Computer Graphics Rendering Techniques
with an Emphasis on Performance Issues

by
Morten Zachrisen,

Division of Computer Systems and Telematics,
The Norwegian Institute of Technology
and
Knowledge Engineering and Image Processing Group,
ELAB-RUNIT
Trondheim 89 12 28

The author's address:
ELAB-RUNIT,
N-7034 Trondheim,
Norway

tel: 47-7-597081.

email address:
zach@dorma.er.sintef.no
Computer Graphics Rendering Techniques with an Emphasis on Performance Issues

Volume 1

Survey and Integration

NTNU
Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap
Gleshaugen
7491 Trondheim
THESIS ABSTRACT

This thesis gives an overview of three-dimensional computer graphics rendering techniques within the context of interactive performance on inexpensive workstations.

Two different approaches to rendering systems are discussed. One is an early attempt to build a rendering toolkit with object oriented system architecture. The second approach is based on algorithms and architecture of the object oriented approach, but emphasizes the integration of an efficient rendering system for inexpensive workstations.

Within the framework of these systems, several new algorithms have been developed. The algorithms all relate to different parts of the rendering pipeline.

New, efficient extent testing and polygon clipping algorithms make graphic navigation in complex models possible.

An efficient implementation of a scan line depth buffer hidden surface algorithm based entirely on fixed-point arithmetic brings interactive shaded imagery within the reach of cheap PCs and workstations.

New extensions to the depth buffer algorithm add the realism of shadow casting to workstation rendering.

Depth buffer algorithms for the rendering of constructive solid geometry (CSG) models allow interactive building of complex objects and real-time simulation of numerically controlled machining operations.
THESIS PREFACE

This thesis summarizes work carried out during the complete period 1984-89, but with most of the work being done in 1985/86. With the exception of the period 1984/85 which was spent as a visiting lecturer at the Computer Science Dept., University of Utah, the work with this thesis has been carried out in parallel with my normal work for RUNIT/SINTEF, and it has thus taken this long to complete. Such a long duration is a problem in field as fast moving as Computer Graphics. For instance, raster graphics was in its early childhood ten years ago. Most of the report writing has been done during 1988/89, and references to recent work of other researchers have thus been incorporated.

The work has been carried out with the sponsorship of many different development programs and projects in addition to the general support from my employer (ELAB-RUNIT):
NTNF dr.ing. scholarship 1983-85 and 1986 (three years).
NAVF travel grant for the U.S.A visit.
DARPA project support (DAAK1184K0017).
NTNF project support - "High-End Raster" (For ICAN a/s).
NTNF project support - "Parallell grafisk presentasjonsenhet".
SINTEF strategic project support - "Flerfaglig simulerings".

In a field where the majority of the literature is in English, I have found it necessary to use this language for writing.

This thesis has been processed by Microsoft Word on a Macintosh. All figures (unless specially noted) are original, and most have been included in-line with the text. Figures have been prepared mainly by MacDraw. Tables and graphs have been prepared using Microsoft Excel.

This thesis is not very formula-intensive. I did not have the need to reference formulas across pages. Formula indices have therefore been omitted.

I would have liked to present colour illustrations in a better way than just photographs taken off the colour screen. From the software point of view this is trivial since Moviebox can drive most raster screens and high-density raster
plotters. However, neither high-resolution colour plotters nor film plotters was available. Therefore, photographs had to be used.

Acknowledgements.

Many thanks to Arne Halaas, who has been the advisor on this thesis. Thanks also to Kristen Rekdal who was my advisor during the first year of the study.

Many thanks to Joe Gorman for his help with the foreign language and also for his comments on technical matters.

Thanks to my superiors at ELAB-RUNIT who have provided the necessary resources, and colleagues at RUNIT and SINTEF who have contributed geometry models and interesting rendering problems.

Thanks also to the members of the Alpha_1 CAGD group at the University of Utah for their cooperation during my stay, and in particular to Richard Riesenfeld who made the stay possible.

I am, however, most indebted to Yngvild, Magne, and Gøril who have patiently awaited the completion of this thesis.
# TABLE OF CONTENTS - VOLUME 1

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>THESIS ABSTRACT</td>
<td>I</td>
</tr>
<tr>
<td>THESIS PREFACE</td>
<td>III</td>
</tr>
<tr>
<td>TABLE OF CONTENTS - VOLUME 1</td>
<td>V</td>
</tr>
<tr>
<td>1. ABOUT THIS THESIS</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Overview</td>
<td>2</td>
</tr>
<tr>
<td>1.2. Thesis Structure</td>
<td>4</td>
</tr>
<tr>
<td>2. WHAT HAS BEEN ACCOMPLISHED?</td>
<td>5</td>
</tr>
<tr>
<td>2.1. GAS</td>
<td>5</td>
</tr>
<tr>
<td>2.2. Moviebox</td>
<td>5</td>
</tr>
<tr>
<td>2.3. Rendering Algorithms for a RasterOp Engine</td>
<td>6</td>
</tr>
<tr>
<td>2.4. A Fast Polygon Clipper</td>
<td>7</td>
</tr>
<tr>
<td>2.5. Adding Structure to Bit-Map Displays</td>
<td>7</td>
</tr>
<tr>
<td>3. A SURVEY OF THE RENDERING PROCESS WITH COMMENTS ON IMPLEMENTATION ISSUES</td>
<td>9</td>
</tr>
<tr>
<td>3.1. Modelling</td>
<td>9</td>
</tr>
<tr>
<td>3.1.1. Polygon Databases</td>
<td>10</td>
</tr>
<tr>
<td>3.1.2. Spline Surfaces</td>
<td>12</td>
</tr>
<tr>
<td>3.1.3. Solid Models</td>
<td>13</td>
</tr>
<tr>
<td>3.1.4. Models of Nature</td>
<td>13</td>
</tr>
<tr>
<td>3.2. Structure Traversal</td>
<td>13</td>
</tr>
<tr>
<td>3.3. Viewing Transformations</td>
<td>15</td>
</tr>
<tr>
<td>3.4. Clipping</td>
<td>22</td>
</tr>
<tr>
<td>3.5. Hidden Surface Removal</td>
<td>22</td>
</tr>
<tr>
<td>3.5.1. Geometry Engines and the Z-Buffer Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>3.5.2. Polygon Output Algorithms</td>
<td>23</td>
</tr>
<tr>
<td>3.5.2.1. List Priority Algorithm</td>
<td>24</td>
</tr>
<tr>
<td>3.5.2.2. &quot;Cookie-Cutter&quot; Subdivision Algorithm</td>
<td>24</td>
</tr>
<tr>
<td>3.5.2.3. Binary Separating Planes</td>
<td>25</td>
</tr>
<tr>
<td>3.5.3. Scan Line Output Algorithms</td>
<td>26</td>
</tr>
<tr>
<td>3.5.3.1. Scan Line Z-Buffer Algorithm</td>
<td>26</td>
</tr>
<tr>
<td>3.5.3.2. Spanning Scan Line Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>3.5.3.3. A-Buffer</td>
<td>28</td>
</tr>
<tr>
<td>3.5.3.4. Ray Tracing</td>
<td>28</td>
</tr>
<tr>
<td>3.6. Aspects of a Z-Buffer Renderer</td>
<td>31</td>
</tr>
<tr>
<td>3.6.1. Scan Line or Scan Buffer Rendering</td>
<td>31</td>
</tr>
<tr>
<td>3.6.1.1. Memory Usage</td>
<td>31</td>
</tr>
</tbody>
</table>
### VI Volume 1 - Survey and Integration

- 3.6.1.2. Rendering Quality .............................................. 32
- 3.6.1.3. Delay ......................................................... 33
- 3.6.2. Depth Resolution ............................................... 33
- 3.6.3. Offset Z-Coordinate Sampling .............................. 36
- 3.6.4. Fixed-Point Arithmetic and Rendering ................. 37
- 3.7. Illumination Models and Shading ............................ 38
  - 3.7.1. Fast Shading Computations ............................. 39
  - 3.7.2. Gouraud Shading ........................................... 40
  - 3.7.3. Phong Shading ............................................. 40
  - 3.7.4. Fast Computation of Unit Normal ..................... 42
  - 3.7.5. "Cheap Phong" Shading .................................. 44
  - 3.7.6. Fast Phong Shading Computations ..................... 46
  - 3.7.7. Torrance and Sparrow Light Model .................... 47
  - 3.7.8. Global Illumination Models ............................ 48
  - 3.8. Anti-Aliasing ................................................ 49
  - 3.8.1. Supersampling ............................................. 51
  - 3.8.2. Adaptive Supersampling ................................ 51
  - 3.8.3. Anti-Aliasing by Area Integration .................... 52
  - 3.8.4. Separable Two-Dimensional Filters .................... 52
  - 3.8.5. Anti-Aliasing in Non-RGB Colour Spaces .............. 53
- 3.9. Texture Mapping, Transparency, Reflectance .............. 55
  - 3.9.1. Texture Model ............................................. 55
  - 3.9.2. Bump Mapping ............................................. 56
  - 3.9.3. Environment Mapping .................................... 56
  - 3.9.4. Contour Mapping ......................................... 57
  - 3.9.5. Fast Texture Mapping .................................... 57
  - 3.9.6. Transparency and Reflectance .......................... 59
- 3.10. Adaption to a Display ........................................ 60
  - 3.10.1. Physiological Limitations .............................. 60
  - 3.10.2. Direct Colour Frame Buffers ........................... 62
  - 3.10.3. Monochrome Frame Buffers ............................. 62
  - 3.10.4. Frame Buffers with Indexed Colour .................. 63
  - 3.10.5. Uniform Quantization and Ordered Dithering ....... 63
  - 3.10.6. Tapered Quantization and Error Dithering .......... 64
- 3.11. Survey Conclusions .......................................... 66

