayed and not recorded.

8. *Dclrwk (approach)*

   Clears the hardware display via an RPC in preparation for the next frame if *approach* indicates the metafile approach is being used.

Following is a brief description of user routines causing an RPC call in the software frame buffer approach:

1. *Dopwk (approach, connection-id)*

   Performs the same functions as in the metafile approach.

2. *Dclwk (approach)*

   Closes the devices and the RPC package.

3. *Duwk (approach)*

Copies the information in the software frame buffer into the hardware frame buffer via RPC(s) if *approach* indicates the software frame buffer approach is used. If not in preview mode, records what is stored in the frame buffer on videotape or videodisk.

For reasons described above, 3D primitives are not displayed using the metafile approach. When displaying 2D fluid flow, a choice is available between the metafile and software approaches. The metafile approach is less computationally expensive on the front end. However, the software frame buffer approach is far superior even when only 500 particles are injected. With the metafile approach, many more RPC's are needed to display a frame, and the front-end has to wait for the PC to finish on each one but the recording RPC. If compression is used with the software frame buffer approach, the front-end is free to continue once the data for the entire frame has been received on the PC end. As mentioned before, display in the hardware frame buffer as well as recording are overlapped with computation on the front-end. In addition, it takes only 100 graphics calls on the PC to display the information in the decompressed software frame buffer (at 4 scan lines per call). With the metafile approach, a graphics call is made for each particle, which is wasteful if 20,000 particles are to be displayed.

Chapter 3

Graphics Issues

3.0. Introduction

In many, if not most, cases of modelling fluid flow, an object perturbs or constrains the flow. In these instances correct visualization of this object is an important aspect of fluid flow display, especially in the 3D case. The geometry of the object is used in modelling the flow, and aspects of that object such as surface discontinuities are important for understanding what is going on in a particular region of the fluid.

To display a 3D object and show the 2D projection of 3D particles, it was necessary to implement several graphics algorithms, and modify and add to already available software. The software available was an experimental 3D viewing package for displaying line drawings that provides an easy means of changing the view of objects [48]. Modifications to this package involved adding the ability to send points and polygons through the 3D viewing pipeline and the addition of two types of polygon clipping. These modifications, and the implementation of the graphics algorithms, were guided by the desire to provide an unambiguous and illustrative display of the object and the flow field in a timely manner, rather than the achievement of photographic realism.

The video workstation is designed more for scientists to gain insight into the large amount of data produced by scientific modelling than for the final presentation of the results, that is, it is designed more for discovery than presentation graphics.

For the purposes of discovery it is more important to be able to look rapidly through the data than it is to have a polished representation. The more sophisticated graphics algorithms for displaying surfaces are potentially very cpu time-consuming, and would require a large expenditure of programming time to implement. Given the desire for speed and economy of programming, an attempt was made to find the minimal implementation that would enable insight into the dynamics of the flow.

3.1. Representation of the Object

In 2D representations, all that is required to display the object constraining the flow are line segments. The choice of what part of the object to display vis a vis the modelling data may require some judgment. For example, the flow field for the backward-facing step is modelled on a rectangular grid [Fig. 3.1]. No modelling was done for that area of the tube to the left of the step. To properly understand the vortices forming to the right and below the step it is necessary to include the boundary of this non-modelled area and not just the boundary of the flow field proper.

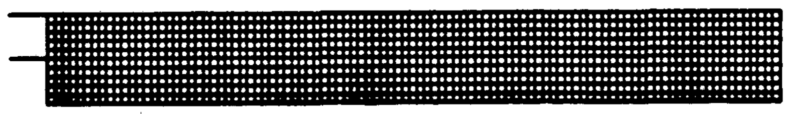


Fig. 3.1. Gridded portion is region modelled.

The representation of a 3D object is much more difficult. When a polygonal representation is available, there are well-known algorithms for rendering the object: performing projection, scan conversion, solving the hidden surface problem, and providing a lighting and shading model. Frequently, however, the object is not given as a surface(s) described as polygons, in which case the algorithms for display are not well developed or don't exist at all. In the general case (not just in flow visualization) the object data available may be 2D cross-sections of an object, such as contour lines representing some value in a scalar field (e.g. a terrain contour map), views of an object taken from many different positions, such as in X-ray tomography [17], a 3D array of scalar values obtained from mathematical modelling [41], or many other representations. For the torus, the data was a collection of strands or fibers, a variation of a parametric representation [26, 32].

The process of generating a polygonal representation from a non-polygonal one is called tessellation. One common method of tessellation used when planar contour slices of an object are available is triangulation. In this method points on one contour are connected to an adjacent contour to form a series of triangles [14] [Fig. 3.2].

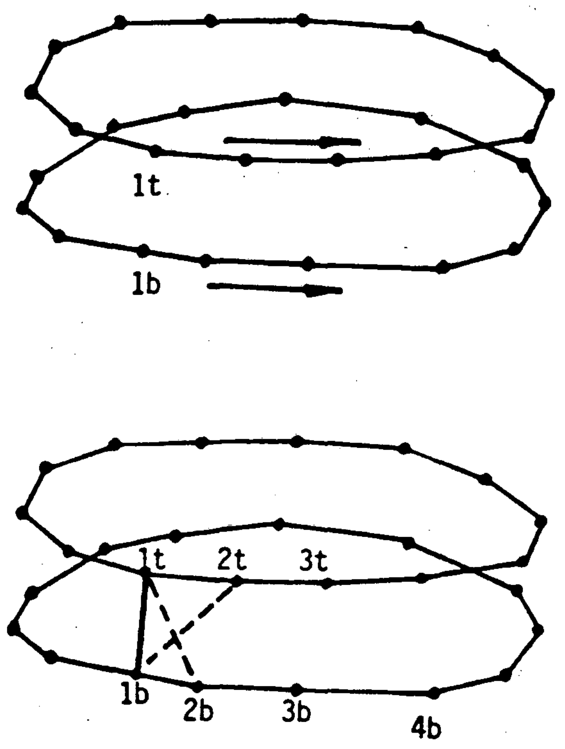


Fig. 3.2. Triangulation between contour pairs. Reprinted from Christiansen and Stephenson [7], 14.2.

The torus is given as a series of criss-crossing inner and outer fibers [Fig. 3.3]. The points making up the fibers are given in such a way that it is possible to triangulate between one fiber and the next and achieve a good" triangulation [25]; that is, where most polygons have roughly equal area, and where there are few very acute angles [7]. Only the outer fibers are used in the triangulation.

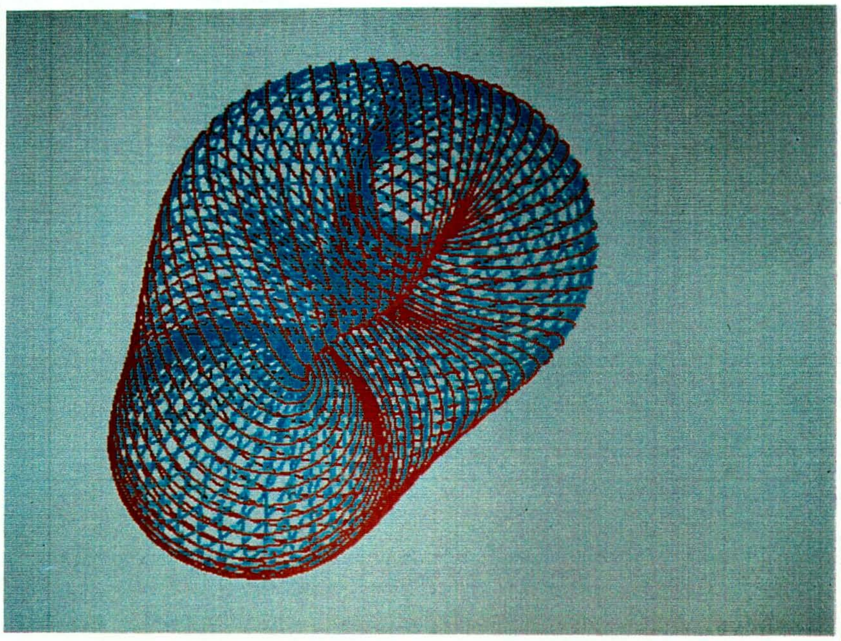


Fig. 3.3. Torus given in form of inner and outer fibers.

BBC 884-4295

An attempt was originally made to find a more general method that would tessellate several different representations of an object. In this method samples are made of parallel slices of the outer strands of the torus to generate a collection of points at each slice. After the points are connected to form contours, the Mosaic package of movie.byu is used for triangulation [7]. This method was abandoned because points from narrow slices of the torus are impossible to connect, and because the Mosaic program has to be guided by hand owing to the complexity of the torus. The Mosaic package especially has difficulty with connecting three contours at one level to two contours at the next, which occurrs where the "bumps" on the torus merge into its main body [Fig. 3.4]. In general, methods of tessellation by connecting contours have problems with saddles [7].

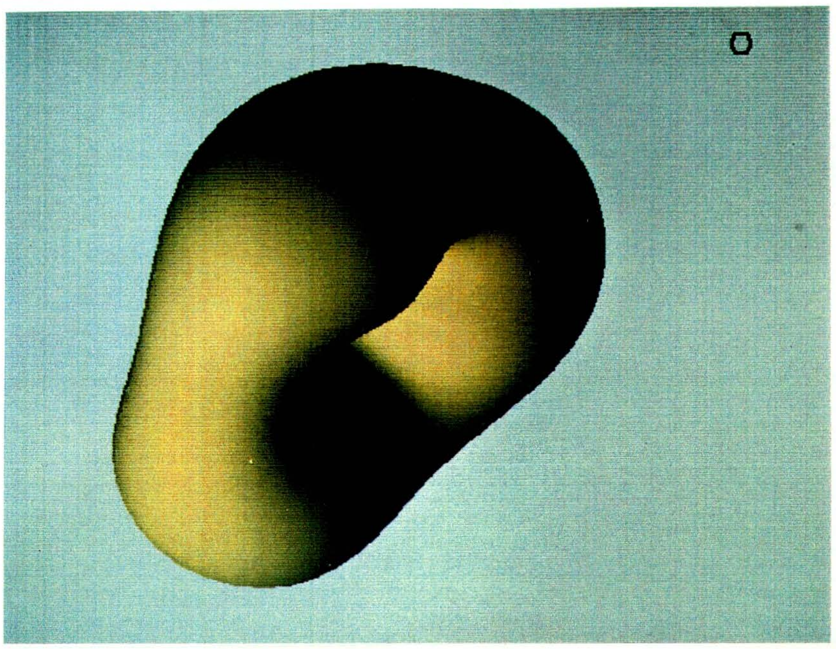


Fig. 3.4. Smooth shaded torus.

BBC 884-4333

A major issue encountered after the torus was tesselated concerned the display of that object in conjunction with the advected particles. The particles move in the interior. If the object is a cylinder or a space shuttle wing [Fig. 3.5], the movement of the particles around it gives a form of depth cueing. On the other hand, particles moving inside the clipped torus are sometimes obscured by it, but always have it as a background [Fig. 3.6]. In this case, the depth cues described in Section 1.2.4 become much more important.

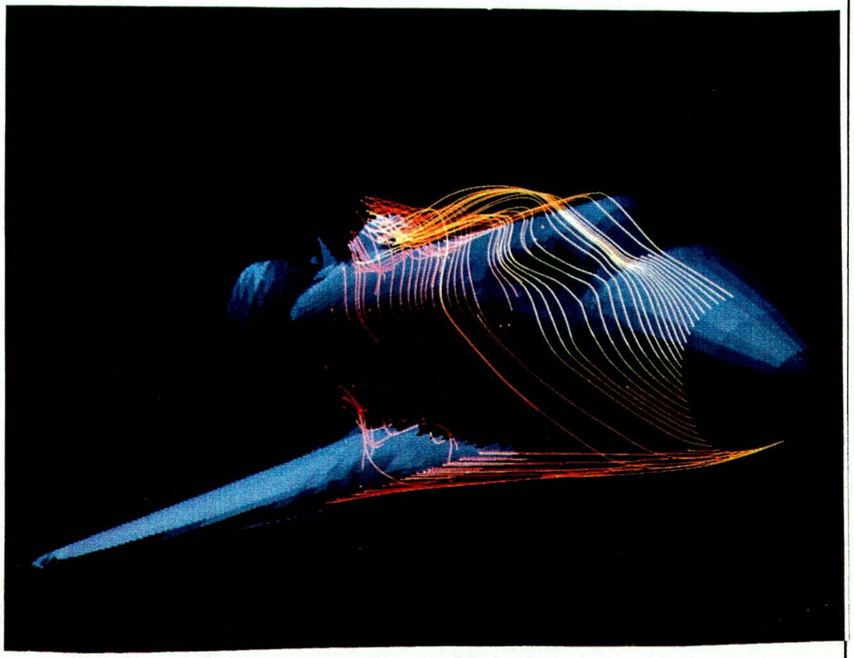


Fig. 3.5. "Flow over the Space Shuttle (Computer simulations by Chaussee, Rizk, and Buning)". Courtesy V. Watson, figure from Watson, et al. [47], 7.

BBC 880-11631

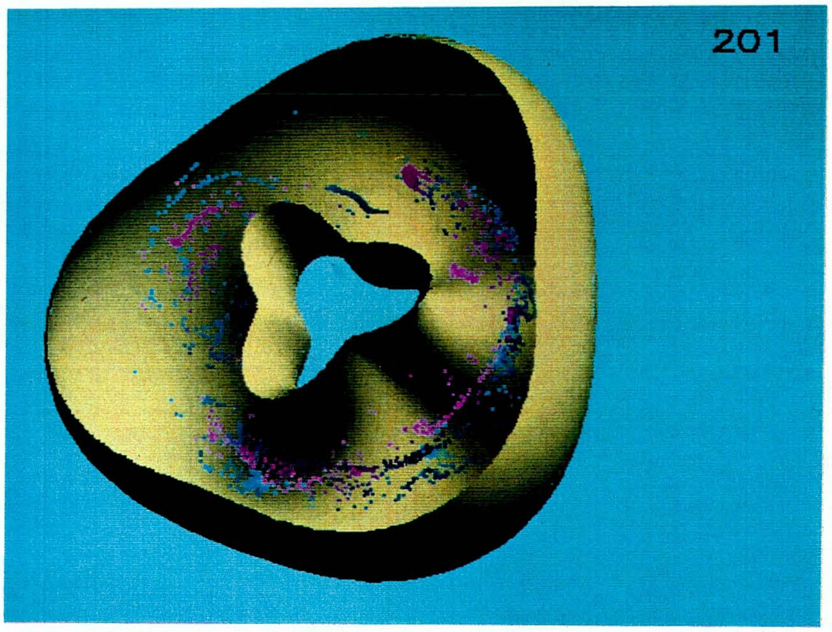


Fig. 3.6. Smooth shaded torus cut away to show advected particles.

BBC 883-2585

Since the particles move in the interior, part of the torus must be clipped away to see them. Two types of clipping are implemented. In both types, all graphics primitives are tested to see if they should be clipped. Care must be taken to turn clipping off before rendering the particles unless it is known beforehand that they are all behind the front clipping plane. Otherwise particles will wink in and out of existence as they pass through the clipping plane.