### 4. GAS - AN EXPERIENCE WITH A LARGE OBJECT ORIENTED RENDERING SYSTEM ........................................... 68

- 4.1. Components of Object Oriented Programming .............. 69
4.1.1. Objects .......................................................... 69
4.1.2. Classes ......................................................... 70
4.1.3. Methods ....................................................... 70
4.1.4. Inheritance .................................................. 70
4.1.5. Metaclasses and Factories ................................. 71
4.2. GAS Infrastructure ............................................ 71
4.3. Efficient Representation of Objects vs. Flexibility ..... 75
4.4. GAS Class Structure .......................................... 75
4.4.1. The Geometry Classes .................................... 77
4.4.2. The Device Model .......................................... 78
4.5. Experience with C++ and Object Oriented Methodology 79
4.6. Efficiency of C++ Programs ................................. 80
4.7. The GAS Renderers ........................................... 81
4.8. The CSG Definition Language .............................. 82
5. MOVIEBOX ......................................................... 85
5.1. Moviebox Architecture ...................................... 85
5.1.1. The Object and Polygon Database ..................... 85
5.1.2. The Command Processor ................................ 85
5.1.3. The Rendering Pipeline ................................. 86
5.2. Experience with C and Object Oriented Methodology ..... 88
5.3. A Conceptual Model .......................................... 89
5.4. Benchmarks and a Performance Model ................... 90
5.4.1. On MOVIE-BYU ......................................... 90
5.4.2. A Coarse Performance Model .......................... 92
5.4.3. On the Test Scenes ...................................... 94
5.4.4. Test Results ................................................ 95
5.4.5. Other Timing Results .................................... 98
5.4.6. Performance Test Conclusions .......................... 99
5.5. Shortcomings and Extensions ................................ 100

APPENDICES:
A. COLOUR PLATES .................................................... A1
B. REFERENCES ......................................................... B1
1. ABOUT THIS THESIS

I have for the last ten years been convinced that single-user workstations are the future of computing. This conviction was due to hearing Alan Kay speak about Smalltalk at an MIT summer school session in 1978, and to some Norwegian visionaries: Knut Skog and Trygve Reenskaug - who brought great news from their sabbatical years at CMU and Xerox PARC; Sverre Frogner and Frank Lillehagen - who were scrupulous enough to start manufacturing engineering workstations in Norway in 1978, and who invited me to take part in that experience.

It has been said that on a single-user workstation you have the whole processor to yourself, so you will not have to optimize on computing time the way you had to when paying for time-sharing. I find this viewpoint only partly valid. The workstations we see today are only just good enough for many of the tasks that we want them to handle. And whenever there are free cycles, we can always use them for improving the user interface. This is the background for my interest in improving the performance of rendering algorithms.

By real time rendering performance in Computer Graphics, the conventional meaning is that the picture is updated once per screen refresh, which means at least 25 updates per second. This frame rate can be realized only at a very high cost, using specially manufactured circuitry utilized in the flight simulator market. For my work, the primary equipment is medium to low cost workstations where real time rendering is far beyond expectations. Thus my goal for rendering performance is what I call interactive time. This means that the delay introduced by rendering does not introduce instabilities in the operations of a human operator. For instance, when trying to rotate a three-dimensional object into proper position by turning a potentiometer knob. Technically speaking, the feed-back system formed by the rendering system and the human operator should be stable.

The algorithms developed in this thesis are related to different stages of the rendering process. In order to place these into a larger context, I have found it worthwhile to present a survey of the rendering process biased towards performance issues. Another reason is that there does not, to my knowledge, exist any single good source which describes the rendering process.
Another similarity between the described algorithms is that they tend to make some process, normally carried out in software, suitable for the implementation on a specialized graphic processor.

In the case of the RasterOp-based rendering algorithms, software algorithms have been restructured, moving their massive computations into simple RasterOps realizable by simple, specialized circuits. In the case of the clipping algorithm, a recursive algorithm has been transformed into a simple and fast push-down automaton, making possible simple boxing tests for structured objects.

1.1. Overview

The figure below presents a functional block diagram of system components involved in a graphic rendering system on a workstation.
Fig. 2.1
Structural diagram of a rendering system with references to the developed algorithms and systems.

GAS and Moviebox, the integrated rendering systems, represent more or less the whole diagram.
The RasterOp-based rendering algorithms cover the lower part of the rendering pipe-line, making it possible to move these stages into a dedicated processor, while retaining flexibility.

The work on an efficient boxing and clipping engine relates to the stages named *structure traversal* (which is the most efficient place to do boxing) and the *viewing and clipping* stage.

The article on structured usage of a bit-map display applies to the "screen manager" stage.

### 1.2. Thesis Structure

Because this thesis describes three relatively independent algorithm developments for specific rendering tasks and two different rendering systems, I have chosen to divide the thesis into four volumes: one for each of the task specific algorithms, and this first volume presenting an overview, and describing the two approaches towards integrated rendering systems.

Each volume has its own list of references and appendices.
2. WHAT HAS BEEN ACCOMPLISHED?

The dissertation, while having its emphasis on computer graphics technology, also makes use of techniques from other fields of computer science and mathematics: Object-Oriented Programming (GAS), Compiler Construction (GAS Parsers), Linear Algebra (4D-clipping), Numerical Methods (Lighting and Shading), and Image Processing (Anti-Aliasing).

2.1. GAS

GAS (Graphics for All Seasons) is a toolbox of different rendering algorithms, providing both a depth-priority renderer, a scan line renderer, and RasterOp-based z-buffer renderers. The toolbox is built as a large set of object classes written in a very early version of C++. It has been used as an environment and testbed for the RasterOp algorithms and several of the modules used in the Moviebox system. Moreover, it was an early attempt (1984/85) for evaluating how well object-oriented programming languages would be suited for implementing rendering algorithms.

In this report, most emphasis has been put on the object-oriented aspects of GAS, since most of the algorithms are discussed in the separate RasterOp algorithm report and under the Moviebox rendering system.

2.2. Moviebox.

Moviebox is a more dedicated rendering engine with only a fast scan line z-buffer algorithm, but designed for optimal performance on modern workstation hardware. Moviebox has an architecture which makes it easy to divide the rendering stages between different rendering processors.

Moviebox contains several new algorithms (or enhanced old algorithms):

- Optimized coordinate resolution and special fixed-point arithmetic in the hidden surface algorithm and shading algorithm make rendering fast on simple microprocessors.

- The renderer uses a special shading technique to take advantage of graphic screens with a limited number of colours.
• Specially developed fast algorithms for boxing and clipping make navigation in huge environments possible.

• Full flexibility viewing algorithms support all types of projections: isometric, axonometric, perspective, and oblique.

With the limitation of running on general-purpose workstations, and not on special purpose graphical processors, Moviebox has proven to be a very fast rendering system. Speed improvements of 3-10 times compared with other rendering systems have been measured.

Another advantage of Moviebox is that it dynamically supports small and large geometric models and all types of screen or plotter resolutions.

Though using specialized fixed-point arithmetic and special programming to get high speed, the "C"-coded system has shown good portability across different workstation configurations. The system is currently implemented on Sun-3 and Sun-4 workstations (Sun UNIX with Pixrect interface or X Window System), ICAN raster workstation (MUNIX - UNIX System 3), VaxStation (VMS with UIS or X Window System), Transputer Network, and IBM PC (Microsoft C and EGA monitor).

Moviebox is in daily use at various test sites and has been used for production of video animation sequences shown at ESA (European Space Agency).

2.3. Rendering Algorithms for a RasterOp Engine

While raster operations (RasterOps) have formed a natural set of basic operations for document-oriented applications on workstations, they have so far been disjunct from rendering algorithms. The rendering algorithms developed in the GAS system show that rendering algorithms implemented on the basis of an extended set of RasterOps can be both effective and flexible.

The algorithms discussed include:

• A simple Z-buffer hidden surface algorithm.

• Enhancements with hidden lines and transparency.
- Shadow casting from multiple light sources.
- Direct rendering of Constructive Solid Geometry by fully exploiting the arithmetic capabilities of extended RasterOps.

Although the algorithms were developed for a special bit-slice rendering engine, the algorithms and RasterOp basics can prove to be a useful architecture for rendering systems also on workstations and graphic systems based on other technology.