The first type of clipping implemented was the standard clipping which is part of 3D viewing [13]. This "viewing" clipping is done against the front of the view volume. The part of the object which is not clipped is mapped into the view volume as part of the viewing operation, in which, among other things, the object is rotated and prepared for projection to 2D. Viewing clipping is inappropriate for displaying particles inside the torus if rotation of the torus is used to provide depth cueing. This is because there is not a fixed relationship between the torus as it rotates and the viewing clipping plane. As the torus is rotated, part of it moves behind the viewing clipping plane. No longer clipped, that part hides the particles in the region behind it [Fig. 3.7]. A different type of clipping had to be devised in which a fixed part of the torus is clipped no matter what the rotation [Fig. 3.8] [Appendix C, Fig. C3.1]. The implementation of this "object-oriented" clipping is described in Section 3.2.4.

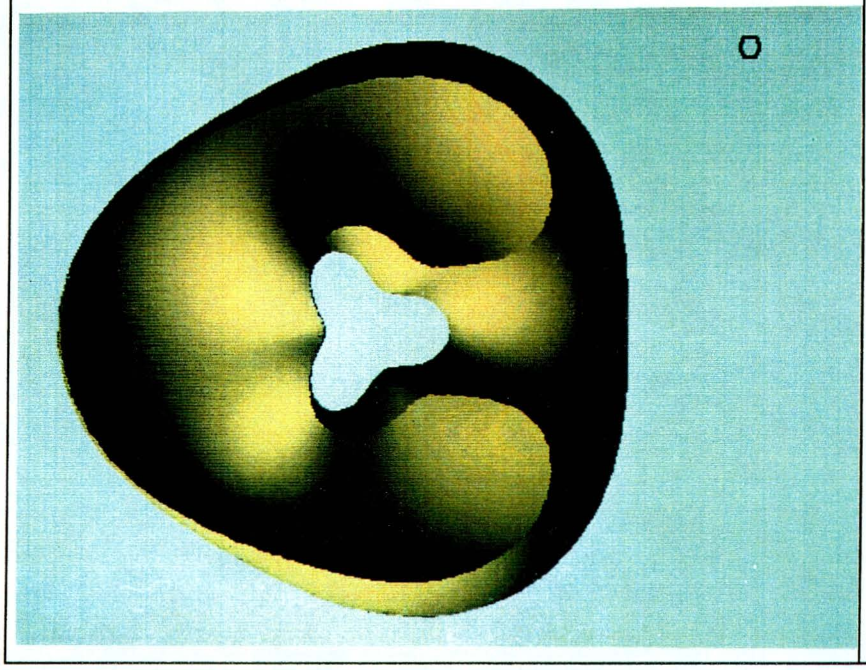


Fig. 3.7. Result of rotation in conjunction with viewing clipping. Note region at right is not clipped away.

BBC 884-4341

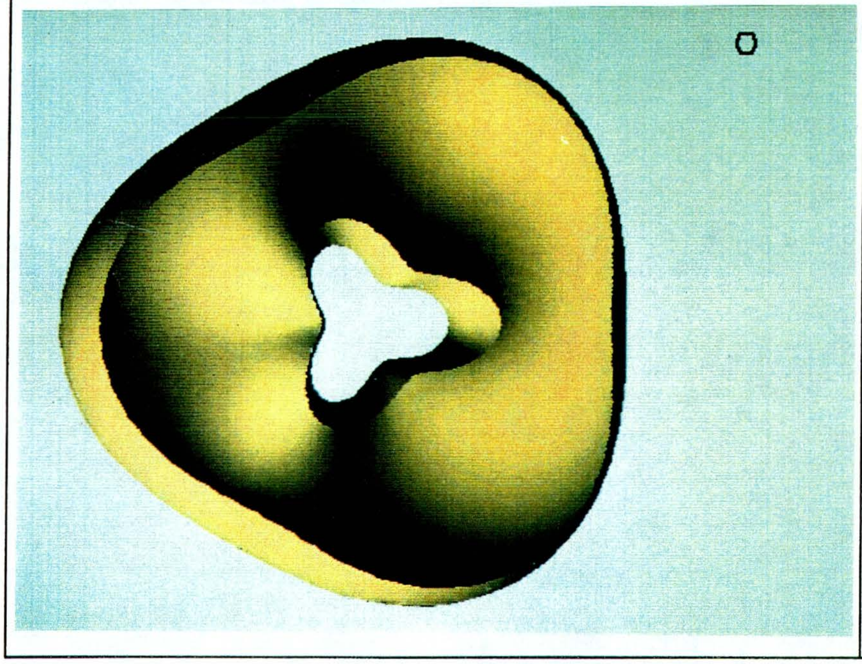


Fig. 3.8. Result of object-oriented clipping with same rotation specified as in Fig. 3.7.

BBC 884-4339

Another problem involved with displaying particles inside the torus has to do with the type of polygon shading used for the polygons that define the torus. Two types of shading are implemented. In constant shading, a polygon is given one color intensity depending on its geometric relation to the light source. This gives the object made up of polygons a faceted appearance. When looking at some instances of clipped objects with this type of shading, the interior appears to "pop out", that is, its curvature appears to be reversed. This happened, for example, to the interior of the torus when it was triangulated using the Mosaic package [Fig. 3.9]. Particles appear to be moving outside instead of inside the torus. Gouraud shading interpolates the normal vectors of adjacent polygons to provide a continuous shading across the surface. This gives a smoother appearance to the object and alleviates the problem [Fig. 3.10].

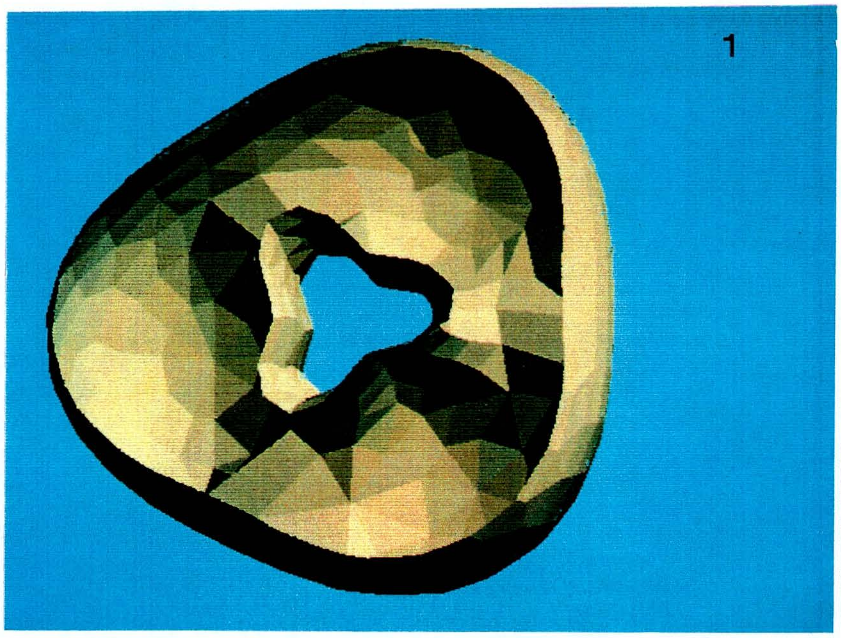


Fig. 3.9. Constant-shaded torus. Polygons arrived at by Mosaic tessellation. Note ambiguous region at upper left.

BBC 884-4319

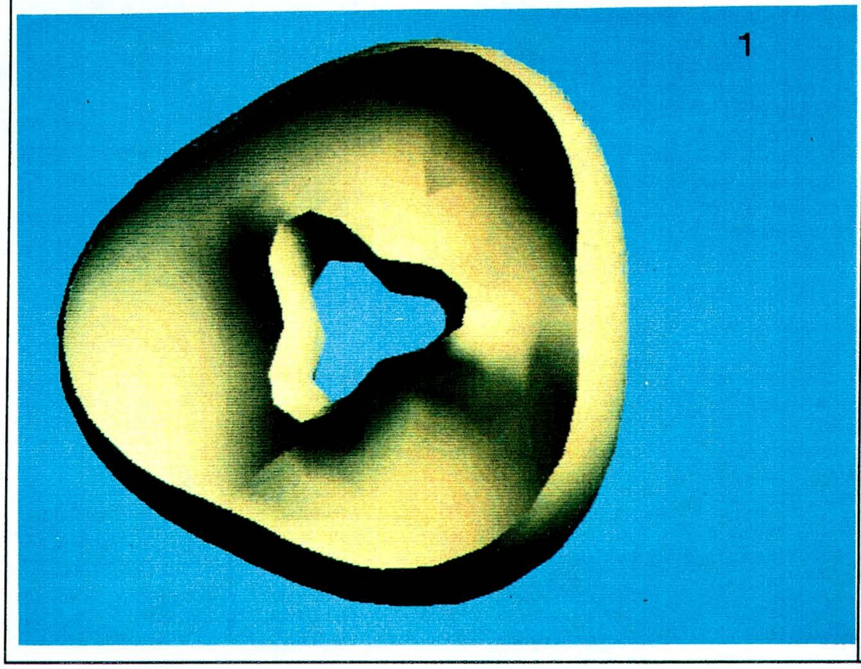


Fig. 3.10. Same as Fig.3.9 with Gouraud shading. Polygons arrived at by Mosaic tessellation.

BBC 884-4321

3.2. Implementation Issues

The starting point for the display of polygons is a file containing a list of the polygons and their vertices, which are given in world (or user) coordinates. Table 3.1 outlines the steps taken to arrive at the final shaded image. Rendering an object made up of polygons entails conversion of a geometric description to a raster (scan-line) format. Rendering requires:

1. 3D viewing: establishment of the view of an object from a particular eye position, and clipping of the object to a finite volume.

2. Projection from 3D to 2D.

3. Scan conversion: the process of determining which pixels of the display will be used to represent each polygon.

Table 3.1.--Rendering steps.

		coordinates		
		x	y	z
1.	object given	world	world	world
2.	find polygon normal	world	world	world
3.	normalize to unit cube	viewing	viewing	viewing
4.	project	viewing	viewing	viewing
5.	convert x and y to NDC	NDC	NDC	viewing
6.	convert x and y to device coordinates	device	device	viewing
7.	scan conversion	device	device	viewing

```
for each scan line crossed by polygon
    for each pair of active edges
        for each pixel between left and right edges
            check z buffer to see if pixel hidden
            if not hidden
                find RGB color based on lighting and shading model
                write pixel color into frame buffer
                write pixel z value into z buffer
        end
    end
end
```

4. Hidden surface removal: the determination of which parts of an object made up of polygons are visible from the given eye point.

5. Lighting and shading: the determination of what color to assign each pixel of the object to account for the position of a light source and the nature of the surface that is represented by, say, the polygons of a tessellated torus.

During this process the vertex coordinates undergo transformations between several coordinate systems. Vertices are transformed from world to viewing coordinates in preparation for clipping and projection. In the case of parallel projection, the unclipped portion of an object transformed into viewing coordinates is confined within a unit cube. The mathematics involved in this transformation and the projection step are described in Foley and van Dam [13] and Wishinsky [48].

NDC in Table 3.1 refers to normalized device coordinates, which is a device-independent means of mapping an object to a portion of the hardware display screen. Device coordinates are the integer, physical coordinates (i.e. number of pixels in the x and y direction) of the hardware display [13]. The z component of a vertex is not transformed beyond viewing coordinates, since the further transformations are 2D. Once the x and y components are transformed to device coordinates, the processes involved with scan conversion begin.

3.2.1. Scan conversion and hidden-surface removal

The first graphics algorithm implemented, a modification of the scan line z-buffer algorithm, performs scan conversion and hidden-surface removal. Scan conversion find what pixels are to be used to represent a graphics primitive and proceeds by finding the intersection of the raster scan lines with the graphics primitives. In the case of a polygon this process generates a filled area in its interior. An efficient method uses a y bucket sort and an active edge list. When the polygon routine is called, it finds information about the polygon such as the number of scan lines intersected, and the top scan line intersected by each of its edges. The data for each edge is placed in an ordered list using a y bucket sort based on its "highest" y point. After the information for all edges has been placed on this ordered list, a separate call is made to perform scan conversion.

Moving only once from the top to the bottom of the polygon, when a scan line is visited, a check is made in the corresponding y bucket for newly "active" edges (first intersection with a scan line). Fig. 3.11 gives the general idea (the algorithm is more complex than illustrated, to handle special cases).

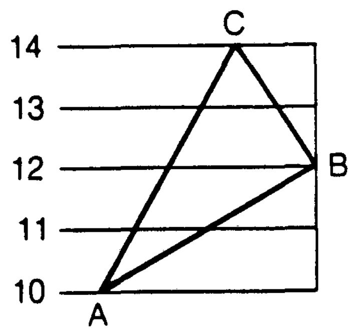


Fig. 3.11. Polygon to scan convert.

The process begins at scan line 14, the highest scan line intersected by the triangle. For each scan line crossed by the polygon a check is made for new active edges, i.e., edges whose upper vertex intersects that scan line. In the example, edges $\overline{AB}$ and $\overline{BC}$ are initially identified as active. Scan conversion between pairs of active edges is performed using incremental calculations from scan line to scan line and from left intersection to right intersection on a scan line. When the count of scan lines for an edge exceeds a pre-calculated number of scan lines crossed by the edge, it is dropped from the active edge list. At scan line 12, edge $\overline{BC}$ drops from the active list and $\overline{CA}$ is added. If no edges in the polygon are active scan conversion is complete. At scan line 10 the triangle has been scan converted.

As a polygon is scan converted, hidden surface removal using a z-buffer is also being performed. Hidden surface removal is the elimination of parts of an object not visible from the eye point. A z-buffer is an in-memory buffer holding the z, or depth, values of each pixel in the frame buffer (which holds the color of each pixel). The z-buffer and the frame buffer are identically-dimensioned arrays the size of the raster image which appears on the screen. When the process of scan converting a polygon gives a pixel to be displayed, the depth of that pixel is compared to the corresponding z value in the z-buffer. If the depth of the pixel in the z-buffer is greater than that of the current pixel, the current pixel is in front of the previously written one, in which case the current depth is written into the z-buffer. The color of the current pixel is written into the corresponding location in the frame buffer [37].

The z-buffer is the simplest hidden-surface removal algorithm. Implementing it requires only a few lines of code. It easily solves special cases such as interpenetrating polygons [Fig. 3.12] and can handle hidden point and line removal as well. The price is the amount of memory required for the z-buffer [37]. In the case of the 512 x 400 frame buffer used for video animation, the z-buffer is 0.8 megabytes in size.

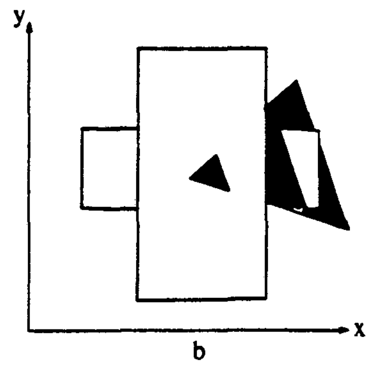


Fig. 3.12. Triangle interpenetrates the two rectangles. Reprinted, by permission, from Rogers [37], 260.

A major motivation for the original developers of the scan line z-buffer algorithm was the fact that the frame buffer and the z-buffer only have to contain one scan line [37]. In the unmodified version a prepass is made for all polygons, during which they are stored in y buckets. An active polygon list is used in addition to an active edge list. Scan conversion proceeds from the top scan line in the frame buffer to the bottom, visiting each scan line only once. However, if particles are advected, the points have to be sorted in order to go through the image a scan line at a time. In addition, sufficient intermediate storage is required for each polygon when they are stored in preparation for scan conversion that scan converting large numbers of polygons becomes impractical due to lack of space. Thus the software frame buffer and the z-buffer are the size of the hardware frame buffer used for display. Points, lines, and polygons are rendered one at a time in random order.

3.2.2. Lighting model

A lighting model simulates light impinging on an object from various directions and reflecting from it. The simplest model, using a version of Lambert's cosine law, was implemented. Lambert's law models diffuse reflection, in which "light is scattered equally in all directions" [37]. It takes the form:

$$I = I_l k_d \cos(\theta), \quad 0 \leq \theta \leq \frac{\pi}{2} \tag{3.1}$$

where I is the intensity of the reflected light, I_l is the intensity of the light originating from a point source at infinity, k_d is a constant which determines the amount of light reflected by the object, and θ is the angle between the direction of the parallel incoming light rays and the normal to the surface (in this case, polygon). If θ is greater than $\frac{\pi}{2}$, that polygon is hidden from the light source and is a special case [37]. The calculation of the intensity was simplified by setting I_l and k_d to 1 [Fig. 3.13].

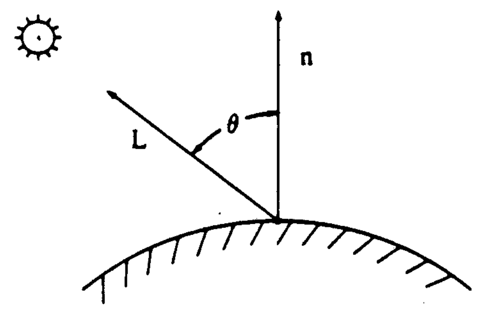


Fig. 3.13. Lambert's law models diffuse reflection. Reprinted, by permission, from Rogers [37], 312.

The direction of the light rays is specified by the light source vector. Specifying it in vector form, e.g. *(0.8, 0.2, -0.7)*, is highly non-intuitive. Instead, a specification of a rotation in 3D space of a unit vector is used to arrive at the light source vector. For example, *Seltsrc (45, 45, 45)* specifies a unit vector rotated 45 degrees in x, y, and z, in the same manner that *Sparall (45, 45, 45)* specifies the rotated view point.

Using this specification of the light source in conjunction with rotation of the object is not always satisfactory. Rotation is achieved by moving the eye point around the object [Fig. 3.14] during the viewing step. As the eye point moves around the object from A to C, it arrives at a point where only the side hidden from the fixed light source is visible. Since in this simple model all polygons hidden from the light source are colored the same (black), the result is an undifferentiated dark mass [Fig. 3.15] [Appendix C, Fig. C3.2].

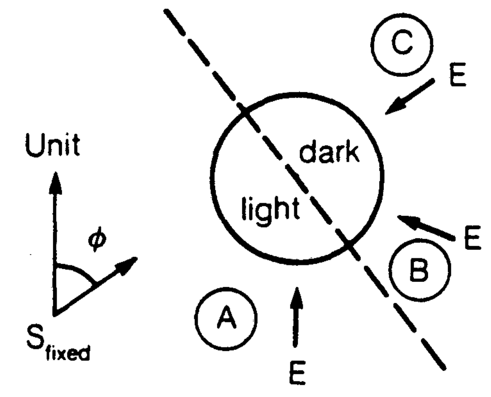


Fig. 3.14. Rotation achieved by moving eye point from A to C. Light source eclipsed at C.

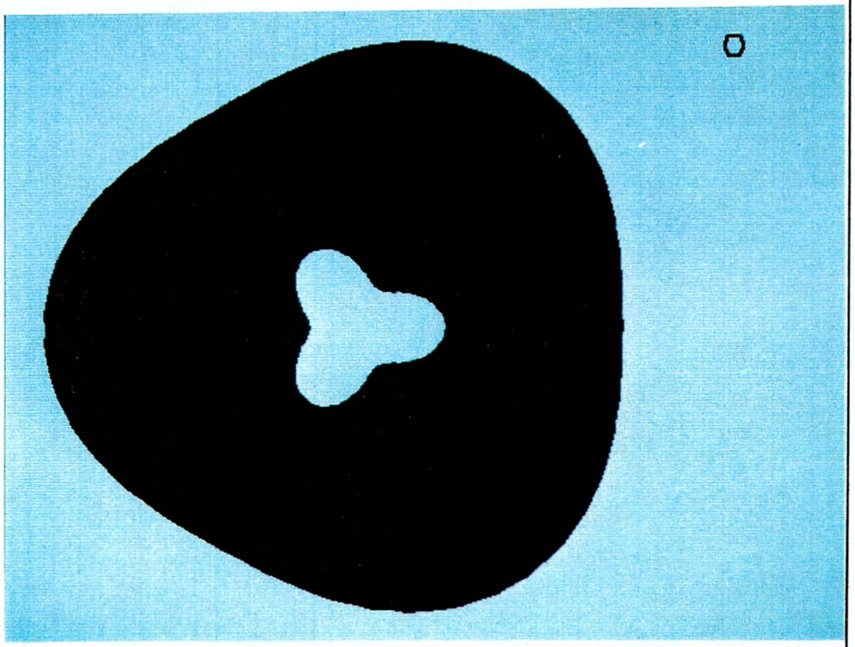


Fig. 3.15. The blob.

BBC 884-4297

If the relationship of the light source to the eye point is held constant as the eye point moves around the object, the above problem should not occur (if it does, it is likely that a faulty specification of the light source is made). In Fig. 3.16 the light source is specified to the left and in front of the torus. The lower left side of the object on the screen faces the light no matter what the movement of the eye point [Appendix C, Fig C3.3]. This mode of lighting is achieved [Fig. 3.17] by rotating the unit vector to arrive at the light source vector $\vec{s_1}$ as before, and then performing a second rotation β to arrive at $\vec{s_2}$. β is the same as the rotation specified to move the eye point from the original unrotated eye point $\vec{e_1}$ to $\vec{e_2}$.

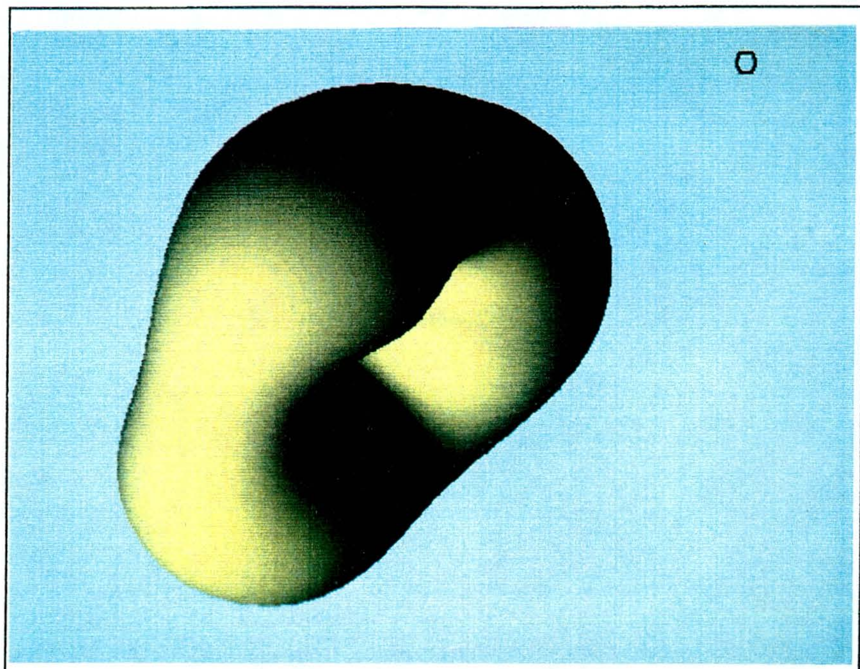


Fig. 3.16. Light source to the left and in front of the torus.

BBC 884-4335

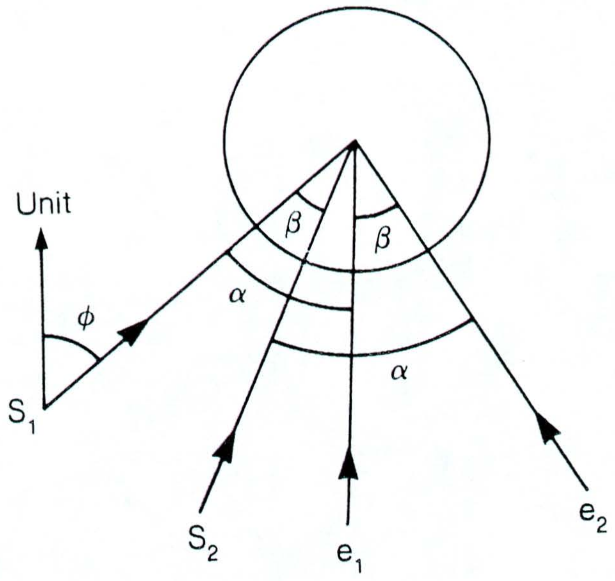


Fig. 3.17. As eye point is moved from $\vec{e_1}$ to $\vec{e_2}$ by angle β, light source is also moved from $\vec{s_1}$ to $\vec{s_2}$ by angle β to remain at the same angle α with the eye point.

The angle θ between the light source vector and the surface normal determines the intensity of the reflected light. θ is not found directly. Instead, $\cos(\theta)$ is found using the equation [34]:

$$\cos(\theta) = \frac{\left(\sqrt{\text{norm}_a{}^2 + \text{norm}_b{}^2 + \text{norm}_c{}^2}\right) \cdot \left(\sqrt{\text{src}_a{}^2 + \text{src}_b{}^2 + \text{src}_c{}^2}\right)}{\text{norm}_a\text{src}_a + \text{norm}_b\text{src}_b + \text{norm}_c\text{src}_c} \tag{3.2}$$

where norm_a, norm_b, and norm_c are the components of the surface normal, and src_a, src_b, and src_c are the components of the light source vector.

The normal to a polygon in 3D space is perpendicular to the plane of the polygon, and is specified by three components a, b, and c, which are coefficients of the plane equation:

$$ax + by + cz + d = 0 \tag{3.3}$$

The normal is calculated by an algorithm [37] that takes as input the vertices of the polygon. The normal is calculated while the vertices are still in world coordinates, since the various coordinate transformations distort the output of the algorithm.

The ordering of the vertices supplied to the algorithm, clockwise or counterclockwise, is important. The direction of the normal conforms to the right-hand rule applied to the vertex numbering. If the polygon were in the plane of this page, the direction of the normal would be up out of the page if the vertices are given in counterclockwise order, and down into the page if in the opposite order. If the vertices for all polygons are given in a consistent order, normals of polygons hidden from the eye point have opposite sign from those visible. Since the intensity of the light reflected from a polygon is calculated using the angle between the light source vector and the polygon normal, inconsistent order (and thus inconsistent normals) results in a quilt of light and dark patches.

Another problem involving the direction of the surface normal is encountered when the front of the torus is clipped to see the advected particles. This sort of clipping exposes usually-hidden polygons. The lighting model is incorrect for these polygons because the exposed side of a polygon is facing in the opposite direction of the surface normal.

In Fig. 3.18, the normals at polygons B and C face away from the light source $\vec{s_1}$ and are thus given the color black. However, since the front of the torus is clipped away, these polygons are visible from the light source. Another problem occurs at B when the light source is at $\vec{s_2}$. The angle between the light source and the normal is identical at A and B, so there is no differentiation in the intensity of their colors. The result is that the part of the torus ordinarily visible (at A) blends in imperceptibly with the interior (at B).

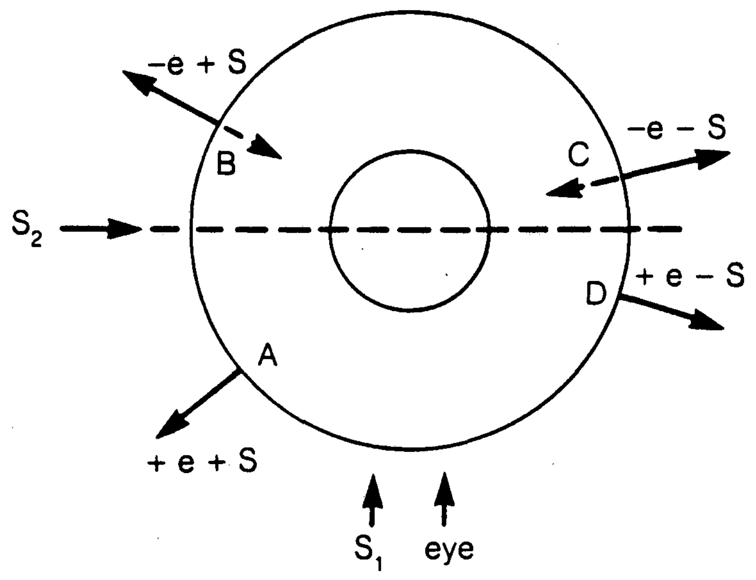


Fig. 3.18. Cross section of cut away of torus. Eye point is at eye. Light source at either $\vec{s_1}$ or $\vec{s_2}$. Solid arrows are polygon normals. $\pm$e refers to angle between normal and eye point. $\pm$s refers to angle between normal and light source.

The solution to these problems takes into account the angle between the polygon normal and the eye point [Table 3.2]. If this angle is positive, the lighting model is used as before. If this angle is negative, a hidden polygon has been exposed by clipping. Reversing the direction of the normal of such a polygon (dotted lines at B and C) eliminates the problems described in the above paragraph. The reversed polygon normals at B and C now form a positive angle θ with the light source at $\vec{s_1}$ and are lit based on $\cos(\theta)$. With the light source at $\vec{s_2}$, the reversed normal at B forms a negative angle with the light source and the polygon is colored black. In implementing this change, instead of reversing the direction of the normal, $\cos(\theta)$ between the polygon normal and the light source is negated if the polygon is hidden from the eye point, and the same effect is achieved.

Table 3.2.--Revised lighting model.

angle between eye point and normal	angle theta between light source and normal	shading
positive	positive	based on cos(theta)
positive	negative	black
negative	positive	black
negative	negative	based on -cos(theta)

It should be noted that this change does not model shadows, though a shadow effect is produced at B when the light source is at $\vec{s}_2$. For example, the polygons to the left of C should be in shadow if the light source is at $\vec{s}_2$, but are not [Fig. 3.19].

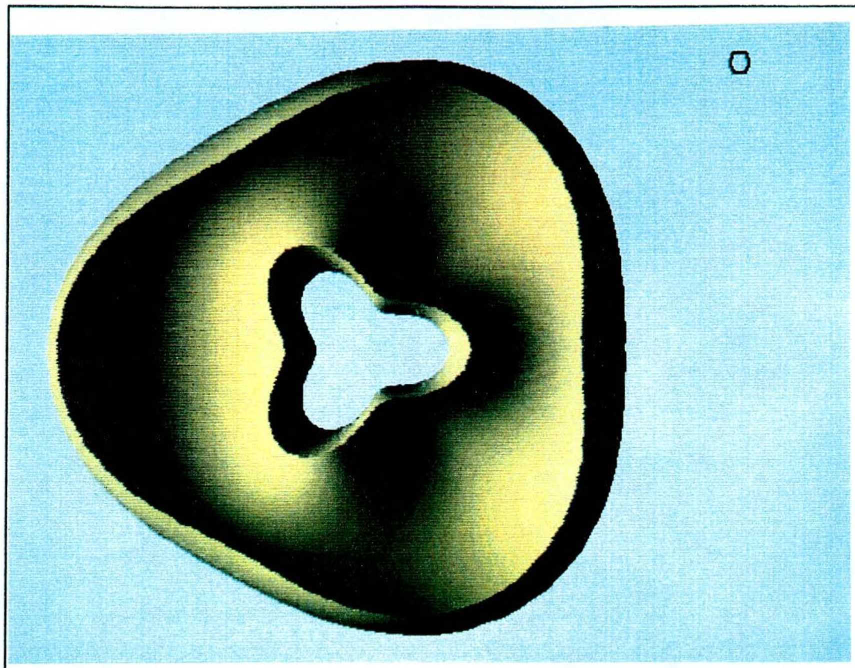


Fig. 3.19. Clipped torus with light source at $\overrightarrow{s_2}$.