In an M.Sc. thesis by T. Melen and A. Sommerfelt [MELEN88b] under my guidance, it has been shown how the CSG rendering algorithm can be implemented as a scan line renderer on a Transputer network.

The presentation of the RasterOp-based algorithm appears in volume two.

2.4. A Fast Polygon Clipper

In volume number three, various aspects of clipping in computer graphics are discussed. For synthesis, it takes a system approach, analyzing the environment before optimizing the clipper for huge scene traversals. Based on this analysis, a fast boxing algorithm and a fast and compact polygon clipping algorithm are designed.

The report also develops a framework for realizing general clipping networks.

An excerpt of the discussion on modelling clipping in homogeneous coordinates has been published as a technical note in Computer Graphics Forum, No. 3, 1989 [ZACHRISEN89f].

2.5. Adding Structure to Bit-Map Displays.

This separate paper, forming volume number four, discusses how the bulk frame buffer of raster graphic displays can be structured to give the application programmer simple control with overlays and double buffering. A recursive algorithm for "layered" structure on raster graphic workstations is described.
This paper has been published in IEEE Computer Graphics and Applications, July 1984 [ZACHRISEN84].
3. A SURVEY OF THE RENDERING PROCESS
WITH COMMENTS ON IMPLEMENTATION ISSUES

The following chapter will survey the whole rendering pipeline, from the geometric model to the display screen. Candidate algorithms for the GAS and Moviebox rendering systems are discussed, particularly with respect to their suitability for a modest workstation. However, as the computational capacity of a "modest workstation" seems typically to increase by a factor of two every third year, also algorithms a little beyond interactive performance today have found its way into this survey.

3.1. Modelling

The model determines many of the properties of a system. How this affects the functionality of the system can be illuminated by considering two common drawing programs: MacPaint and MacDraw†. Though both programs are capable of drawing two-dimensional entities such as rectangles and circles, their capabilities are very different. This is due to the representational differences: in MacPaint a bit-map representation, and in MacDraw a list of object types, represented with their current geometric parameters and visual attributes.

In graphic systems, criteria for determining which representation shall be used for the scene are:

- The model must provide the necessary operations to the user.
- The representation must provide efficient display.

The most common representations are described below.

One of the problems we face is that of integrating objects which have disjoint geometric representations (man-made geometry, models of natural objects, and actual sampled data from natural objects; pixels or voxels). The normal way of integration is currently either by blending of raster images or by using the z-buffer.

† MacPaint and MacDraw are copyrighted products of Apple Computer.
3.1.1. Polygon Databases

The most commonly used representations in graphic systems are polygons. Solids are represented by their bounding surfaces, approximated by polygon meshes.

As an example of this, Moviebox represents objects ("parts") by a structure similar to the so-called "winged edge" representation [BAUMGART72]. This representation makes it possible to detect which polygons are adjacent across an edge or which polygons share a vertex. More complex scenes are formed by grouping together instances of parts or instances of groups. Visual attributes (colour, visibility, texture etc.) are part of the "instance". Data model diagrams are shown below†.
Fig. 3.1.

In Moviebox, each "part" is represented by a list of polygons. Each polygon is represented by a list of edges. Edges have references to vertices. For detecting polygons adjacent to a vertex (necessary when averaging vertex normal), vertices have references to adjacent edges, and edges have a reference to their polygon.
Fig. 3.2.
In Moviebox, complex scenes can be constructed by making multiple "instances" of "parts" or "groups", and collecting these together into larger assemblies, called "groups". In addition to carrying the geometric transformation, the instance also contains other visual attributes.

3.1.2. Spline Surfaces

Another common way of representing solids in terms of bounding surfaces is spline representation. The advantage of using higher order surfaces is that they can approximate smooth surfaces with fewer patches, and can give a smoother presentation.

Various representations are in use: B-splines, Beta-splines, Box-splines etc. [BARSKY84]. Though higher order surface patches can often be handled within the same framework, quadric and cubic patches are most common.

One of the representations, "non-uniform rational B-splines" (NURBS), has been accepted so as to make its way into the set of graphic primitives in PHIGS+ [VERBECK87].
For displaying higher order surfaces, two techniques are possible: either
direct scan conversion of the patches, or approximation of the surface with
polygons before display. Though there are examples of graphic renderers that
are based on higher order (mostly bicubic) patches [LANE80,
VANOVERVELD86], breakdown of patches into polygons is most common.
By computing the true surface normal at polygon corners, and using this for
Gouraud or Phong shading, a relatively good visual presentation of the
surface is possible.

3.1.3. Solid Models

With solid modelling, complex objects are formed by the union, intersection,
and subtraction of simple primitives [MÄNTYLÄ88]. The representation is
commonly called Constructive Solid Geometry (CSG). A CSG representation
can be directly rendered by ray tracing renderers.

Algorithms for high speed, direct rendering of CSG geometry have been
developed in the GAS system [ZACRISEN89b] (see also Ch. 4).

3.1.4. Models of Nature

It is often desirable to add elements of nature to the images of man-made
objects. And it is also a challenge in itself to closely regenerate natural
phenomena by a computer modelled process.

Special techniques, normally including some stochastic element, have been
successfully used to model terrain, fire, trees, clouds, and water.

Another field in the borderland between computer graphics and image
processing which is starting to attract researchers, is the topic of volume
rendering. In volume rendering, the data set is a three-dimensional array of
data (voxels) originating from MRI or CT scanners etc.

3.2. Structure Traversal

Today there is an on-going competition between computer graphic systems
using one of two display architectures:
"Direct view".

Structured display file.

With the direct view concept, the image is drawn directly from the geometric model under control of the application program.

With the structured display file concept, the structured display file is compiled and updated from the geometric model. Then the structured display file is traversed by a display processor.

The display file stems back to the dawn of computer graphics, where the display file was the interface between a CPU running an application and a DPU which accessed the display file over a DMA channel, refreshing the CRT-image as fast as possible to avoid flicker.

Typical exponents of the direct view concept are raster graphic workstations, made popular during the 1980's.

With the display file, it will always be a question how much structure should be added to it.

With the traditional CRT refresh displays, display files were merely sequences of drawing instructions, perhaps linked together by jump-instructions. Later, better subpicture structures were made possible by push-jump and pop-jump instructions. High performance displays added geometric transformation capabilities. With subpicturing and transformation capabilities added, we have what we call structured display files.

The major drawback with the structured display file architecture is that the application program has to maintain two data structures, and ensure consistency in response to user interactions. To avoid the two parallel representations, some applications store their non-graphical data along with the graphical data in the display file, which we then call a graphical data structure.

One major advantage with the structured display file is that it allows distribution of the graphic system over a network. Complex analysis can be run on a large host computer, downloading necessary changes to the display.
file in the workstation. The screen can be rapidly redrawn from the display file allowing locally controlled panning and zooming of the data set.

Device independent computer graphic systems like GPGS-F [ZACHRISEN76], GKS [ENDERLE84], PHIGS+ [VERBECK87], and PEX [PEX88] have adapted the structured display file concept to provide a unified interface to the display equipment; whether it be a plotter, storage tube, raster display, or a refresh calligraphic display. Whereas GKS does not allow any subpicturing structure, GPGS-F allows subpictures and a limited set of instancing transformations. PHIGS+ and PEX allow the application programmer to build and edit structured display files, and extend the functionality of the display file traverser with the ability to branch conditionally depending on boxing information. The latter is useful for switching between alternate representations, depending on the viewing distance.

But, when branching is allowed, why not add other control structures: do-loops and case-statements, and general parameterization in subpictures? The distinction between a structured display file and a general display programming language may become vague. The latter is exemplified in PostScript-based systems for printers as well as screen layout control [ADOBE87]. With PostScript the display instructions, subpictures, and macros have been replaced with a threaded, interpretive language extended with graphic functionality.

3.3. Viewing Transformations

A general viewing module was implemented for GAS and later translated to Moviebox usage. The viewing follows the same principles for specifying views as proposed in the SIGGRAPH CORE specification [GSPC79]. This way of specifying views allows any planar geometric projection to be specified (isometric, axonometric, perspective, or oblique). This topic has been discussed by Michener and Carlbom [MICHENER80], Foley and van Dam [FOLEY84], and Singleton [SINGLETON86]. However, there are contradictions between the view derivations in the mentioned sources. The uvn-coordinate system described here is right-handed while the original specification of SIGGRAPH CORE demanded a left-handed viewing
reference coordinate system, and the perspective transformation prepares the coordinate data for clipping in homogeneous coordinates, not in three-space as is customary. Therefore, the matrices implementing the view transformation for a perspective projection will be shown in detail.

For a perspective view, the parameters specifying the view are as follows†:

\[ \text{VRP}_{(x,y,z)}: \] View reference point (focal point).
\[ \text{COP}_{(x,y,z)}: \] Centre of projection (eye position), relative to VRP.
\[ \text{NORM}: \] View plane normal (defines the view plane).
\[ \text{UP}: \] Up-vector (projects to the vertical axis).
\[ U_{\text{min, max}}, V_{\text{min, max}}: \] Window size.
\[ \text{fd}: \] Front (hither) clipping plane distance (from VRP).
\[ \text{bd}: \] Back (yon) clipping plane distance (from VRP).
\[ \text{vd}: \] View plane distance (from VRP).

Derived parameters:
\[ \text{n}: \] Unit normal.
\[ \text{v}: \] Unit view up.
\[ \text{u}: \] Unit view right.
\[ \text{COP}_{u,v,n}: \] Components of vector to COP along computed \( u, v, n \) axes.
\[ \text{SIZ}_{u,v}: \] Window size \((= U_{\text{max}} - U_{\text{min}}, V_{\text{max}} - V_{\text{min}})\).
\[ \text{CEN}_{u,v}: \] Coordinate of window centre
\[ (= (U_{\text{max}} + U_{\text{min}}) / 2, (V_{\text{max}} + V_{\text{min}}) / 2).)\]

The \( u, v, n \) vectors defining the viewing reference coordinate system are derived as follows:

\[
\text{n} = \frac{\text{NORM}}{||\text{NORM}||} \\
\text{v} = \frac{\text{UP} - (\text{UP} \cdot n) \, n}{||\text{UP} - (\text{UP} \cdot n) \, n||}
\]

† Here, three-element row vectors, representing points and normals, are denoted by boldface names.
\( \mathbf{u} = \mathbf{v} \times \mathbf{n} \)

First, we translate the centre of projection to the origin (the centre of projection is defined relative to the view reference point):

\[
\mathbf{T} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
(-\mathbf{VRP} \cdot \mathbf{COP}) & 1
\end{bmatrix}
\]

Then we will have to change coordinate system so that our new coordinate system is defined by the computed \( \mathbf{u}, \mathbf{v}, \) and \( \mathbf{n} \) unit vectors.

\[
\mathbf{R} = \begin{bmatrix}
\mathbf{u}^t & \mathbf{v}^t & \mathbf{n}^t & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

The \( \mathbf{T} \cdot \mathbf{R} \) matrices comprise the so-called *view orientation transformation*. After being transformed by \( \mathbf{T} \cdot \mathbf{R} \), the geometry is relative to a coordinate system at the COP, with axes aligned with the projection plane. This coordinate system is known under various names: the *viewing reference coordinate system*, the *viewing coordinate system*, or the *eye coordinate system*. 
Fig. 3.3.

a) A viewing transformation is basically specified by a view reference point (VRP), a view plane normal (NORM), and a view up vector (UP). The view plane can be offset (vd) from the reference point. The window is specified in the view plane (u, v coordinates).

Additionally, back and front clipping planes can be defined in terms of their distance to the VRP. For a perspective view, the centre of projection (COP) is specified (relative to VRP).

b) After the view orientation transformation, the coordinate system has been aligned with the projection plane (and the up vector).

The viewing reference coordinate system is useful for doing shading computations and removing back faces. The eye position has been transformed to the origin. The view orientation transformation as specified here will only contain translations and rotations, and will thus have angle preserving properties.

Then we shall compute what is called the view mapping transformation to map the geometry into a normalized viewing pyramid, which is useful for clipping.

Before computing the view mapping transformation, we need to compute the components of the COP vector in the viewing reference coordinate system:

\[
\text{COP}_u = \text{COP} \cdot u
\]
\[ \text{COP}_v = \text{COP} \cdot v \]

\[ \text{COP}_n = \text{COP} \cdot n \]

We shear to centre the window on the z axis:

\[
\mathbf{H} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
-\frac{\text{COP}_u - \text{CEN}_u}{\text{COP}_n - \text{vd}} & \frac{1}{\text{COP}_n - \text{vd}} & 1 & 0 \\
\frac{\text{COP}_v - \text{CEN}_v}{\text{COP}_n - \text{vd}} & 0 & 0 & 1
\end{bmatrix}
\]

Then we scale to create the canonical view volume:

\[
\mathbf{S} = \begin{bmatrix}
\frac{\text{(COP}_n - \text{vd}) \ast 2}{\text{(COP}_n - \text{bd}) \ast \text{SIZ}_u} & 0 & 0 & 0 \\
0 & \frac{\text{(COP}_n - \text{vd}) \ast 2}{\text{(COP}_n - \text{bd}) \ast \text{SIZ}_v} & 0 & 0 \\
0 & 0 & 1 & \frac{\text{COP}_n - \text{bd}}{\text{COP}_n - \text{vd}} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Fig. 3.4.
a) For oblique views, the window centre must be aligned to the z-axis by a shear transformation. b) Scaling the view frustrum is needed to produce the so-called canonical view volume (see Fig. 3.5. below).

Finally, the **perspective transformation** transforms the canonical view volume into the volume defined by: \(-1 \leq x \leq 1, -1 \leq y \leq 1, -1 \leq z \leq 0\):

\[
P = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & \frac{COP_n - bd}{fd - bd} & -1 \\
0 & 0 & \frac{COP_n - fd}{fd - bd} & 0
\end{bmatrix}
\]

The centre of projection (or eye position) is transformed into a point at infinity (with homogeneous representation \([0, 0, 1, 0]\)). The "far" clipping plane is mapped into the plane \(z = -1\), and the near clipping plane is mapped to the plane \(z = 0\).
Fig. 3.5.

a) The canonical view volume for perspective viewing and b) the view volume after being mapped by the perspective transformation \( a = (\text{COP}_n - \text{fd}) / (\text{COP}_n - \text{bd}) \).

Homogeneous coordinate clipping can be performed after the perspective transformation (see [ZACHRISEN89c]). The homogeneous division accomplishes the perspective (z-coordinate) foreshortening and preserves relative depth and straight lines and planes. The latter is important when preparing data for a hidden surface algorithm, as in Moviebox or the GAS system.

The perspective viewing transformation becomes:

\[
M = T \ast R \ast H \ast S \ast P
\]

The following conditions result in singularities and should be avoided:

- **NORM** is parallel to **UP**.

- The hither clipping plane is not between the back clipping plane and the **COP**.

- The view plane is at **COP** or behind **COP**.
For a parallel projection the computations are similar, however a direction of
projection vector (DOP) is specified instead of the centre of projection
(COP) [MICHERNER80].

3.4. Clipping

A survey of various clipping algorithms and environments is provided in the
third volume of this thesis [ZACHRISEN89c] and is thus omitted from this
survey.

3.5. Hidden Surface Removal

Since Ivan Sutherland wrote his "Ten Unsolved Problems in Computer
Graphics" article in 1966, where the problem of removing hidden lines was
included [SUTHERLAND66], numerous articles have been published
describing different solutions to hidden-line and hidden-surface removal.

In 1974, Sutherland, Sproull, and Schumacker published a survey of ten
hidden surface algorithms, characterizing the algorithm from the point of view
of sorting the geometry [SUTHERLAND74]. How an algorithm sorts the
geometry strongly affects how the algorithm is able to exploit geometric
coherence in the scene and coherence between frames.

Because the main target of this work is inexpensive raster workstations, the
discussion concentrates on algorithms considered relevant for these. The
algorithms will be categorized according to their graphic output primitives.
Only features relevant for polygon types of scenes are considered.

3.5.1. Geometry Engines and the Z-Buffer Algorithm

It is a paradox that while there have been developed scores of sophisticated
algorithms for the removal of hidden surfaces and presentation of high quality
images, the algorithm with the by far the largest commercial success is the
"brute force" depth-buffer algorithm [NEWMAN79]. The commercial
success is due to the incorporation of the depth-buffer algorithm into the
firmware of high-speed graphic workstations. The typical workstation is
equipped with an intensity buffer and a z-buffer along with hardware for transforming, clipping, and shading polygons.

While shading algorithms until recently only exceptionally incorporated support for Gouraud (and "Cheap Phong") shading [ZACHRISEN81], the most advanced workstations today also have support for full Phong shading.

The interface between the user and the graphic output engine is typically a structured display file containing transformations, polygons, and attributes. For simpler systems, transformations, clipping, and lighting computations must be performed by the host computer, providing a rendering engine interface of polygons with shading colours specified at the vertices.

The virtues of the full-frame z-buffer hidden surface algorithm are:

- Processing time is $O(n)$. 
- Infinitely many polygons can be rendered (no intermediate storage).
- Restricted output quality (no transparency, no anti-aliasing, no texturing).

3.5.2. Polygon Output Algorithms

The "GKS Engines" are workstations particularly constructed to support this standard.†† Though a "cell array" interface to the frame buffer is normally provided, it is desireable to make use of the built-in polygon drawing (and filling) circuitry for speeding up the display process. "GKS Engines" limit the shading technique to simple "constant (cosine) shading" (no interpolation). A hidden surface algorithm appropriate for this kind of equipment must either sort polygons back-to-front, or clip polygons against each other such that only the non-hidden parts of polygons are output.

† In this section, $O(n)$ means linear growth with the number of polygons.
3.5.2.1. List Priority Algorithm

Newell, Newell, and Sancha have implemented a hidden surface algorithm which is able to sort polygons back-to-front so that they can be output to a raster screen "overpainting" the previous contents [ROGERS85]. The algorithm uses efficient extent testing to determine the major sorting sequence, and goes through successively more time-consuming operations to establish a depth relation between polygons.