BBC 884-4337

3.2.3. Shading techniques

When a polygon is scan converted, each pixel must be colored based on the lighting model. The intensity reflected is represented using the value component of the HSV color scheme described in section 1.2.4, that is, lower intensity is modelled by decreasing the value component of the color of a pixel. Shading techniques involve various geometry-dependent modifications of the intensity across a scan line of a single polygon face. Three techniques for varying the intensity with scan conversion are constant, Gouraud, and Phong shading [37]. As mentioned above, only the first two are incorporated in the current implementation.

Using constant shading, the same color is given to all pixels in the polygon, based on the polygon normal. Thus, individual polygons making up the object can be seen, giving the object a faceted appearance [37] [Fig. 3.9]. In addition to the illusory curvature reversal described in section 3.1, the faceted appearance is detrimental in that it distracts attention from the particles when the torus is rotated. This distraction is caused by the many moving edges of the facets [Appendix C, Fig. C3.4].

Gouraud shading gives a smooth appearance to the object and eliminates much of the distracting discontinuities. Instead of using the single polygon normal to calculate the intensity for the whole polygon, "normals" at each vertex are found. A vertex normal is approximated by averaging the normals of all the polygons incident upon it. The intensity at the vertex is then found using the lighting model. A form of bilinear interpolation similar to that used in advecting particles [see Equation 1.2] is employed to find the intensity at each pixel during the process of scan conversion. Incremental calculations are used instead of using interpolating equations for each pixel. For two active edges $\overline{AB}$ and $\overline{CB}$ [Fig. 3.20], the difference in intensity between A and B, and C and B, is used to find the increment in intensity δy along each edge from scan line to scan line [13]. The increment in intensity δx along each scan line is found by taking the difference in the intensity at the current scan line between the left edge $\overline{AB}$ and the right edge $\overline{CB}$ and dividing it by the difference in x values between the two edges [37].

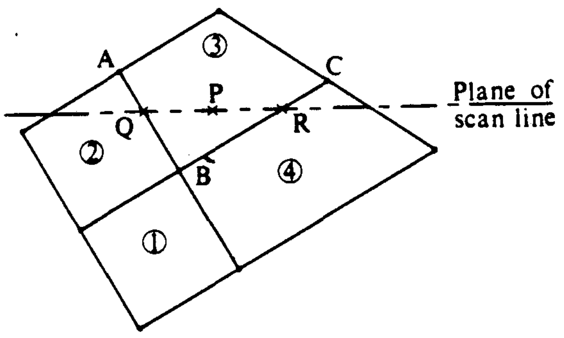


Fig. 3.20. Bilinear interpolation to arrive at intensity component of pixel. Reprinted, by permission, from Rogers [37], 324.

Phong shading finds the vertex normals as before. However, instead of finding the intensity at each vertex and then interpolating in intensity, this technique interpolates to find the normal at each pixel, and then applies the lighting model at each pixel. Phong shading provides a more realistic result than Gouraud, particularly a better representation of specular highlights (for example, the bright spots on a reflective surface such as a metal ball). However, it is also much more computationally expensive. Given the goals of a unambiguous, but not necessarily most realistic, representation of an object, and the desire for speed, this shading technique was not implemented. Gouraud shading was judged to be sufficient for most purposes.

In both Gouraud and Phong shading, the vertex normals must be calculated. To do this, advantage is taken of the fact that the polygon data base read to produce an object consists of a list of all vertices with no vertices duplicated, and a polygon list, where each polygon vertex is indicated by an index into the vertex list [7]. A prepass is made before scan conversion in which the normal of each polygon is found. As mentioned above, a vertex normal is found by averaging (adding) the normals of each polygon incident upon that vertex. An array parallel to the vertex array is used to hold the vertex normals to be built up. When a polygon normal is found it is added to the vertex normals in the normal list at the same positions as the positions of each vertex of that polygon in the vertex list. Since there is no duplication in the vertex list, when all polygon normals have been found and added to the vertex normals, the process of calculating all vertex normals is complete.

Once the vertex normals are found, Gouraud shading performs interpolation in intensity as described above. A problem arises because of the limited number of intensities of a single color provided by the particular hardware frame buffer used (5 bits for intensity of each primary color). Color aliasing gives a clearly-visible scalloped appearance [Fig. 3.21]. Here for once NTSC encoding provides an advantage. Its blurring effect alleviates scalloping except when polygons with areas that are large compared to the frame buffer are used, as in the "top" [Appendix C, Figs. C3.3, C3.5].

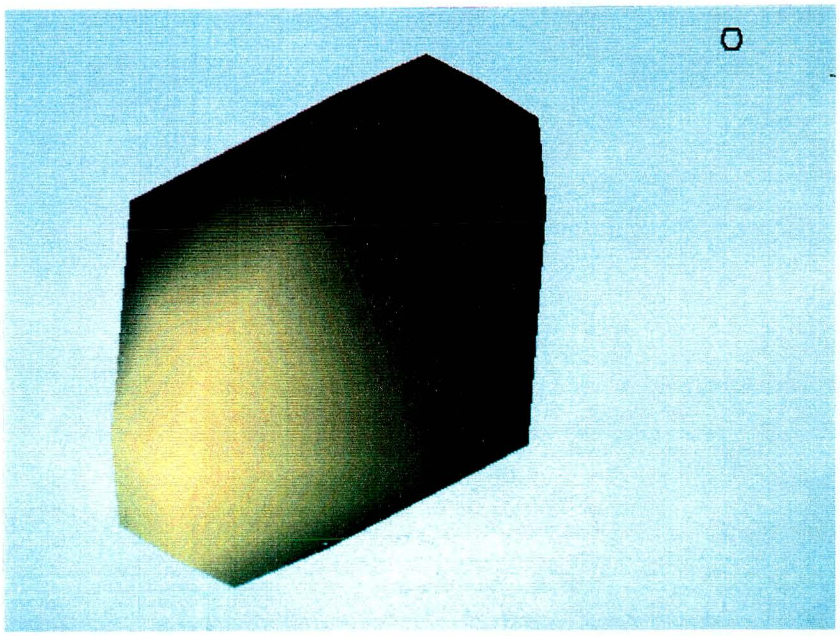


Fig. 3.21. Gouraud-shaded image of "top". Scalloping due to color aliasing.

BBC 884-4299

Another problem occurs because colors are calculated in the HSV color model, but the hardware frame buffer expects an RGB representation of color. The conversion is simple [42], but an execution profile of the program performing Gouraud shading pointed out that most of the CPU time was being spent in the HSV to RGB conversion routine. With Gouraud shading, each pixel may have a different value and hence the conversion routine was called for each pixel, i.e. millions of times in producing a sequence of frames.

Using a table with precalculated RGB values as entries eliminates this problem. The user specifies a HSV representation for a particular color table entry, which is then converted to RGB. The only component of the polygon color modulated during Gouraud shading is the value (intensity). The intensity range of the hardware frame buffer is limited; as mentioned above, there are only 5 bits each for the red, green, and blue components. When a HSV representation is entered into the color table, the RGB representation is found for 128 intensities with that particular hue and saturation. In the case of the torus, all polygons are given the same HSV values. Thus instead of performing the HSV to RGB conversion routine millions of times, it is done only 128 times when a color is entered into the color table. During scan conversion, the intensity of a pixel is used to perform a simple table lookup. Using a color table saves around 20 seconds per frame for the clipped torus.

3.2.4. Clipping

The front of the torus must be cut away to see the advected particles. Rather than trying to clip 3D polygons, which is a very complex process, once again an approximation is resorted to which is appropriate to the final display: a video movie. A prepass part way through the viewing pipeline identifies completely clipped, not clipped, or partially clipped polygons with respect to the clipping plane [Fig. 3.22].

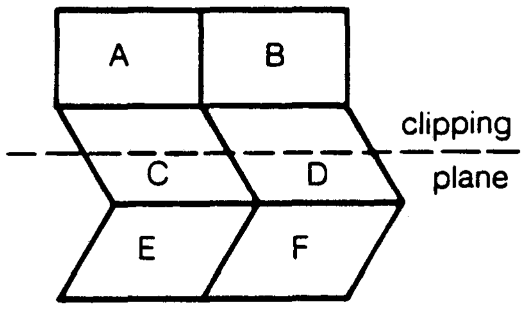


Fig. 3.22. Polygons A and B unclipped, C and D partially clipped, and E and F clipped.

This identification is made using the Cohen-Sutherland line clipping algorithm [13], which takes advantage of the fact that the object is at this point in viewing coordinates. The portion that will not be clipped has been mapped into a unit volume [see Table 3.1]. If any of the edges of a polygon intersect the clipping plane (the front of the unit volume), that polygon is identified as partially clipped. Fully clipped polygons are discarded at this point.

During scan conversion, each pixel in a partially clipped polygon is tested to see whether it lies in front of or behind the clipping plane. In the case of the parallel projection used in this implementation, the z coordinate of the front of the unit volume (the front clipping plane) is zero. Thus this test only involves a comparison against zero. However, this simple mechanism is inappropriate for displaying particles inside the torus. As described in section 3.1, particles are hidden by part of the object moving behind the front clipping plane when the torus is rotated. To deal with this, clipping is done against a clipping plane which is kept in a fixed relationship to the geometry of the torus (irrespective of the rotation) from one frame to the next ("object-oriented" clipping).

The prepass identifies partially clipped polygons as before. Three points in world coordinates on the clipping plane are used to find the equation of the plane. After the prepass, as the torus is rotated the three points are rotated also and are used to find the equation of the rotated clipping plane during the scan conversion process.

Equation 1.6 finds the distance of a point from a plane. If the distance of a pixel to the clipping plane is negative, that pixel should be clipped, since it is in front of the rotated clipping plane. As those polygons that were identified as partially clipped are scan converted, each pixel is tested to see if its distance from the plane is positive, and only if so, its color placed in the frame buffer. Equation 1.6 is modified to:

$$\mathrm{clip} = -\left(\mathrm{ax} + \mathrm{by} + \mathrm{cz} + \mathrm{d}\right) \tag{3.4}$$

since the denominator in 1.6 is always positive and only the sign of the result matters.

This method is not as inefficient as it might seem, for unless the object is a honeycombed structure, only a relatively few polygons are identified as partially clipped. Its main drawback is that the object must be in the proper orientation in world coordinates to use the viewing pipeline front clipping plane at *(0, 0, 0)* rotation as the clipping mechanism. This is completely outside of the 3D viewing model, and represents an added (ad hoc) complexity at the user interface. This drawback might be rectified if a prepass modelling transformation [13] on the object in world coordinates were performed to rotate the object to the desired orientation for clipping.

Chapter 4

Video Recording

4.0. Introduction

Properly understanding the information in a sequence of flow fields requires graphical display at speeds at which the viewer can understand the temporal evolution of the flow, i.e. at least 10 to 15 frames per second. The data must be either displayed in real time or recorded a frame at a time on video or film for later playback. The workstation used is generally not powerful enough to generate images in real time except for the display of the tracks of a few 2D particles. The approach taken for this thesis was to use video recording almost exclusively.

Watson, et al. [47] describe a system in which not only real time display but interactive walk-through of a flow field is possible. This use of interactivity, along with high-quality video recording for permanent storage, is obviously a good way to understand the results of fluid modelling. Unfortunately, the expense of this system is beyond the reach of the funding available to many scientists. Another drawback is that even this system cannot render complex Gouraud-shaded objects such as the space shuttle shown in Fig. 3.5 in real time.

Recording using the video workstation offers the ability to inexpensively display a sequence of highly complex images at real-time playback speeds. Two different mechanisms of recording are used, videotape and videodisk. The price of the videodisk unit is roughly twice that of the videotape recorder (VTR). The device drivers for the two recording units are described in Johnston, et al. [20] and Johnston, et al. [21] and were integrated into the distributed graphics package.

4.1. Advantages and Disadvantages

Video recording offers many advantages. As mentioned earlier, the time to arrive at a finished product is much less than when using film. Video allows the dynamic and inexpensive display of complex 3D smooth-shaded objects, and scenes with tens of thousands of advected particles. It allows repeated viewings of flows that are too complex to grasp as a whole, without tying up CPU time for each viewing. It provides a permanent record that can be viewed anywhere a home VTR is available, and under normal lighting conditions. With a more sophisticated VTR or videodisk, playback is possible at slower speeds, a single frame can be held for closer inspection, and the movie can be reversed if desired without image degradation.

One of the main disadvantages of video is the use of NTSC encoding. To use NTSC encoding, the resolution of the image should ideally be 525 x 486. If the resolution is higher, it must be reduced [10]. In addition, as mentioned in section 1.2.3, the spatial resolution of colors such as blue is limited, and there is a complex interaction between various-sized colored swatches. A higher-quality (but more expensive) means of generating a NTSC signal would somewhat mitigate the latter two problems.

A disadvantage of recording on videotape is the slow recording process, typically from 10 to 20 seconds per frame. When using the videodisk, however, a frame can be recorded in 0.5 seconds. At 0.5 seconds per frame it is possible to record a minute's worth of video in 5 minutes, at 10 frames a second. A disadvantage of videodisk is the cost ($130) of the recording media.

4.2. Previewing

Given the time necessary to record an image in the case of videotape and the expense of the recording media in the case of videodisk, it makes sense to preview a movie before recording it. Bugs in the images can be identified and corrected beforehand, saving time and/or money. Previewing was especially valuable when developing new graphics software, and in integrating the compression algorithms. When the preview option is set, the image is displayed as before but recording does not take place. As mentioned in section 2.4, the process of placing the image a scan line at a time into the hardware frame buffer is overlapped with advection and rendering, thus saving several seconds for each frame previewed.

Previewing still takes too long for images such as the torus. The speed of the CPU performing the rendering is the main bottleneck. The video workstation can be upgraded by using a more powerful computer as the front end. If that is not available it would be a simple matter to select and display periodic images from a movie. This strategy is based on the assumption that many mistakes will not occur at isolated frames but will have temporal continuity. Another option is to display only polygon edges and not perform scan conversion [3].

Some mistakes, such as discontinuities in motion, can not be caught until the movie is in the process of being recorded [Appendix C, Fig. C4.1]. Previewing has to occur at close to real time to catch this type of problem. The preroll mechanism on the VTR turns out to be useful in identifying these mistakes before an entire movie is made and thus wasted.

The preroll on the VTR brings its complex inner mechanisms necessary for single-frame recording up to speed before the actual split second of recording. This preroll accounts for much of the time necessary to record a frame on videotape. While the preroll is occurring, the last several seconds recorded are played back for each frame. While recording a movie it is only necessary to check the process during the preroll every 15 minutes or so to be sure that mistakes such as motion discontinuities have not occurred.

Recording on videodisk does not require a preroll, thus contributing to its 0.5 second recording speed. However, none of the movie is seen at video speed until the entire movie is done. For this reason a simulated preroll every 20 frames or so is incorporated into the videodisk driver.