Algorithm characteristics:

- The algorithm is efficient with simple environments.
- The computation time grows with $O(n^{3/2})$.
- When depth complexity is high, pixels are overpainted many times.
- Only partial sorting is required between successive frames.

The algorithm may simply be reconfigured to output polygons front-to-back, which is suitable preprocessing for a hidden-line algorithm.

3.5.2.2. "Cookie-Cutter" Subdivision Algorithm

The Weiler-Atherton algorithm has many of the same characteristics as the list priority algorithm [WEILER77]. The big difference is that the Weiler-Atherton algorithm outputs polygons front-to-back, and that output polygons have been clipped so as not to overlap polygons with higher priority.

The algorithm can also be used for computing shadows; the polygon database is run through the hidden surface algorithm as viewed from the light source, and all visible polygon descriptions are added to the data base as surface detail. Upon rendering, only tagged polygon parts are illuminated from the light source.

Algorithm characteristics:

- The algorithm is relatively efficient with simple environments.
- The computation time grows with $O(n^{3/2})$. 
3.5.2.3. Binary Separating Planes

Rather than sorting the scene for each frame, it is possible to sort the scene contents once and for all into a binary tree. The concept is called "Binary Separating Planes" or BSP for short [FUCHS80]. Each node in the tree contains one polygon with a plane equation vector defined with a "front" direction, and pointers to two subtrees. One subtree contains polygons inside the "front" halfspace defined by the extended polygon plane, and the other subtree contains polygons inside the "back" halfspace. The polygons can be output back-to-front for a "painter's algorithm" or front-to-back for simple "bed-of-nails" anti-aliasing. All that is necessary during output is to traverse the binary tree and perform a four-element dot-product between the homogeneous representation of the eye point and the plane equation vector at each node. Which of the subtrees is traversed first depends on whether the eye point is in front of or behind the dividing plane. Back-faces can be removed with the same dot-product test.

A recursive tree traversal pseudo-code algorithm for outputting polygons back-to-front and removing back-facing polygons is shown below:

```c
traverse_subtree(node, eye) {
    if (node != NIL) {
        dot = dotproduct(node.normal, eye);
        if (dot > 0) {
            traverse_subtree(node.rear, eye);
            draw_polygon(node.polygon);
            traverse_subtree(node.front, eye);
        } else {
            traverse_subtree(node.front, eye);
            traverse_subtree(node.rear, eye);
        }
    }
}
```

During tree building / scene pre-sort, polygons are sorted into the correct subtree/halfspace. Whenever a polygon intersects the node's plane, the polygon is divided into two polygon parts which are inserted into the respective subtrees.

† Using homogeneous coordinates allows the eye point to be at a finite distance (xe, ye, ze, 1) for perspective projection, or at infinity (dx, dy, dz, 0) for parallel projection.
The algorithm has been implemented in the GAS environment (see Ch. 4).

Algorithm characteristics:

- The algorithm is relatively efficient with simple environments.
- Generation of multiple frames is very efficient.
- Changing relative positions of objects in the frame requires special attention.
- Polygons can become very fragmented, depending on how the tree is built.
- Pixels are overpainted many times if polygons are output back-to-front.

The BSP-algorithm can also be used for handling the depth relation between objects or clusters of geometry, rather than handling all tiny polygons in the scene.

3.5.3. Scan Line Output Algorithms

The pixel output algorithms process the image scan line by scan line, outputting sequences of pixels to the frame buffer. These algorithms require direct access to the frame buffer or at least efficient transport of pixel scans. For simple scenes, the transportation requirements can be alleviated somewhat when using some kind of pixel compression (run length coding etc.) during pixel transport.

3.5.3.1. Scan Line Z-Buffer Algorithm

The z-buffer algorithm can be implemented in a scan line algorithm rather than working random access on the full frame buffer. In the scan line implementation, all transformed polygons are stored in a y-bucket list before output starts.
Algorithm characteristics:

- All geometry is collected in a bucket sorted list before output.
- Efficient for simple and relatively complex scenes.
- Processing time is $O(n)$.
- No span information is available for output "run" compaction.

Though only full frame z-buffer and scan line z-buffer algorithms are discussed here, z-buffer algorithms can also be implemented on slices of the full frame buffer. This is a viable solution, especially for parallel processing [MELEN88a].

3.5.3.2. Spanning Scan Line Algorithm

There exist multiple implementations of spanning scan line hidden surface algorithms. Most of these are based on the Watkins Algorithm [WATKINS70]. Unlike the z-buffer algorithm which computes the relative priority at each pixel, the spanning scan line algorithm does this only at segment edges. However, the computations involved at segment edges in the spanning scan line algorithm are more complicated than the depth calculation in the z-buffer algorithm.

In addition to the $y$-sort (bucket sort), the spanning scan line algorithm requires an $x$-merge for each edge (when inserted in the active edge list), an $x$-sort of segments (bubble sort) for each scan line to keep the list sorted on increasing $x$, and a search for the visible span for each span encountered.

Algorithm characteristics:

- All geometry is collected in a bucket sorted list before output.
- The algorithm is efficient with simple environments (comparable with scan line z-buffer).
- With simple and medium complex scenes, the span depth search is dominant.
• With very complex scenes, the x-merge becomes dominant (increases with $O(n^{3/2})$).

• Compression of output data to "runs" is implicit in the algorithm.

3.5.3.3. A-Buffer

The A-buffer algorithm was designed to provide anti-aliased output, while retaining high efficiency [CARPENTER84]. The algorithm uses subpixels for solving the depth buffer relations at polygon edges, where aliasing otherwise would occur. In reported implementations, 8x4 or 8x8 subpixels are used.

For an image with approx. 1000 polygons at 512x512 resolution, statistics show that 85.6% of fragments cover a whole pixel, 13% are on a simple edge, while the rest (<2%) have more complex shapes [ABRAM85]. This means that the vast majority of pixels can be handled by normal z-buffer logic, while the anti-aliasing computations on subpixels with a higher complexity will only affect a minor part.

In a scan line implementation, the A-buffer has most of the same characteristics as the scan line z-buffer algorithm.

Algorithm characteristics:

• All geometry is collected in a bucket sorted list before output.

• Efficient for simple and relatively complex scenes (slower than z-buffer, but faster than z-buffer with supersampling).

• Processing time is $O(n)$.

• High quality, anti-aliased output.

• No span information is available for output "run" compaction.

3.5.3.4. Ray Tracing

Ray tracing algorithms have gained immense popularity recently, particularly in the academic community. The reason for this is that ray tracing algorithms
can generate images with very high quality and thrilling effects, and that ray tracers can be implemented with a minimum of effort compared with the complexity of writing a scan line render. However, simple ray tracers are immensely slow. Processing time for rendering a scene with two spheres is reported to 0.5 hours on a Sun-3 [WHITTED86].

A (reverse) ray tracing algorithm follows the reverse path of rays which reach the eye of the observer. The first order rays are cast from the observer through each pixel on the display. Whenever a ray intersects an object, several new rays are cast: one towards each light source (for determining shadows), one in the direction of reflection (for glossy surfaces), and one in the direction of the refracted ray (for transparent objects).

![Diagram of ray tracing](image)

Fig. 3.6.
When a ray (v) intersects an object, new rays are cast towards each light source (L_j), in the direction of reflection (r), and in the direction of refraction (p).

Efficient implementations of ray tracers require some efficient searching structure to be applied to the scene (octrees, or BSP trees etc.). With this added complexity ray tracers can to some extent compete with scan line renderers (see Ch. 5.4.5.).

Anti-aliasing is normally implemented in ray tracers as an adaptive subdivision scheme: whenever the computed colour of each pixel corner deviates too much, the pixel is subdivided into four subpixels, and (5) new rays are cast.
Fig. 3.7.
A common way of casting rays in a ray tracer and to reduce aliasing, is to fire primary rays through each corner of the pixel (P1..4), and if the computed intensity of each corner deviates too much, subdivide each pixel into four, and cast new rays through subpixel corners (S1..5). This subdivision can be repeated if necessary. The pixel intensity would be computed as a weighted sum of intensity samples.

So-called distributed ray tracing will also oversample the image, but in addition to avoiding aliasing, such effects as penumbras (soft shadows), depth of field, and motion blur can be generated[COOK84].

Algorithm characteristics:

- Simple to implement, but very inefficient in simple implementations.
- Can increase speed by exploiting spatial coherence.
- Supports global illumination model with shadow casting, reflection, and refraction.
- Can readily support texture mapping and anti-aliasing (adaptive subdivision).
• Can support effects such as depth of field, soft shadows, and motion blur (distributed ray tracing).

3.6. Aspects of a Z-Buffer Renderer

The primary candidate for a fast, flexible, general purpose rendering algorithm seems to be the z-buffer algorithm. Some of the more detailed aspects of z-buffer algorithm implementations will be discussed in the following.