4.3. Videotape vs. Videodisk

Recording on videodisk is more than an order of magnitude faster than recording on videotape. In addition, videodisk is random access; any portion of it can be accessed in 0.5 seconds [31]. Videotape is sequential; to access frame 20000 it is necessary to pass through all the preceding frames.

It is a simple matter to play back a movie recorded on videodisk at any integral multiple or fraction (i.e. 1/3, 1/4) of the normal playback speed, either forward or backward [31]. The VTR has only a few speeds available; to play back at exactly one half speed it is necessary to record an image on two consecutive frames. This problem can be mitigated (obviously at additional expense) by acquiring a VTR that has more sophisticated playback facilities but which does not have the ability to record a frame at a time. Tapes produced with the recording unit can be dubbed (copied) onto tapes that can be displayed by a playback-only unit.

The videodisk unit can also be programmed to hold a frame for any length of time. It was found to be desirable to hold the first and last image of a movie for several seconds; to do this is trivial with the videodisk unit, but the image must be recorded on videotape on several seconds' worth of frames to achieve this effect.

A videotape must undergo a process called blackstriping before it can be recorded on. To be able to record a single frame, each frame has a time code, giving its absolute position on the tape, recorded in one of the two audio tracks. Blackstriping lays down this time code sequentially from the beginning to the end of the tape. It does this in real time, and so takes about one half hour to an hour depending on the tape length [46]. Videodisks are preformatted and do not require this preparation.

A drawback of videodisk is that it is Write Once Read Many times (WORM). If 500 frames are recorded and the 150th frame contains an error due to a bug in the movie-making program, it is necessary to go through and record all 500 frames again since a videodisk frame can not be recorded over, an unfortunate prospect since the videodisk media is expensive. The videodisk unit could be programmed to play from frame 1 to 149, jump to a newly recorded ungarbled frame, and then jump back to frame 151. The problem is that with the 0.5 second access time, there would be a noticeable glitch in the video signal at that point.

In practice, however, this problem has not been so severe. Five hundred frames are not recorded in one session unless the movie has been well debugged and previewed. Videotape was used for almost a year before the videodisk was acquired. Mistakes were such that it was often necessary to re-record the whole section of movie. For example, if particles are being advected it is necessary to start again from the last checkpoint (see section 1.2.3) to arrive at their state at a glitched frame.

It is not always possible to record over a frame on videotape either. For unknown reasons, a glitch occasionally occurs in the recording process, and a frame is skipped, causing a highly-noticeable blank spot when the tape is played back. This blank spot can almost never be recorded over. It is hypothesized that the time code on such a frame has been corrupted in some manner, preventing both the original recording and subsequent attempts.

A disadvantage of videodisk is that an image is noisier than a videotape one. This noise is more noticeable in 3D, shaded images than in 2D ones [Appendix C, Fig. C4.2]. It also contains occasional horizontal colored spots and streaks called dropouts, resulting from a recording error due to a media defect. In addition, the faster recording time is inconsequential for images that take longer to render than the videotape recording speed. Since recording is overlapped with computation, these images take the same time to render and record on videodisk as on videotape. However, what takes 20 seconds on one CPU may take 2 on another, in which case this equality is eliminated.

Given the fast recording time and ease of use, in cases where a higher quality image is not essential, recording on videodisk has been preferred. If material were to be presented at a conference a workable strategy would be to fine tune the movie on the videodisk and then record once on a rented unit.

4.4. Ancillary Information

A video movie presents many megabytes of data, but without ancillary information such as frame numbers and titling its usefulness is diminished. The same data may be recorded in several different ways. For example, one movie may show the whole range of the data set while another may zoom in on a part of it. A frame number identifies corresponding images in the two movies. It also indicates exactly where a problem occurred if a movie is made without someone to continuously watch over its production.

Titling is used in the finished form of the movie. It indicates in summary form the modelling involved in producing the data as well as credits and funding information. It should be as concise as possible since the point of using this media is to display pictorial information. An accompanying paper should provide the details of the modelling and its implications.

A video movie is as much a form of scientific data as the numbers produced by the modelling, so it should be reproducible. Enough information should be permanently stored to be able to remake the movie. One form of accomplishing this is to have every parameter that is set, such as the colors of graphics primitives, the position of the clipping plane, the light source, etc., recorded in a log for the movie. This process could be accomplished by having a parameter-setting routine write to a file whenever it is called if a "log" command-line option is given. Alternatively, in more sophisticated animation packages, a file of commands is used to drive the making of the movie [3], in which case it is only necessary to save that file.

A log is only useful if a general purpose movie-making program is used, not an ad hoc program for each movie. Otherwise the entire program must be stored to be able to reproduce a movie. Reproducibility was a problem during the course of this thesis, since the modules involved in the distributed graphics package were under constant development. Some movies, while pointing out interesting errors, could not be remade at this time without considerable effort. The moral is to use a program that has been well-tested and is no longer in development when recording the final form of a scientific movie.

Chapter 5

Conclusions

5.1. Evaluation

The major design goals guiding the writing of the software for the distributed graphics system used for flow visualization were (1) achieving rapid image generation from, and recording of, the data; (2) finding the minimal implementation providing an unambiguous and illustrative graphics display; (3) gaining the ability to visualize important features of the flow starting from the information contained in flow fields; and (4) when finished, having a set of tools usable for the display of a wide variety of scientific data. Were (and how were) these goals met?

5.1.1. Distributed Graphics Package Implementation

Rapid image generation and recording is essential for "discovery" graphics, i.e. graphics whose purpose is insight and not that of providing a polished, finished product. "Rapid" does not necessarily mean close to real time; recording a video movie overnight is more than ten times faster than other techniques such as film recording are able to achieve. In practice, the video movies referenced in this thesis took from several minutes to around ten hours to record. Alternative algorithms were not implemented to discern whether the algorithms that are implemented are any faster; however, many measures were taken to improve the rate of image generation and recording over the history of the development of the algorithms for the distributed graphics system described herein.

The first major change made to improve the rate of recording was the porting of the server portion of the Sun RPC library to the PC. Instead of the three-step process of generating a metafile on the front end, transferring the metafile over the network, and then reading the metafile and generating the graphics calls on the PC, a one-step process occurs in which the data is automatically sent across the network and displayed and recorded. Recording the full 760 frames of the backward facing step movie using the three-step approach took about two days; using the RPC server approach, it took about 1/7 the time: a little over an hour real time to record 100 frames.

The recording process on the VTR took about 10 seconds (the preroll was later increased to reduce the number of glitched frames, increasing the recording time to about 20 seconds). To allow the process of recording to be overlapped with computation on the front end, the server on the video workstation was changed to send an acknowledgement back to the front end before commencing the recording process. This change saved about 20 minutes real time in recording 100 frames of the backward facing step movie.

The use of compression of the frame buffer saves at least 10 seconds per frame real time (sometimes more depending on the image) because much less data is transmitted over the local-area network. Over a wide-area network with a 2K per second effective bandwidth, the savings are much more substantial. It would take over 200 seconds to send the whole 400K image over a wide-area network, but a typical compressed image of from 10 to 20K would only take five to 10 seconds to transmit. The use of compression also allows the building up of the image in the hardware frame buffer to be overlapped with execution, saving several seconds per frame in both previewing and recording (see section 2.4).

In the 2D case, creation of a software frame buffer on the front end machine, instead of the metafile approach, saves time because many fewer graphics calls are made to the hardware frame buffer, which is driven by a slow processor. As mentioned in the preceding paragraph, display generation is overlapped with graphics generation on the front end. Also, a raster image compresses much better than the graphics primitives that it represents.

Previewing a graphics representation saves time (in the case of the VTR) and money (in the case of the videodisk). It allows mistakes and poor choices in the design of the graphics representation to be caught before the recording process takes place. Another feature, the preroll, enables correction of mistakes that are only seen at video speed, before the whole movie is made.

Recording became much easier when a video optical disk was added to the system. Recording with the optical disk takes 0.5 instead of 20 seconds (with the VTR). Improvements in software that had caused faster previewing time now also caused faster recording time. For example, generating a 2D image of the backward facing step (500 particles per injection) in the software frame buffer and compressing it on a front-end workstation takes only a few seconds, after which the front-end has to wait if the VTR is being used. With all the improvements mentioned above, it still takes at least 35 minutes to record 100 frames of the backward facing step. Using the videodisk, the video workstation is idle instead of the front end. The same 100 frames now takes 4 1/2 minutes.

In addition to rapid image generation and recording, an unambiguous presentation of the data is necessary. For the display of the clipped torus, achieving this goal requires Gouraud shading. Using constant shading has the potential of causing the curvature reversal described in section 3.1. Furthermore, the lighting model has to be adjusted to account for normally-hidden polygons that become visible when the front of the torus is clipped away. Otherwise there is the possibility that a large portion of the interior will be shaded ambiguously, i.e. be assigned all black or merge into the exterior portion of the torus.

The minimal implementation necessary to display the data was also strived for, for reasons of both speed of execution and programming constraints. The potentially expensive and most difficult algorithms to implement are those necessary to display a shaded 3D surface with its hidden portions removed. The simplest means of hidden surface removal, the z-buffer, is used. Keeping the z-buffer and frame buffer a fixed size, expense increases at most linearly with additional polygons. As the number of polygons in an image increases, the area of each polygon decreases. There are thus fewer pixels to be tested against the z-buffer for each polygon [37]. A rough comparison of various hidden-surface removal algorithms shows that the z-buffer is more expensive for small numbers of polygons (100), but less expensive for large numbers (60,000) [13].

The simplest version of the simplest lighting model, that using Lambert's law, is used. Ambiguity is high if a fixed light source is used and the eye point is moved to the side of the object facing away from the light (the result is a featureless mass). Using a specification of the light source relative to the eye point eliminates much of the difficulty, though that part completely in shadow is featureless. However, rotation brings the parts in shadow to a point where they are not hidden from the light source.

Constant shading is inadequate: the curvature reversal mentioned above occurs, and the many moving facets are distracting when displayed at video speeds. Gouraud shading does not produce as realistic a result as Phong shading. However, Phong shading is more computationally expensive. Gouraud shading is judged to be sufficient for the purposes of accurately illustrating the geometry of the object in most cases.

Another simple algorithm is used to clip 3D polygons. A test is made for each pixel during scan conversion of a polygon previously identified as partially clipped to see whether the pixel should be clipped or not. Two versions of the algorithm are used. "Object-oriented" clipping requires solving the equation for the distance from a plane for each tested pixel, while "viewing" clipping only requires a comparison against zero. Object-oriented clipping is used to display particles inside the torus since it allows more particles to be seen (see below and section 3.2.4).

No formal analysis is known to have been made of the clipping algorithm's efficiency, but an argument can be made, similar to that made for use of a z-buffer, that it is not very expensive for larger numbers of polygons. As the number of polygons in an image increases, the area of a polygon and thus its number of pixels decrease. With more polygons, more polygons will be partially clipped, but the number of tested pixels will not increase much.

The main purpose for implementing clipping was to enable the viewing of the advected particles in the torus' interior. Thus only front clipping was implemented. It is planned to implement viewing clipping against the sides of the view volume to enable zooming in on a region of a surface; otherwise polygons will be written outside the bounds of the frame buffer. Most polygons in an image do not intersect a clipping plane (unless there are very few polygons), and of those that do, few will intersect more than one clipping plane. Thus the expense in most cases will be an additional field in the polygon data structure stating which edge of the clipping volume is intersected by a polygon and a comparison during scan conversion of each pixel in a partially clipped polygon against the relevant coordinate of that edge.

No attempt was made to use anti-aliasing. Aliasing occurs because of the limited resolution of display devices. The main manifestation of aliasing is jagged lines, especially for nearly horizontal or nearly vertical lines or polygon edges. The scan-line z-buffer and the clipping algorithms used do not provide for anti-aliasing. The two main reasons for not attempting anti-aliasing are (1) the goal is not the achievement of realism; and (2) the algorithms are complex and expensive.

5.1.2. Fluid Flow Visualization

Advection was chosen as the method of flow visualization because it is simple to implement, effective in conveying features of the flow, and well-suited for animated display. In the 3D case, neither converting the data to scalar form (such as pressure contours), nor displaying vectors of the flow field directly, conveys as much global information as advection. Only a few surfaces can be used to display scalar data, and only one surface made up of vectors in a 3D flow field can be usefully displayed at one time.

Advection requires calculating the new position of each particle at each time step. The Runge-Kutta method for solving a differential equation numerically was chosen for finding the new position. Several other methods were tried and did not provide sufficient accuracy. Interpolation in time of the velocity components of the flow fields was also chosen to improve accuracy.

Two issues will have to be addressed before actual 3D flow fields can be used in advecting particles. The first issue involves the size of 3D flow fields; a single time step of the flow fields obtained from modelling flow inside the torus is several megabytes in size. To interpolate in time, two time steps must be accessible at the same time. A way must be found to manage the large amount of memory required in addition to the already large amount required for the frame and z-buffers, and the particle data. This problem has not been encountered so far, because the test case of section 1.2.2 constructs a 3D flow field from a much smaller 2D flow field, letting the y component serve for the z component as well.

Another issue that will have to be addressed is the possibility of particles overshooting the boundary of the torus. This problem does not occur in the test case because particles are advected using the 2D flow field for the backward facing step (particles do not overshoot -- see section 1.2.1), and their positions mapped to torus space. It may be necessary to use a smaller time step when using the real 3D data.

Many issues involved with the visual aspect of advecting particles have been addressed. Showing the track a particle makes as it is advected was found to be counter-productive except for small numbers of particles. Instead, larger numbers of particles are "injected" at various points to simulate the motion of dye tracers dropped into a fluid. The use of NTSC encoding for particle colors has to be taken into account, since particles produce high spatial frequencies, which the NTSC encoding tends to filter out. The positions at which particles are injected, the particle size, the number of particles injected, the frequency of injection, and the spacing of particles at injection have to be adjusted carefully to maximize the amount of information gained, especially if a portion of the region of interest is enlarged for viewing. Zooming reduces the ambiguity in viewing complex regions of the flow, both by enabling greater detail, and by eliminating inaccuracies in particle color due to the compression scheme used. On the other hand, it also reduces the amount of contextual information which is presented.

In the 3D case, the depth of a particle is impossible to see on a 2D display unless depth cueing is used. Of the three types of depth cueing tried, rotation, intensity variation, and size variation, rotation is the most effective. The angular change per frame during rotation has to be chosen carefully. If it is too great, it is hard to tell whether the motion of particles through the torus is due to the global rotation or fluid advection. If it is too slow, the depth cueing is lost. The rate of change of rotation must also be adjusted to avoid distracting jerkiness of movement.

Intensity cueing is the next most effective depth cueing. Two types are implemented, one in which the intensity component of a particle's color is based on the z coordinate, and one in which it is based on the particle's distance from the front of the object. The latter is used for displaying particles inside the torus, since the shape of the torus makes intensity cueing based on z coordinate alone redundant as the torus is rotated (all the particles in the forward "arm" become bright; see section 1.2.4).

Size cueing is the least effective, and has been discarded as a form of depth cueing. It introduces ambiguity as to whether clumps of particles are coming together or whether particles are moving towards the front of the object. It also effectively decreases the resolution for the larger particle sizes.

If particles are moving in the interior of a hollow object, as in the case of fluid flow inside the torus, the front of the object must be clipped. Clipping with respect to the front of the view volume ("viewing" clipping) results in part of the torus obscuring particles as the torus is rotated and rotates into and out of the clipping plane. Clipping with respect to a plane fixed with respect to the object ("object-oriented" clipping) ensures that the same part of the torus is always clipped regardless of rotation. The problem is not completely solved, because the walls of the torus obscure particles if the torus is rotated too far.

Future work addressing this problem involves implementing an algorithm to simulate transparency. This effect may be useful in order to see particles through the walls of the torus. Transparency may also be useful in depicting large numbers of advected particles. Large numbers (millions) of dark transparent particles are more informative than more moderate numbers (thousands) of brightly colored ones [45], though the use of zooming increases the effectiveness of the latter. Two difficulties with transparency are that intensity cueing will be more difficult to use, and the distortions caused in color by NTSC encoding might overcome the subtlety of transparency, though this remains to be seen.

The z-buffer algorithm cannot be used to achieve the simulation of transparency. If a polygon is transparent, its color must be blended with that of the polygon(s) behind it. The z-buffer itself does not save enough information in order to do this. It is planned to implement an algorithm that uses "separate transparency, intensity, and weighting factor buffers" [37], in addition to the z-buffer, to achieve transparency effects [29].

5.2. Usability

The distributed graphics system is designed to be easy to use. A high-level interface hides the details of RPC's, the socket level, and the video workstation from the user. Calls made to control the display on the workstation are similar to the GKS 2D standard. In the 2D case, the graphics primitives in both the software frame buffer and the metafile approach are the same; the GKS level transparently chooses the appropriate lower-level calls based on the approach chosen. The 3D viewing package adapted from Wishinsky [48] uses SIGGRAPH Core-style 3D calls [13].

The details of clipping, shading, and intensity cueing are hidden from the user. Each requires only making a call to set an attribute such as "object-oriented" or "viewing" (clipping), constant or Gouraud (shading), or to select one of the two types of depth cueing. In object-oriented clipping, for example, the z range of the object in world coordinates and the equation of the rotated clipping plane must be found. These items are taken care of automatically by the package when object-oriented clipping is specified. In general, the user-level calls are kept to a minimum and kept as simple as possible.

Color is specified in the easy-to-use HSV format. Rotation and light source orientation are not required in the non-intuitive viewing vector format [48]. Both are specified by a rotation in x, y, and z. If a relative light source is chosen, its new orientation is calculated automatically every time the eye point changes.

The distributed graphics system is modular by design. The advection, 3D viewing, rendering (scan conversion, hidden surface removal, etc.), compression, and RPC levels are, for the most part, self contained modules. (The modules are listed in Fig. 2.1 and are further described in Appendix B.) In one experiment, advection, 3D viewing, and rendering have been run on one machine, and the resulting information in the software frame buffer piped to another machine to be compressed and sent to the video workstation. Potentially all the modules mentioned above could be run as pipelined processes on different machines, with only minor changes. Since the modules are connected by a procedure call interface, this would provide a useful coarse-grained parallelism.

More sophisticated 3D rendering algorithms could replace the current algorithms in the rendering module as long as they write their results to the software frame buffer in the form expected by the compression module and RPC transport interface. Polygons and advected particles could be displayed on different devices than the video workstation by adding new device drivers to the GKS level situated between 3D viewing, and rendering. The GKS calls themselves are device and approach independent.

The compression and RPC transport interface is written so that it can be used by itself, without any of the above modules. In a separate project, a driver for the software frame buffer was added below a PostScript interpreter, allowing PostScript files to be displayed on the video workstation. The compression and RPC level only expects a raster image in a specified form, and does not care how that image is arrived at.

The distributed graphics system is designed to be usable for many scientific applications, and not just fluid flow visualization. It has been used in work unrelated to this thesis to make a movie of two groups of vortices interacting [Fig. 5.1] [Appendix C, Fig. C5.1], and to display 3D, scalar surfaces [Fig. 5.2] [Appendix C, Fig. C5.2]. In general, it is well suited for the dynamic display of particle and 3D polygon data.

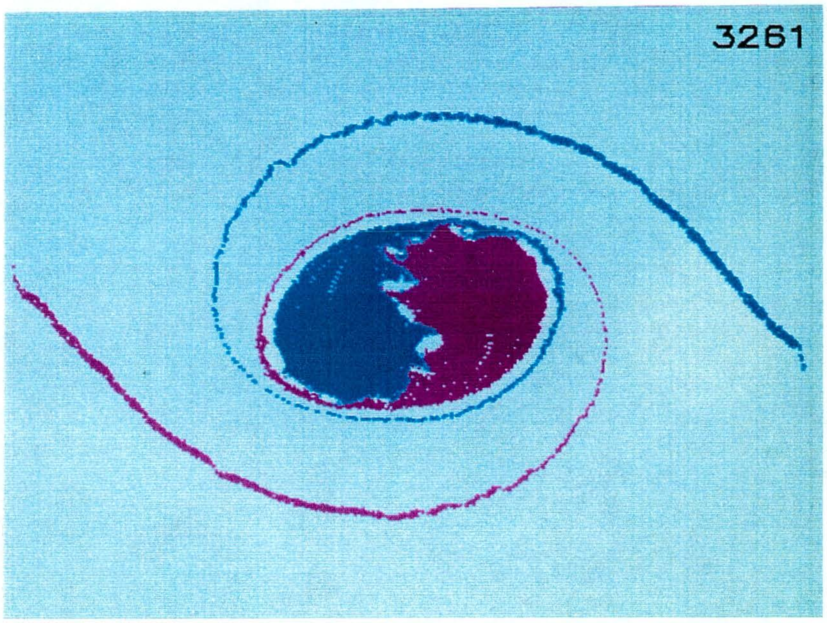


Fig. 5.1. Two vortices interacting. Simulation described in Baden [1].

BBC 883-2581

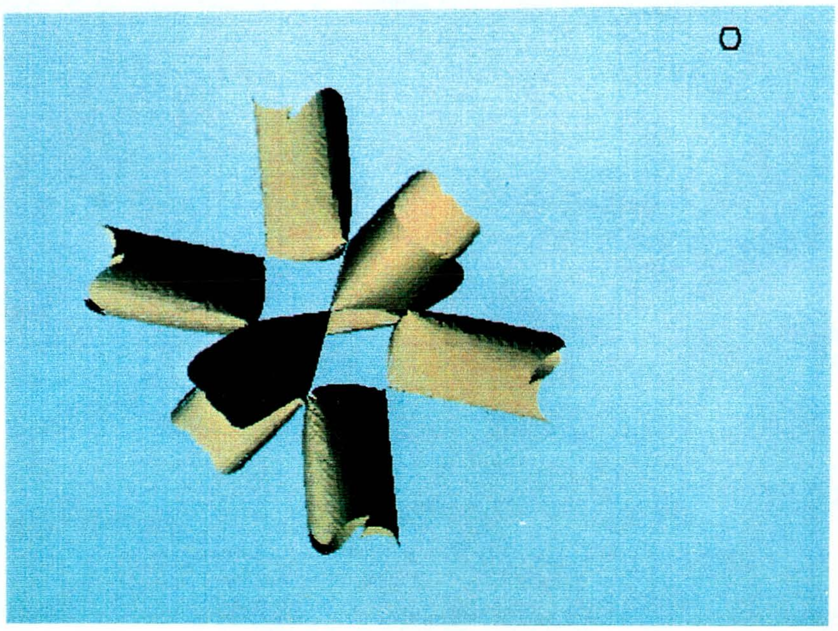


Fig. 5.2. Kummer surface.

BBC 883-2583

The distributed graphics system is proving a valuable aid for the interpretation of data produced at remote supercomputer centers. Many types of scientific modelling generate huge amounts of floating-point data, which are incomprehensible unless translated into visual form [24]. Sending uncompressed raster images across a wide-area network is impractical. The distributed graphics system, in conjunction with the video workstation, provides a way to automatically display and record images generated at remote supercomputer sites, and takes into account the low bandwidth available for cross-country networking using a fairly sophisticated, multistage compression.

As a feasibility study, the 2D portion has been ported to a remote Cray 2 in Minneapolis, Minnesota, where frames of the backward facing step movie have been generated, compressed, and then sent over the ARPA Net (a wide-area network) to be displayed and recorded on the video workstation. Typically a raster image can be compressed to a much smaller size than the floating point data which was used to produce the graphics primitives that went into the raster [18]. Transporting the gigabytes of raw floating point data in a sequence of 3D flow fields across a network is completely unfeasible. Sending the orders-of-magnitude smaller, compressed raster image resulting from advection and rendering of the boundary makes possible the achievement of an original goal of this thesis, the visualization of information in 3D flow fields.

REFERENCES

1. Baden, Scott. Very Large Vortex Calculations in Two Dimensions. Proceedings from the UCLA Workshop on Vortex Methods, Los Angeles, CA, 20-22 May. Lecture Notes in Mathematics. Springer-Verlag, New York. 1988.

2. Bell, R. J. T. An Elementary Treatise on Coordinate Geometry of Three Dimensions. 3rd ed. Macmillan & Co., Ltd., London. 1960.

3. Blinn, J. F. The Mechanical Universe: An Integrated View of a Large Scale Animation Project. Notes accompanying Course #6, presented at SIGGRAPH 1987, 27-31 July, Anaheim, CA. 1987.

4. Buning, P. G., and J. L. Steger. Graphics and Flow Visualization in Computational Fluid Dynamics. 7th AIAA Computational Fluid Dynamics Conference (1985), 162-167.

5. Burden, R. L., and J. D. Faires. Numerical Analysis. 3rd ed. PWS Publishers, Boston. 1985.

6. Campbell, G., T. A. DeFanti, J. Frederiksen, S. A. Joyce, L. A. Leske, J. A. Lindberg, and D. J. Sandin. Two Bit/Pixel Full Color Encoding. SIGGRAPH 1986 Proc. 20, 4 (Aug. 1986), 215-223.

7. Christiansen, H., and M. Stephenson. Graphics Utah Style - 80. A Workshop on Interactive Computer Graphics with Emphasis on the MOVIE system. Workshop presented July 28- August 1, 1980,.Salt Lake City, UT.

8. Comer, D. Operating System Design. Volume II, Internetworking with XINU. Prentice Hall, Inc., Englewood Cliffs, NJ. 1987.

9. Cowan, W. B., and C. Ware. Colour Perception Tutorial Notes: SIGGRAPH '85. Notes accompanying course presented at SIGGRAPH 1985, 22-26 July, San Francisco, CA. 1985.

10. DeFanti, T., and D. Sandin. The Usable Intersection of PC Graphics and NTSC Video Recording. IEEE Computer Graphics and Applications 7, 10 (Oct. 1987), 50-58.

11 Enderle, G., K. Kansy, and G. Pfaff. Computer Graphics Programming: GKS - The Graphics Standard. Springer-Verlag, Berlin. 1984.

12. External Data Representation Protocol Specification. Sun 3.0 Documents, Revision G of 17 February 1986. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043.

13. Foley, J. D., and A. van Dam. Fundamentals of Interactive Computer Graphics. Addison-Wesley Publishing Company, Reading, MA. 1982.

14. Fuchs, H., Z. M. Kedem, and S. P. Uselton. Optimal Surface Reconstruction from Planar Contours. Communications of the ACM 20, 10 (Oct. 1977), 693-702.

15. Grotjahn, R., and R. M. Chervin. Animated Graphics in Meterological Research and Presentations, Bull. Am. Meteorol. Society (USA) 65, 11 (Nov. 1984), 1201-1208.

16. Heckbert, P. Color Image Quantization for Frame Buffer Display. SIGGRAPH 1982 Proc. 16, 3 (July 1982), 297-307.

17. Herman, G. T. and H. K. Liu. Three-Dimensional Display of Human Organs from Computed Tomograms. Computer Graphics and Image Processing 9, 1 (Jan. 1979), 1-21.

18. Huang, J. Numeric Data Compression for Graphics. LBL-25037, University of California, Lawrence Berkeley Laboratory, Berkeley, CA. 1988.

19. Irani, N.B. Networked Graphics Workstation for Computational Fluid Dynamics. Proc. Trends and Applications 1985: Utilizing Computer Graphics (1985), 63-71.

20. Johnston, W. E., D. E. Hall, F. Renema, and D. Robertson. Low Cost Scientific Video Movie Making. Computer Physics Communications 45 (1987), 479-484. North Holland, Amsterdam.

21. Johnston, W. E., D. E. Hall, J. Huang, M. Rible and D. W. Robertson. Distributed Scientific Video Movie Making. LBL-24996, University of California, Lawrence Berkeley Laboratory, Berkeley, CA. 1988.

22. Kroos, K. A. Computer graphics techniques for three-dimensional flow visualization. Frontiers in Computer Graphics: Proceedings of Computer Graphics Tokyo '84, edited by T. L. Kunii. Springer-Verlag, New York. 1985.

23. Lynch, T. J. Data Compression: Techniques and Applications. Wadsworth, Inc., London. 1985.

24. McCormick, B. H., T. A. DeFanti, and M. D. Brown, eds. Visualization in Scientific Computing. Special Issue on Visualization in Scientific Computing. Computer Graphics 21, 6 (Nov. 1987).

25. McQueen, D. M. Telephone conversation with author, May 1987.

26. McQueen, D. M., and C. S. Peskin. Three-Dimensional Computational Method for Blood Flow in the Heart: (II) Contractile Fibers. Submitted to Journal of Computational Physics.

27. Microsoft C Compiler for the MS-DOS Operating System. Run-Time Library Reference. Version 5.0. 1987. Microsoft Corporation, 16011 NE 36th Way, Box 97017, Redmond, WA.

28 Moler, C. Single-User Supercomputers or How I Got Rid of the BLAS. Proc. Thirty-Third Computer Society International Conference (1988), 448-451.

29. Myers, A. J. An Efficient Visible Surface Program. Rep. to NSF, Div. of Math. and Comp. Sci., Computer Graphics Res. Group, Ohio State University, July 1975. Cited in Rogers [1985].

30. Network Software for IBM Personal Computers Running DOS. Reference Manual. Revision A, May 1986. Excelan, Inc., 2180 Fortune Drive, San Jose, CA.

31. Operating Instructions. Panasonic Optical Disc Recorder TQ-2026F, Optical Disc Player TQ-2027F. Panasonic Industrial Company, Division of Matsushita Electric Corporation of America, One Panasonic Way, Secaucus, NJ 07094.

32. Peskin, C. S. and D. M. McQueen. Three-Dimensional Computational Method for Blood Flow in the Heart: (I) Immersed Elastic Fibers in a Viscous Incompressible Fluid. Submitted to Journal of Computational Physics.

33. Peyret, R., and T. D. Taylor. Computational Methods for Fluid Flow. Springer-Verlag, New York. 1983.

34. Rektorys, K., ed. Survey of Applicable Mathematics. The M.I.T. Press, Cambridge, MA. 1969.

35. Remote Procedure Call Programming Guide. Sun 3.0 Documents, Revision G of 17 February 1986. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043.

36. Robertson, D. W., W. E. Johnston, D. E. Hall, and M. Rosenblum. Video Movie Making Using Remote Procedure Calls and UNIX IPC, LBL-22767, University of California, Lawrence Berkeley Laboratory, Berkeley, CA. 1986.

37. Rogers, D. F. Procedural Elements for Computer Graphics. McGraw-Hill Book Company, New York. 1985.

38. Rosenblum, M. The Performance of Sun's Remote Procedure Call. Term paper, CS266, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, California 94720.