3.6.1. Scan Line or Scan Buffer Rendering

The choice between a scan line or a scan buffer algorithm is taken based on memory usage and degree of realism required. The performance of a z-buffer algorithm running as a scan line or as a scan buffer algorithm is not very different. For a very simple scene test, Roberts reports a 1 : 1.7 performance ratio in favour of the scan buffer algorithm (see Ch. 5.4.5).

3.6.1.1. Memory Usage

For a scan buffer algorithm, the memory usage in the display subsystem is independent of the scene complexity. If the z-buffer and the screen buffer need $P_b$ bytes per pixel, the memory requirements for a screen of size $S_x \times S_y$ would be:

$$M_{sb} = P_b \times S_x \times S_y.$$  

For a typical high-resolution (1024 x 1024) RGB screen with a 32 bits z-buffer, the memory usage is 7 bytes per pixel, a total of 7 Megabytes.

The scan line algorithm needs the depth-buffer and the screen buffer only for a single scan line. In addition, it needs to store (and sort) each polygon in a bucket-sorted list. Moreover, the algorithm needs an active edge list for each polygon intersecting the active scan line.

Assuming an even distribution of polygons in space, the estimated number of polygons intersecting a scan line would be $(D_c \times F_e \times (S_x / S_y))^{1/2}$, where $F_e$ is the number of polygons in the scene, $D_c$ is the depth complexity (see Ch. 5.4.2.), and $S_x$ and $S_y$ are the screen dimensions. $A_b$ is the size of a polygon.
entry in the active edge list, and \( B_b \) is the size of a polygon in the bucket list. The memory required would be:

\[
M_{sl} = \begin{align*}
Z\text{-buffer and screen buffer:} & \quad P_b \cdot S_x + \\
Polygons \text{ in bucket list:} & \quad B_b \cdot F_e + \\
Polygons \text{ in active edge list:} & \quad A_b \cdot (D_c \cdot F_e \cdot (S_x / S_y))^{1/2}.
\end{align*}
\]

For simple scenes (\( F_e = 100 \)), the scan line algorithm will use much less memory than the scan buffer algorithm. For complex scenes, the scan buffer algorithm will gradually become at advantage. The break-even point will be when:

\[
B_b \cdot F_e = P_b \cdot S_x \cdot S_y,
\]

\[
F_e = (P_b / B_b) \cdot S_x \cdot S_y
\]

(When this happens, the contribution from the scan line buffer and the active edge list can be ignored.)

In a typical scan line algorithm (Moviebox), the memory usage is as follows:
\( P_b = 5 \) bytes.
\( B_b = 64 \) bytes.
\( A_b = 72 \) bytes.

With these constants, and a screen of 1024 x 1024, the break even point would be at 78000 polygons.

For high performance system, locality of data can give large performance gains with fast caching memories. The scan line algorithm has very good locality on its output buffer, but poorer locality on its active edge list. In contrast, the scan buffer algorithm has very good locality on its input stream of polygons, but poor locality on its output buffer.

3.6.1.2. Rendering Quality

The rendering quality of the scan line buffer can be improved over the scan buffer algorithm, because the whole scene (or at least all polygons on the active scan line) is available to the scan line renderer, while the scan buffer
renderer only sees one isolated polygon at a time. The z-buffer scan line renderer can thus be improved with effects such as transparency, shadow casting, and anti-aliasing, without major changes to the rendering system.

3.6.1.3. Delay

Unless the rendering system uses a double-buffer technique for output, the user sees the two algorithms quite differently. With the scan buffer algorithm, the screen will receive polygons in random order, and new polygons will appear to be drawn in front of or behind the existing scene contents. With the scan line algorithm, nothing will happen until the whole scene has been transformed and placed on the bucket list. Then the completed image will gradually appear on the screen, scan line by scan line from top to bottom.

3.6.2. Depth Resolution

While X and Y coordinates are represented in 9 to 11 bits in screen coordinates, this is certainly not sufficient for the depth coordinate. Especially when polygons are clustered in small groups in a large object space, depth resolution can be a problem inside the clusters. While some so-called solid modelling engines (along with the hardware implementation of the GAS engine [ZACHRISEN89b]) are limited to 12 planes for representing the depth coordinates, rendering systems normally allocate from 16 to 32 bits for the depth coordinate. The Moviebox system, which uses the scan line hidden surface algorithm, is set up for 20 bits for representing the z-coordinate. This is a typical compromise between efficient computation on a 32-bit computer and depth resolution.

We shall take a closer look at the depth resolution in object space:

The z-mapping provided by the viewing transformation in Ch. 3.3 (S * P) and the division by the homogeneous coordinate becomes:

\[
Z' = - \frac{(\text{COP}_n - bd) \times (Z + \text{COP}_n - fd)}{Z \times (fd - bd)}
\]

The hither clipping plane is mapped from \( Z_h = -(\text{COP}_n - fd) \) to \( Z'_h = 0 \), the yon clipping plane is mapped from \( Z_y = -(\text{COP}_n - bd) \) to \( Z'_y = -1 \).
In the z-buffer the region between the hither and yon clipping planes is mapped onto the available range of depth coordinates, in the case of Moviebox, the integer range \([0, 2^{20} - 1]\).

The reverse mapping is:

\[
Z = M(Z') = -\frac{(\text{COP}_n - \text{bd}) \times (\text{COP}_n - \text{fd})}{Z' \times (\text{fd} - \text{bd}) + (\text{COP}_n - \text{bd})}
\]

The resolution in object space (viewing reference coordinate system), when \(n\) bits are used in the z-buffer is:

\[
R(Z') = M(Z' + 2^{-n}) - M(Z') = \frac{(\text{COP}_n - \text{bd}) \times (\text{COP}_n - \text{fd}) \times (\text{fd} - \text{bd})}{2^n \times (Z' \times (\text{fd} - \text{bd}) + (\text{COP}_n - \text{bd}))^2}
\]

Due to the perspective transformation, depth resolution varies greatly in object space. Depth resolution is best near the hither clipping plane:

\[
R(0) = \frac{(\text{COP}_n - \text{fd}) \times (\text{fd} - \text{bd})}{2^n \times (\text{COP}_n - \text{bd})}
\]

and falls off quickly towards the yon clipping plane:

\[
R(-1) = \frac{(\text{COP}_n - \text{bd}) \times (\text{fd} - \text{bd})}{2^n \times (\text{COP}_n - \text{fd})}
\]

If the hither clipping plane is near the centre of projection \((\text{COP}_n - \text{fd} \ll \text{COP}_n - \text{bd})\), the difference in resolution is most noticeable.

On curve c) in the figure below, one can see that only 12% of the equally spaced tick marks fall in the rear 50% of the viewing frustrum.
Fig. 3.8.

Tick marks equally spaced in depth in screen space are transformed into object space through inverse perspective transformation. Centre of Projection (COP) is at $z = 0$. The yon clipping plane is at $z = -100$. The three curves show the effect of moving the hither clipping plane towards the COP. The hither clipping plane is at -40, -20, and -10 for a), b), and c), respectively. Note how resolution is high (marks are dense) near the hither clipping plane, and low towards the rear of the scene. Note also how this effect is amplified when the hither clipping plane is near the COP.

The depth resolution can be improved by setting the hither clipping properly, to limit the object space to be mapped into screen space. In Moviebox, the exact depth extent is computed for each scene, and the depth clipping limits are set dynamically so as to optimize the depth resolution.
3.6.3. Offset Z-Coordinate Sampling

Regardless of the depth resolution, the z-buffer hidden surface algorithm can have trouble determining the visual priority when rendering front and rear surfaces which meet at silhouette edges. With depth-buffer algorithms the sequence of polygons to render is arbitrary. When part of the surface of a solid, back facing polygons can be eliminated by a simple back-face culling test before reaching the hidden surface algorithm. However, when just part of some sheet object, both front-facing and back-facing polygons must be rendered. Front facing and back-facing polygons can differ greatly in their shading colour. Because they share the same edge, it is arbitrary whether the illuminated front-facing or darker back-facing polygon will be shown on the silhouette edge.

To solve this problem, Moviebox rounds each polygon edge to an integer coordinate of screen space during scan conversion, but does the depth sampling one half pixel offset from the computed edge.

Fig. 3.9.
Showing Moviebox's depth sampling technique. The x-coordinate is rounded to the nearest integer value. The z-coordinate (and shading value) is computed one half pixel offset from this, giving a reasonable chance of distinguishing front faces from back faces.
3.6.4. Fixed-Point Arithmetic and Rendering

While workstations have evolved into being pure 32-bit computers, the precision requirements of graphics have only slightly grown with the recent years development. For screen addressing 11 bits normally suffices. For film recorder or high-quality plotter output, resolution seldom goes beyond what can be addressed using 12 bits. Colour output is normally in the range of 64 to 256 levels (6 - 8 bits) for each primary colour, with only some of the top-line film production environments using 12 bits per colour component [LEVINTHAL84].

The Moviebox system uses 14 bits for each x and y coordinate, allowing 2 bits for subpixel addressing and 20 bits for the z-coordinate. Colours are represented either as an 8 bits colour index or as 8 bits for each of red, green and blue colour component for full colour rendering.