39. Sechrest, S. An Introductory 4.3 BSD Interprocess Communication Tutorial. 4.3 Berkeley Software Distribution, Virtual VAX-11 Version. University of California, Berkeley, CA. April, 1986.

40. Sethian, J. A. Identifying and Tracking Coherent Structures in Turbulent Flow. To be submitted to Journal of Fluid Mechanics.

41 Sethian, J. A., and A. F. Ghoniem. A Validation Study of Vortex Methods. J. Computational Physics 74, 2 (Feb. 1988), 283-317.

42. Smith, A. R. Color Gamut Transformation Pairs. SIGGRAPH 1978 Proc., 12 (Aug. 1978), 12-19. Cited in Rogers [1985].

43 Sporer, M., F. H. Moss, and C. J. Mathias. An Introduction to the Architecture of the Stellar Graphics Supercomputer. Proc. Thirty-Third Computer Society International Conference (1988), 464-467.

44. Texier, N., W. E. Johnston and D. W. Robertson. Encoding Synthetic Animated Pictures. LBL-24236, University of California, Lawrence Berkeley Laboratory, Berkeley, CA. 1987.

45. Upson, C.. The Visual Simulation of Amorphous Phenomena. The Visual Computer 2, 5 (Sept. 1986), 321-326.

46. Video Animation Software for the DQ-400V System. DIAQUEST Documentation DQ-400. 1985. DIAQUEST, Inc., 1442 San Pablo Avenue, Berkeley, CA 94702.

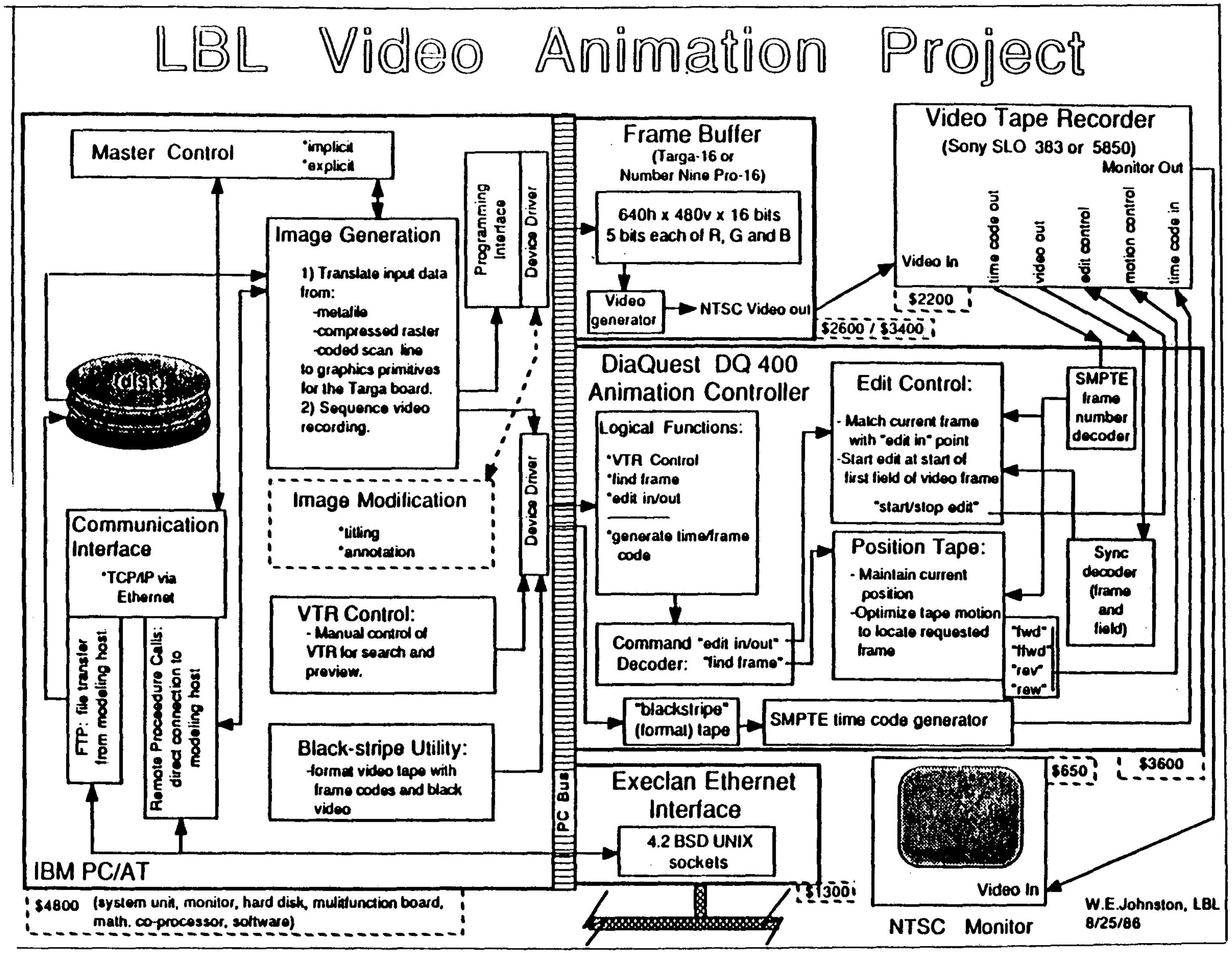
47. Watson V., P. Buning, D. Choi, G. Bancroft, F. Merritt, and S. Rogers. Use of Computer Graphics for Visualization of Flow Fields. Paper read at the AIAA Aerospace Engineering Conference and Show, 17-19 Feb., Los Angeles, CA. 1987.

48. Wishinsky, B. J. A Simplifed Interface for SIGGRAPH Core Viewing. LBL-25038, University of California, Lawrence Berkeley Laboratory, Berkeley, CA. 1987.

Appendix A

Video Workstation Configuration

The following page is a diagram of the video workstation configuration from Johnston, et al. [21]. It includes the PC controller, the frame buffer, the monitor, and the Ethernet board. This particular configuration is for the version using videotape, and includes the video controller and the video tape recorder.


LBL Video Animation Project
Master Control
•implicit
•explicit
Image Generation
1) Translate input data from:
-metafile
-compressed raster
-coded scan line
to graphics primitives for the Targa board.
2) Sequence video recording.
Programming Interface
Device Driver
(disk)
Image Modification
•titling
•annotation
Device Driver
Communication Interface
•TCP/IP via Ethernet
FTP: file transfer from modeling host
Remote Procedure Calls: direct connection to modeling host
VTR Control:
- Manual control of VTR for search and preview.
Black-stripe Utility:
-format video tape with frame codes and black video
IBM PC/AT
PC Bus
$4800 (system unit, monitor, hard disk, mulitfunction board, math. co-processor, software)
Frame Buffer
(Targa-16 or Number Nine Pro-16)
640h x 480v x 16 bits
5 bits each of R, G and B
Video generator
NTSC Video out
$2600 / $3400
Video Tape Recorder
(Sony SLO 383 or 5850)
Monitor Out
Video In
time code out
video out
edit control
motion control
time code in
$2200
DiaQuest DQ 400 Animation Controller
Logical Functions:
•VTR Control
•find frame
•edit in/out
•generate time/frame code
Command Decoder: "edit in/out" "find frame"
Edit Control:
- Match current frame with "edit in" point
-Start edit at start of first field of video frame
"start/stop edit"
Position Tape:
- Maintain current position
-Optimize tape motion to locate requested frame
"fwd"
"ffwd"
"rev"
"rew"
SMPTE frame number decoder
Sync decoder (frame and field)
"blackstripe" (format) tape
SMPTE time code generator
$3600
Execlan Ethernet Interface
4.2 BSD UNIX sockets
$1300
$650
Video In
NTSC Monitor
W.E.Johnston, LBL
8/25/86

Appendix B

Documentation of System

The following pages documenting the distributed movie-making system are organized as follows:

Section L: Overview

1. summary of system
2. diagram of modules
3. description of applications using system

Section 3: Program modules' descriptions

Note on this LBL report version of the thesis:

Much of the code written for this thesis was originally included as part of Appendix B. It is now available in an expanded form as public domain software, called Scry, and is thus not included here. The original thesis also included call graphs resulting from profiling, which are not provided here because routine names and calling sequences have been changed since that time.

Scry is provided as a professional academic contribution for joint exchange. Thus it is experimental, is provided "as is", with no warranties of any kind whatsoever, no support, promise of updates, or printed documentation. Scry is available by anonymous ftp (login: "anonymous", password: "guest") from csam.lbl.gov

(128.3.254.6) in *pub/scry.tar.Z* (a compressed tar file, so don't forget to set binary mode in ftp). Be aware that the compressed file is just under 1 megabyte. Once on your machine, run uncompress on *scry.tar.Z*, and extract the files using

```
tar xvf scry.tar scry
```

NAME

distributed movie-making system

DESCRIPTION

The distributed movie-making system consists of a video workstation that acts as a movie server for a client program running on a variety of systems, including Suns, Vaxen, and Crays. The components of the video workstation are a PC compatible equipped with an Ethernet board, an AT&T TARGA 16-bit frame buffer, and a recording device. At the present there are two video workstations located in Building 50B, Rooms 2265 and 2267, at the Lawrence Berkeley Laboratory. Workstation #2 is equipped with a videotape recorder and animation controller. Workstation #1 has a video optical disk recorder and a 68020 coprocessor to aid in decompression.

The client program converts the results of time-dependent scientific simulations into a graphics representation such as points, lines, and polygons. These graphics primitives are either displayed directly via remote procedure calls or rendered into a software frame buffer located in main memory on the front-end machine. This software frame buffer is then compressed and sent using remote procedure calls to the video workstation, where it is decompressed and displayed. The latter approach is used almost exclusively, especially when the front-end is communicating over a wide-area network.

The client program is logically separated into several modules which are described in section 3 of this appendix. The organization of the modules is illustrated in Fig. B.1 (following page). When the data to be displayed is two dimensional, there is a GKS module, a 2D scan conversion module, and two modules at the RPC level that implement the two different means of displaying the graphics primitives. When the data to be displayed is three dimensional, 3D viewing, GKS, and 3D scan conversion modules are used, as well as the RPC module implementing the software frame buffer approach.

Three sample applications are described in **dye2d**(L), **woggle**(L),and **dye3d**(L). Assuming symbolic links to the various libraries in the application program's directory, one links in the 2D libraries with:

```
cc -o dye2d dye2d.o dye2dsupt.o gks2d.a client.a metaclient.a -lrpcsvc -lm
```

for example, and 3D libraries with:

```
cc -o woggle woggle.o bj3d.a gks3d.a scan3d.a client.a metaclient.a -lrpcsvc -F77 -I77 -U77 -lm
```

for example.

The 3D polygon input data format is described in **triang**(L).

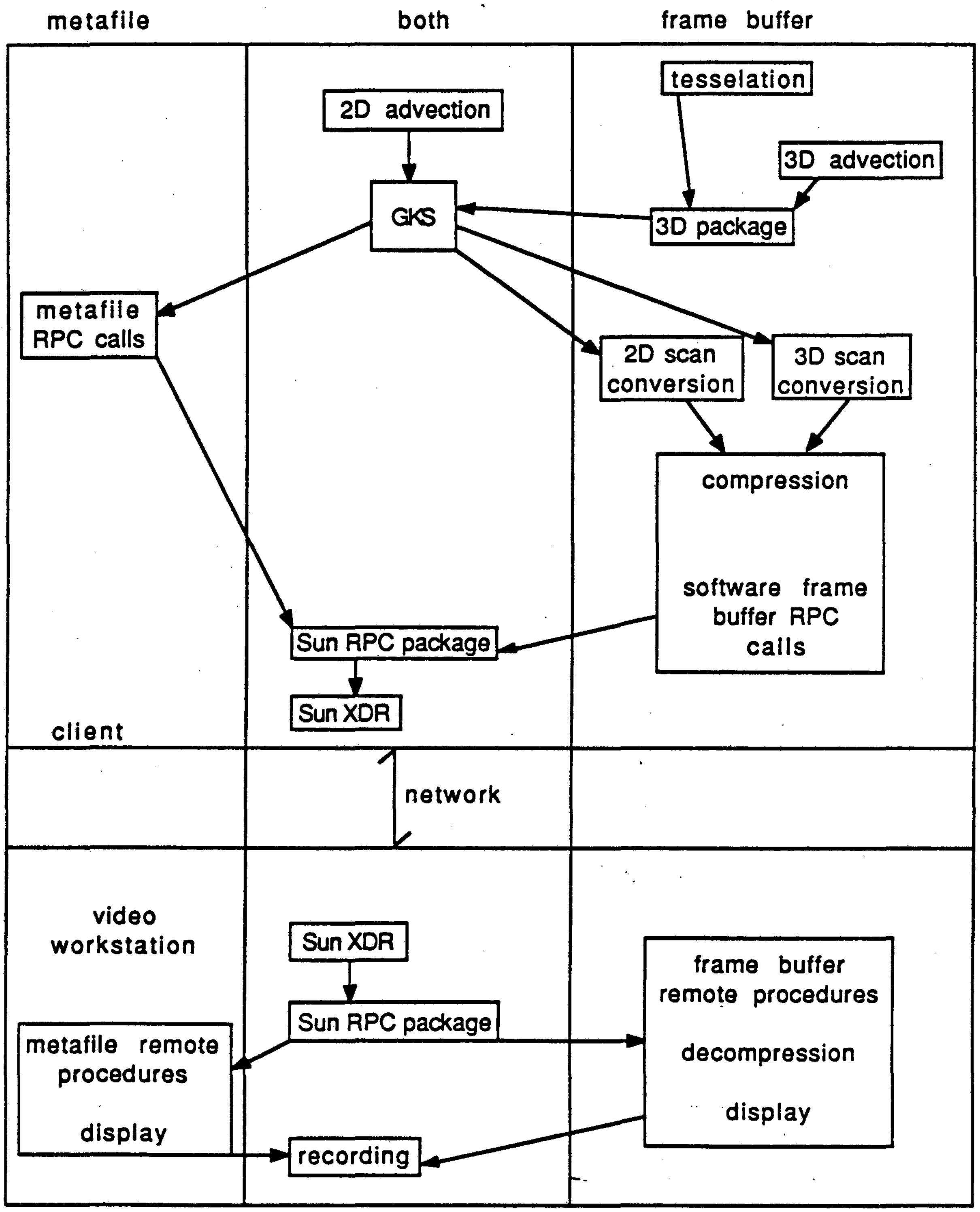
SEE ALSO

triang(L),**woggle**(L),**dye2d**(L),**dye3d**(L),
bj3d.a(3),**gks**(3),**scan2d.a**(3),**scan3d.a**(3),
client.a(3),**metaclient.a**(3),**server**(3)

BUGS

metaclient.a has to be linked in with 3D code even though it is never used by it. Such linking makes it easier for 2D and 3D GKS libraries to share code. It would be better to have two different versions of GKS for 2D and 3D.

Fig. B.1. Module Diagram



NAME

dye2d – view flow over a backward-facing step

SYNOPSIS

dye2d [**–1** **–2** **–p** *protocol* **–r** *frame*

–c *compression* **–a** *x1 y1 x2 y2*

–s **–w** *approach* **–h**]

DESCRIPTION

dye2d allows one to periodically inject dye particles at 2 arbitrary points within the flow over the 2D backward facing step. The input data is a sequence of flow fields generated by J. Sethian of U.C. Berkeley and A. Ghoniem of M.I.T. It is an example of an application program using the 2D portion of the movie-making system. **dye2d** creates the files *endframe2, begin,* and *position* in the directory the program is running in. Do not delete these if you want to use the **–s** option (see below).