To compute intersections with each scan line, a DDA (Digital Differential Analyzer) algorithm [NEWMAN79] is used. Division is required for computing the x-increment, z-increment, and the colour increment (Δx/Δy, Δz/Δy, and Δc/Δy), but each scan line intersection simply requires one 32-bits fix-point add for x and one 32-bits fix-point add for the colour index. To retain enough depth resolution while staying inside a 32 bit word, the z-coordinate is represented with 20 bits and the radix point between bit 11 and 12. The x-coordinate and the colour component use a radix point between bit 15 and 16 (l.s.b. is bit 0). This makes it simple on most workstation processors to pick up the integral value. The actual formats used in the DDA are shown on the figure below.
Fix-point arithmetic is used for computing the scan line intersections for each polygon edge. 'X', 'Z', and 'C' are integral bit positions, while 'F' denotes the fractional part.

Having a fraction of \( n \) bits, means that \( 2^n \) additions can be performed before the round-off error in the DDA increments will show up in the integral coordinate. In Moviebox, at least 10 fractional bits are provided, allowing polygons to be 1024 units in size before errors are introduced.

### 3.7. Illumination Models and Shading

The emphasis in realistic synthesis of pictures is on generating a colour shade in each pixel of the display which mimics the visual impression of the eye when exposed to the actual scene and lighting conditions. However, this is always a compromise between realism and computational complexity on the one hand and cost and speed on the other.
As the illumination model must take into consideration all light sources in the display and the material characteristics of the scene objects (including light reflectance and refractance), the evaluation of the light model becomes very expensive. In order to speed things up a bit, we try to do the light computation in only a few places in the display and then do shading, that is some kind of interpolation, in-between. Normally, the light equation is evaluated in polygon vertices, and shading is used in-between. Though the light computation and the shading can be handled as two separate items as is done in the PHIGS+ and PEX specification, simple shading schemes are normally used with simple light models and more elaborate shading techniques go with more complex light models.

An illumination model is designed to make it possible to compute the light that reaches the eye from a pixel on the screen. A local illumination model takes only a single polygon, light source and the observer's position into consideration. The effect of one object casting shadow on another is disregarded. A global illumination model allows objects to obscure light, reflect light and refract light.

As global illumination models are computationally too expensive for interactive use in the immediate future, the implementation work has concentrated on the use of local illumination models.

### 3.7.1. Fast Shading Computations

As the shading computations must be carried out for each visual pixel, it is of great importance to the performance of the renderer how this is carried out. The two most well-known shading methods are the so-called Gouraud [GOURAUD71] and Phong [PHONG75] shading methods.

In all shading computations, it should be remembered that gross errors can be tolerated since the final colour range on a CRT tube is normally restricted to 256 levels (8 bits). This contrasts with the 16, 20 or 32 bit resolution

---

† The eight bits dynamic range is normally sufficient for giving the impression of smoothly changing shades of colours while limiting the resolution needed in the video-DAC's, and also adapting well to the byte representation of normal computers.
normally used for general computing. We can thus get away with some rather crude approximations.

3.7.2. Gouraud Shading.

Gouraud shading means that the colour value is computed at polygon vertices (using an appropriate illumination model) and then linearly interpolated along the polygon edges, and along a scan line from left polygon edge to right polygon edge. Gouraud suggested that for simulation of smooth surfaces using polygon mesh, the surface normal used for illumination computation at a vertex should be formed by averaging the normals of adjacent polygons.

For a monochrome display, Gouraud shading requires a linear interpolation of a single value. On a full RGB-display, linear interpolation of three separate colour components is required. It should be noted that for anything but triangles, the resulting shading is not rotationally invariant.

Gouraud shading results in a surface appearing very smooth compared to a display of polygons with constant colour. Due to the interpolation method, the surface colour will have $C^0$ continuity across edges. However, the human eye emphasizes edge information in order to aid object discrimination, and the $C^1$ discontinuities present in a Gouraud shaded image can be perceived as lighter bands, called Mach bands.

3.7.3. Phong Shading.

The Phong lighting equation divides the reflected light into an ambient term, a diffusely reflected term and a specular term. When enhanced with a distance attenuation factor (to avoid parallel surfaces being exactly the same colour), and allowing multiple light sources, the light model becomes:

$$I = I_a k_a + \sum_{j=1}^{m} \frac{I_{lj}}{d + \frac{1}{K}} (k_d \cdot L_j) + k_s (n \cdot H_j)^n$$

$I_a$ and $I_{lj}$ are the intensities of the ambient light and light source $j$, respectively. $k_a$, $k_d$, and $k_s$ are coefficients for the ambient, diffuse and specular reflection of the material. $n$ is the unit surface normal. $L_j$, $H_j$ are the unit lightvector and the unit halfvector (bisecting the angle between the
lightvector and the eyevector) for the j'th light source, respectively. d is the distance to the observer. K is a coefficient determining the effect of the distance attenuation.

Phong shading means that the surface normal is linearly interpolated along the polygon edges, and along a scan line from left polygon edge to right polygon edge. Interpolating the normal instead of the colour value will produce better continuity across edges and reduce Mach band effects. However, as shown by Duff [DUFF79], Phong shading does not eliminate Mach bands, and can in certain cases even produce larger discontinuities in the derivative across an edge than the Gouraud method.

The most noticeable effect of Phong shading compared to Gouraud shading is the ability to simulate shiny materials with distinct highlights.

When Phong shading is carefully carried out, this means that for each pixel on a scan line, the surface normal is linearly interpolated between the end vectors. But before one can perform the dot products with light vector or halfvector, the normal must be normalized to unit length. The necessary arithmetic operations per pixel are three additions for the interpolation and three multiplications, two additions, one square root computation, one reciprocal computation and then three multiplications for making the unit normal.

Fig. 3.11.

With Phong shading, surface normals are linearly interpolated across the surface and need to be adjusted to unit length before being used in intensity calculations.
While Phong shading gives the impression of smoothly curved surfaces, the actual underlying geometry is the planar polygon, which is used for drawing edges and which is used for hidden surface computations. To retain good realism in the scene, the surface normals in the polygon vertices should not defer much from the polygon normal. If the normal vectors on each side of a span are nearly colinear, the length of each interpolated normal vector is nearly the length of its predecessor. This makes the computations suitable for some iterative approximation method.

3.7.4. Fast Computation of Unit Normal

If \( c \) is the squared length of the vector \( \mathbf{n} \), we need an efficient method to find the normalizing factor \( 1/\sqrt{c} \). The iterative approach called Newton's method seems to be a good choice. Suitable iteration formulas for \( 1/c (x_{n+1} = 2 * x_n - c * x_n^2) \) and \( \sqrt{c} (x_{n+1} = (x_n + c / x_n) / 2) \) are given in the literature [KNUTH81 and BJÖRCK69]. With Newton's method and a good iteration formula, we can find a sufficiently correct approximation to a function within a few iterations, if given a reasonably correct starting value. In our case, we could of course use the formulas for \( 1/c \) and \( \sqrt{c} \) to evaluate \( 1/\sqrt{c} \), but a more direct approach, without any division would be desired. Some experiments with various formulas have been carried out. The iteration formula

\[
x_{n+1} = x_n * (3 - c * x_n^2) / 2
\]

was finally chosen. It is computationally simple and gives a fastly converging value for \( x \) towards \( 1/\sqrt{c} \).

If a small error term \( \varepsilon \) is present in the \( n \)'th approximation;

\[
x_n = (1 - \varepsilon) / \sqrt{c}
\]

we have for the \( n+1 \)'th approximation:

\[
x_{n+1} = (1 - 3/2 * \varepsilon^2 * (1 + \varepsilon)) / \sqrt{c}
\]

which shows that we have convergence to second order.
Because the interpolated, normalized vectors are computed in sequence, we can use the result of the previous computation as a starting point. The initial approximation to the normalizing factor \( x_0 \) can be obtained from the previous normalizing factor using a first order Taylor approximation:

\[
    f(b) = f(a) + (b-a) \times f'(a)
\]

For \( f(x) = 1/\sqrt{x} \), we have \( f'(x) = -f(x)^3 / 2 \) and the approximation becomes:

\[
    f(b) = f(a) - (b-a) / 2 \times f(a)^3
\]

The table below shows the Taylor approximation \( X_0 \) and the first (\( X_1 \)) and second (\( X_2 \)) iterations using the iteration method. The error values are shown in \( E_0 \), \( E_1 \), and \( E_2 \), respectively. The table shows that the error is less than \( 10^{-3} \) after a single iteration and less than \( 10^{-6} \) after two iterations.