OPTIONS

–1 or –2
Video workstation #1 or #2. Default is #1.

–p *protocol*
Selects Internet protocol to use in sending data to the video workstation. *protocol* is either **udp** or **tcp**. Default is **udp**.

–r *frame*
Record starting at frame number *frame*. Be sure if using videodisk that it is possible to record at that location. If videotape is used, check to see if something is already recorded at that location. If this option is not specified, the video workstation goes into preview mode.

–c *compression*
Sets compression type. *compression* can be **none, btc**, or **color** (for color map). Default is color map.

–a *x1 y1 x2 y2*
2 injection points. $0.0 < x < 9.9$, $0.0 < y < 1.0$. For best results, $0.2 < x$, $0.05 < y < 0.95$. Default is (0.3, 0.34), (0.9, 0.7).

–s Start with checkpoint file *endframe2*, which should be generated on a previous run (one run is 100 frames or to the last frame possible). Default is to start from the beginning. The maximum number of frames is 764 (frame number is in upper right hand corner of display). Do not attempt to use **–s** for frames past this.

–w *approach*
approach can be **meta** (for metafile) or **frameb** (for software frame buffer). Default is **frameb**.

–h Help message.

SEE ALSO

movie-making(L)

BUGS

Doesn't die gracefully if injection point out of bounds.

NAME

triang – triangulates a torus data file

SYNOPSIS

triang *infile outfile*

DESCRIPTION

triang is an example of generating polygon data in the form required by the 3D viewing package. The torus is generated from data provided by Charles Peskin and David McQueen of the Courant Institute of New York University. The data is provided as a series of strands or fibers. **triang** laces together the points on one strand and the next to form a series of polygons (in this case, triangles). The graphics algorithms used by the 3D viewing and 3D scan conversion modules expect surfaces to be made up of polygons. The process of generating polygons is called triangulation.

The specific polygon input format expected by the 3D viewing package is very similar to the input format expected by the Mosaic package of movie.byu. The following file will be rendered as two filled triangles.

```
4 6
 0.80000e+02 0.00000e+00 0.13000e+03 0.69282e+02 0.40000e+02 0.14000e+03
 0.40000e+02 0.69282e+02 0.13000e+03 0.00000e+00 0.00000e+00 0.14000e+03
    4   1  -2   4   2  -3
```

The first line contains the number of vertices in the vertex list and the number of vertices in the edge list. The vertex list is contained in the next two lines, i.e. x1, y1, z1, x2, y2, z2, etc. The third line is the edge list, consisting of indices into the vertex list. A negative index signals the end of a polygon. Vertices are numbered starting with 1 rather than 0. The 3D scan conversion module connects the last vertex to the first vertex – don't do that in this file. It is important that the vertices of all polygons be listed in a consistent counter-clockwise order. Otherwise the polygon normals will be computed incorrectly.

OPTIONS

infile Strand data file.

outfile Mosaic-style polygon data file.

SEE ALSO

movie-making(L),dye3d(L)

NAME

woggle – makes movie of rotating object

SYNOPSIS

woggle [**–1 –2 –p** *protocol* **–r** *frame*

–c *compression* **–t** *total* **–d** *rotation-increment*

–b *rotx roty rotz* **–g –h**

–i *file*]

DESCRIPTION

woggle provides a smoothly changing view of an object described by a Mosaic-style polygon file, and optionally records the sequence of frames generated on videotape or videodisk.

OPTIONS

–1 or –2
Video workstation #1 or #2. Default is #1.

–p *protocol*
Selects Internet protocol to use in sending data to the video workstation. *protocol* is either **udp** or **tcp**. Default is **udp**.

–r *frame*
Record starting at frame number *frame*. Be sure if using videodisk that it is possible to record at that location. If videotape is used, check to see if something is already recorded at that location. If this option is not specified, the video workstation goes into preview mode.

–c *compression*
Sets compression type. *compression* can be **none, btc**, or **color** (for color map). Default is color map.

–t *total*
Total number of frames in sequence. Default is 2.

–d *rotation-increment*
Increment in x, y, and z rotation from frame to frame. Default is 1.

–b *rotx roty rotz*
Starting view of object specified by rotation in x, y, and z. Default is 0, 0, 0.

–g Use Gouraud shading. Default is constant shading.

–h Help message.

–i *file* Polygon input file. Default is a chambered torus.

SEE ALSO

movie-making(L), triang(L)

NAME

dye3d – view flow through a cut-away torus

SYNOPSIS

dye3d [**–1 –2 –p** *protocol* **–r** *frame*

–c *compression* **–a** *x1 y1 z1 x2 y2 z2*

–s –h]

DESCRIPTION

dye3d allows one to periodically inject dye particles at 2 arbitrary points within the "flow" within a torus. Injections stop after the 210th frame to avoid filling the torus with particles. Input data is a sequence of 2D flow fields for a backward-facing step generated by J. Sethian of U.C. Berkeley and A. Ghoniem of M.I.T that has been promoted to 3D by letting y velocities stand for z velocities as well. Particles are advected in step space and then transformed into torus space. This is not a useless exercise, since it tests the ability of the movie-making system to display particles in such a way that it is obvious they are moving in 3D space. Global rotation and depth intensity cueing are used to achieve the 3D effect.

dye3d creates the files *endframe*, *begin*, and *position* in the directory the program is running in. Do not delete these if you want to use the **–s** option (see below).

The torus is generated from data provided by Charles Peskin and David McQueen of the Courant Institute of New York University. Transformation of this data to polygonal form is accomplished with **triang**.

OPTIONS

–1 or –2
Video workstation #1 or #2. Default is #1.

–p *protocol*
Selects Internet protocol to use in sending data to the video workstation. *protocol* is either **udp** or **tcp**. Default is **udp**.

–r *frame*
Record starting at frame number *frame*. Be sure if using videodisk that it is possible to record at that location. If videotape is used, check to see if something is already recorded at that location. If this option is not specified, the video workstation goes into preview mode.

–c *compression*
Sets compression type. *compression* can be **none**, **btc**, or **color** (for color map). Default is color map.

–a *x1 y1 z1 x2 y2 z2*
2 injection points. $0.0 < x < 9.9$, $0.0 < y < 1.0$, $0.0 < z < 1.0$. For best results, $0.2 < x$, $0.05 < y < 0.95$, $0.05 < z < 0.95$, y not equal to z. Default is (0.3, 0.34, 0.7), (0.9, 0.7, 0.34).

–s Start with checkpoint file *endframe*, which should be generated on a previous run (one run is 100 frames or to the last frame possible). Default is to start from the beginning. The maximum number of frames is 764 (frame number is in upper right hand corner of display). Do not attempt to use **–s** for frames past this. In practice the display is too busy after frame 300.

–h Help message.

SEE ALSO

movie-making(L), triang(L)

BUGS

Doesn't die gracefully if injection point out of bounds.

NAME

bj3d.a – 3D viewing library

DESCRIPTION

Performs 3D viewing. This process includes setting up a view from a certain angle of an object made up of polygons, for example, as part of the projection to 2D, and also 3D clipping. Information necessary for scan conversion of 3D polygons, such as the polygon normal and the light source vector, is also calculated using this library. Calls GKS routines to display projected graphics primitives (points, lines, and polygons).

A large proportion of the library was modified from an experimental 3D viewing package, for displaying line drawings, that provides an easy means of changing the view of an object. The experimental package was coded in FORTRAN by BJ Wishinsky. User-level routines starting with "S" are C routines that make it easier to access the FORTRAN 3D routines from within C. For example, these C routines make sure that each argument is passed by reference to FORTRAN. Additions to the package were made both in C and FORTRAN. Routines written by BJ Wishinsky and those written or modified by David Robertson are identified in the code.

The first statement of the C application program must be f_init() and the last statement f_exit() to allow the use of FORTRAN routines from within C. (f_init and f_exit deal with FORTRAN I/O.)

SEE ALSO

movie-making(L), triang(L), woggle(L), gks(3)

NAME

GKS – device-independent graphics calls

DESCRIPTION

GKS (the Graphical Kernel Standard) provides a device-independent manner of writing graphics applications programs. It is described in GKS - the Graphics Standard, by G. Enderle, K. Kansy and G. Praff [11].

Routines in the gks2d.a and gks3d.a libraries are best described as GKS-like, rather than as even a minimal GKS implementation. GKS does not support 3D; 3D graphics primitives have an additional non-standard z argument for use by the z-buffer. The HSV rather than the RGB color model is used. Calls are routed to one of two device drivers in the 2D case. Instead of controlling two different workstations, however, the device drivers are for two different ways of generating graphics on the same workstation, i.e. the metafile and software frame buffer approaches. Only one approach can be "open" at a time. In the 3D case only the software frame buffer approach can be used.

User-level routines starting with "D" are callable from C. These perform no action other than to call the real GKS routines which start with "d" and have an underscore appended. All arguments are passed to the "d" routines by reference. The "d" user-level routines are designed to be callable from FORTRAN, which unfortunately makes them more difficult to call from C, hence the "D" routines.

Routines common to the 2D and 3D version include those controlling GKS and the workstation, controlling the use of color, converting from world to normalized device coordinates, and setting the workstation window and workstation viewport. Strictly 2D graphics primitives (points, lines, and text) are callable by the user. Strictly 3D calls are not user-level. They are only made from within the 3D viewing package (see **bj3d.a**). Besides the 3D graphics primitives (points, lines, and polygons), information relating to polygon shading, clipping, and the use of depth intensity cueing is routed to the software frame buffer device driver.

SEE ALSO

movie-making(L), bj3d.a(3), scan2d.a(3), scan3d.a(3), client.a(3), metaclient.a

BUGS

dtxt has no corresponding "D" routine and is not callable from FORTRAN. There is no 2D polygon primitive. Virtually no error checking is done.

NAME

scan2d.a – 2D scan conversion library

DESCRIPTION

GKS routes 2D calls into routines in this library when the software frame buffer approach is chosen. Clips points and lines to the intersection of the current Normalized Device Coordinates viewport and the workstation window. Scan converts remaining points and lines, as well as unclipped text, into the software frame buffer. The software frame buffer has the same dimensions as the hardware display frame buffer. The other component library of the driver for this approach, **client.a**, optionally compresses the software frame buffer, and transmits it to the video workstation for display.

SEE ALSO

movie-making(L), gks(3), client.a(3)

NAME

scan3d.a – 3D scan conversion library

DESCRIPTION

GKS routes 3D calls into routines in this library. Scan converts points, lines, 2D text, and polygons into software frame buffer. Polygons are rendered depending on the lighting and shading model, and optionally clipped during scan conversion. Hidden points, lines, and polygons are removed using a z-buffer. The software frame buffer has the same dimensions as the hardware display buffer. The other component library of the 3D driver, **client.a**, optionally compresses the software frame buffer and transmits it to the video workstation for display.

SEE ALSO

movie-making(L), gks(3), scan2d.a(3), client.a(3)

NAME

client.a – library for software frame buffer compression and transmittal

DESCRIPTION

Takes software frame buffer generated by routines in **scan3d.a** or **scan2d.a**, optionally compresses it, and transmits to the video workstation via Sun RPC's for decompression and display. If record mode is set, controls recording as well. Over a local-area network, Block Truncation Coding combined with a color map encoding is usually used, since it is the least time consuming option (the other two options are no compression and BTC by itself). Over a wide-area network a further compression step, using Lempel-Ziv encoding, is used as well. To use Lempel-Ziv include the -DWIDEAREA option in the compile line (which #**define**'s WIDEAREA).

This library can also be used in a stand-alone fashion. It has been used, for example, to display Postscript files on the video workstation. If the appropriate header file is included in the user's program and pixels written into the software frame bufferin TARGA format, this library does not care how the image was generated.

SEE ALSO

movie-making(L), woggle(L), scan2d.a(3), scan3d.a(3)

NAME

metaclient.a – library for metafile approach for transferring 2D graphics primitives

DESCRIPTION

GKS routes 2D calls into routines in this library when the metafile approach to displaying points, text, and lines is chosen. Clips points and lines to the intersection of the current Normalized Device Coordinates viewport and the workstation window. Transmits positions and colors of text, and remaining points and lines, to the video workstation via Sun RPC's, where display via scan conversion into the hardware frame buffer takes place. All points to be displayed are stored until the Duwk (update workstation) call is made, when they are transmitted in blocks of 100. If record mode is set, controls recording as well.

SEE ALSO

movie-making(L), dye2d(L), gks(3), client.a(3)

NAME

server – remote procedure call server running on video workstation

DESCRIPTION

Serves incoming remote procedure calls (RPC's) to display and optionally record graphics sent using either metafile or software frame buffer approach. Sun RPC package calls the remote "program" (actually a dispatch procedure) based on information in header of RPC call. Cases in the dispatch procedure correspond to remote procedures. The remote procedure number identifying a particular case is also located in the RPC call header, which is internal to the Sun RPC package. Routines starting with **xdr_** decode incoming information from network byte to PC byte order. See Robertson [34] for more information on how RPC's are implemented on the PC.

SEE ALSO

movie-making(L), client.a(3), metaclient.a(3)

Appendix C

Videotape and Color Print Availability

Color prints are necessary in the case of about half the photos, especially where particles are color coded by injection point. The videotape is also necessary: visualizing scientific data via video movies is a main thrust of this thesis.

If this copy of the thesis has black and white photographs, copies which have color photographs are available elsewhere. A copy of the thesis with color prints is located in the Computer Science department office, located in Thornton Hall, Room 969, San Francisco State University. A copy of the videotape is available from the Audio-Visual section of the San Francisco State J. Paul Leonard Library.

A copy of the thesis with color prints is also available as a Lawrence Berkeley Laboratory technical report, LBL-25274, available from the Building 50 library at Lawrence Berkeley Laboratory, 1 Cyclotron Road, Berkeley, CA 94720. The phone number for the Building 50 library is (415) 486-5621. A copy of the videotape is available from the same location as technical report LBL-25275 (video).

Following are the captions for each video clip on the videotape.

Captions for Video Figures

Figure C1.1. Note continuous color coding of magnitude and tracking of vortices [41].

Figure C1.2. Remake of Figure C1.1 [41]. 500 particles per injection point. Time step = 0.125 s.

Figure C1.3. 10000 particles per injection point.

Figure C1.4. Color inaccuracies due to NTSC encoding.

Figure C1.5. Zoom on outlined region.

Figure C1.6. Zoom on outlined region. Note passage of vortices.

Figure C1.7. Too little rotation to give proper depth cueing for particles.

Figure C1.8. Note concealment of particles by walls of torus and use of size cueing.

Figure C1.9. Final version. No size cueing.

Figure C2.1. Inaccuracies due to BTC and NTSC encoding of color.

Figure C2.2. Zoom on Figure C2.1. Note almost all inaccuracies removed.

Figure C3.1. First sequence: "viewing" clipping. Second sequence: "object-oriented" clipping.

Figure C3.2. Fixed light source at (0,0,0) rotation.

Figure C3.3. Relative light source at (-45,0,45) rotation.

Figure C3.4. Constant shaded torus.

Figure C3.5. Color aliasing (scalloping).

Figure C4.1. Motion discontinuity at middle of sequence.

Figure C4.2. First sequence: recorded on videotape. Second sequence: recorded on videodisk.

Figure C5.1. Two vortices interacting [1].

Figure C5.2. Level surfaces of equation for Kummer surface. Kummer surface proper is held frame in middle of sequence.