<table>
<thead>
<tr>
<th>( c )</th>
<th>( X_0 )</th>
<th>( X_1 )</th>
<th>( X_2 )</th>
<th>( 1/sqrt(c) )</th>
<th>( E_0 )</th>
<th>( E_1 )</th>
<th>( E_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>1.41421356</td>
<td>1.41421356</td>
<td>1.41421356</td>
<td>1.41421356</td>
<td>2.3731E-09</td>
<td>2.2204E-16</td>
<td>2.2204E-16</td>
</tr>
<tr>
<td>0.6</td>
<td>1.27279221</td>
<td>1.29061132</td>
<td>1.29099428</td>
<td>1.29099445</td>
<td>0.01820224</td>
<td>0.00038315</td>
<td>1.7056E-07</td>
</tr>
<tr>
<td>0.7</td>
<td>1.18312419</td>
<td>1.19504535</td>
<td>1.19522857</td>
<td>1.19522861</td>
<td>0.01210442</td>
<td>0.00018326</td>
<td>4.2144E-08</td>
</tr>
<tr>
<td>0.8</td>
<td>1.10971114</td>
<td>1.11794128</td>
<td>1.11803398</td>
<td>1.11803399</td>
<td>0.00832284</td>
<td>9.2705E-05</td>
<td>1.1530E-08</td>
</tr>
<tr>
<td>0.9</td>
<td>1.04808154</td>
<td>1.05404123</td>
<td>1.05409255</td>
<td>1.05409255</td>
<td>0.00601101</td>
<td>5.1318E-05</td>
<td>3.7477E-09</td>
</tr>
<tr>
<td>1.0</td>
<td>0.99489090</td>
<td>0.99969521</td>
<td>1.00000000</td>
<td>0.00045109</td>
<td>3.0477E-05</td>
<td>1.3932E-09</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>0.94997409</td>
<td>0.95344347</td>
<td>0.95346259</td>
<td>0.95346259</td>
<td>0.00348849</td>
<td>1.9122E-05</td>
<td>5.7524E-10</td>
</tr>
<tr>
<td>1.2</td>
<td>0.91010687</td>
<td>0.91285839</td>
<td>0.91287093</td>
<td>0.91287093</td>
<td>0.00276406</td>
<td>1.2541E-05</td>
<td>2.5844E-10</td>
</tr>
<tr>
<td>1.3</td>
<td>0.87482367</td>
<td>0.87704949</td>
<td>0.87705802</td>
<td>0.87705802</td>
<td>0.00223435</td>
<td>5.5305E-06</td>
<td>1.2474E-10</td>
</tr>
<tr>
<td>1.4</td>
<td>0.84331747</td>
<td>0.84514827</td>
<td>0.84515425</td>
<td>0.84515425</td>
<td>0.00183678</td>
<td>5.9835E-06</td>
<td>6.3543E-11</td>
</tr>
<tr>
<td>1.5</td>
<td>0.81496483</td>
<td>0.81649227</td>
<td>0.81649658</td>
<td>0.81649658</td>
<td>0.00153175</td>
<td>4.3077E-06</td>
<td>3.4089E-11</td>
</tr>
<tr>
<td>1.6</td>
<td>0.78927615</td>
<td>0.79056624</td>
<td>0.79056942</td>
<td>0.79056942</td>
<td>0.00129326</td>
<td>3.1717E-06</td>
<td>2.0587E-11</td>
</tr>
<tr>
<td>1.7</td>
<td>0.76566125</td>
<td>0.76696261</td>
<td>0.76696499</td>
<td>0.76696499</td>
<td>0.00101374</td>
<td>2.3815E-06</td>
<td>1.1092E-11</td>
</tr>
<tr>
<td>1.8</td>
<td>0.74405052</td>
<td>0.74535417</td>
<td>0.74535599</td>
<td>0.74535599</td>
<td>0.00095097</td>
<td>1.8192E-06</td>
<td>6.6601E-12</td>
</tr>
<tr>
<td>1.9</td>
<td>0.72464999</td>
<td>0.72547484</td>
<td>0.72547625</td>
<td>0.72547625</td>
<td>0.00082624</td>
<td>1.4110E-06</td>
<td>4.1167E-12</td>
</tr>
<tr>
<td>2.0</td>
<td>0.70683437</td>
<td>0.70710557</td>
<td>0.70710678</td>
<td>0.70710678</td>
<td>0.00072331</td>
<td>1.1095E-06</td>
<td>2.6111E-12</td>
</tr>
<tr>
<td>2.1</td>
<td>0.68942809</td>
<td>0.69006468</td>
<td>0.69006556</td>
<td>0.69006556</td>
<td>0.00063747</td>
<td>8.306E-07</td>
<td>1.6951E-12</td>
</tr>
</tbody>
</table>

With a single iteration, only 8 multiplications and 3 additions are necessary for obtaining the normalizing factor.

The normalizing formula can also be used for other vector normalizing during the rendering process. When there is no basis for a Taylor approximation, the
initial approximation to the normalizing factor can be obtained by a table look-up.

3.7.5. "Cheap Phong" Shading.

"Cheap Phong" shading has the same computational complexity as Gouraud shading while retaining some of the "brilliance" of the Phong shading. This means that highlights are generated, but with a lower fidelity.

"Cheap Phong" shading means that the $N \cdot L$ dot product is linearly interpolated along polygon edges and along scan line from left to right edge. If the scene is limited to a single light source at the eye position, the intensity becomes a function of the dot product. The function depends on the viewed object's material.

With computer graphic systems with indirect colours, the look-up table is a possible place to realize even complex functions.

In the Moviebox system, the look-up table is divided into "slices", one for each material. The slices for material $k$ use table entries between $l_i$ and $l_{i+j-1}$. The dot product is mapped from the value $d_r \in [0.0, 1.0]$ into the integer value $d_i \in [i, i+j-1]$, before being interpolated across the polygon surface. In each look-up table entry, the intensity function is tabulated.

The problem with this approach is that, for shiny materials, the specular reflection component results in a very steep function. Thus contouring effects are likely to show near the highlights. The specular reflection as a function of the dot product is shown for two materials in the graphs below. It can be noted that the specular component is unnoticeable until the angle between the light normal and the surface normal is $\approx 40^\circ$ for a material with specular exponent $= 20$ (plastic), and only $= 12^\circ$ for a shiny material with specular exponent $= 200$. 
Fig. 3.12.
The specular light component as a function of the dot product $N \cdot L$ for a material with low shinyness (specular exponent = 20).

Fig. 3.13.
The specular light component as a function of the dot product $N \cdot L$ for a material with high shinyness (specular exponent = 200).

The coarse approximation to the intensity near highlights can also be noticed on colour plate 2, where the "cheap Phong" colour space is visualized.
3.7.6. Fast Phong Shading Computations

Bishop and Weimer [BISHOP86] have described a method for doing Phong shading cheaply. They express the normal vector as a linear interpolation of the true surface normals (of a triangle):

\[ \vec{N}(x, y) = \vec{A}x + \vec{B}y + \vec{C} \]

The diffuse light component depends on the dot product of the surface normal vector and the light vector (L):

\[ I_{\text{diffuse}}(x, y) \propto \frac{\vec{L} \cdot (\vec{A}x + \vec{B}y + \vec{C})}{|\vec{A}x + \vec{B}y + \vec{C}|} \]

This can be rewritten as:

\[ I_{\text{diffuse}}(x, y) = \frac{\vec{L} \cdot \vec{A}x + \vec{L} \cdot \vec{B}y + \vec{L} \cdot \vec{C}}{|\vec{L}| \cdot |\vec{A}x + \vec{B}y + \vec{C}|} \]

Performing the indicated dot products and expanding the vector magnitude gives:

\[ I_{\text{diffuse}}(x, y) = \frac{ax + by + c}{\sqrt{dx^2 + exy + fy^2 + gx + hy + i}} \]

Using forward differences, this formula can be evaluated for successive values of x and y with only 3 additions, 1 division and 1 square root per pixel, which according to Bishop and Weimer is too expensive for real time use.

(However, using the technique for computing \(1/\sqrt{c}\) described above, this is not necessarily true.)

Bishop and Weimer goes on, expanding to a Taylor series to the second degree:

\[ I_{\text{diffuse}}(x, y) = T_5 x^2 + T_4 xy + T_3 y^2 + T_2 x + T_1 y + T_0 \]
Again using forward differences, 2 additions are necessary per pixel.

Similarly, the specular reflection depends on the dot product of the halfvector (between the light vector and the eye vector) and the surface normal:

\[
I_{\text{specular}}(x, y) = \frac{\mathbf{N}(x, y) \cdot \mathbf{H}(x, y)}{||\mathbf{N}(x, y)|| \cdot ||\mathbf{H}(x, y)||}
\]

Using the same technique, this can also be computed with two additions per pixel. A table look-up will be necessary for doing the exponentiation.

Bishop and Weimer's technique is able to calculate pixels remarkably fast. However, computing the coefficients involves the computation of multiple dot products. The authors report a break even point between this algorithm and a traditional software implemented Phong algorithm for polygons of size ten pixels.

Performance ratios for a fairly complex scene (14000 polygons at 2048x2048 resolution) Gouraud: Fast Phong: Phong is reported as 1:2:10.

3.7.7. Torrance and Sparrow Light Model

The higher order cosine function used for simulating specular lighting effects in the Phong illumination model was chosen because it gave a reasonably good visual effect and was computationally simple. There is no underlying physics which advocates the use of this formula.

Based on the theoretical work of Torrance and Sparrow, Blinn [BLINN77] and Cook and Torrance [COOK81] have developed illumination models with a very high degree of realism. The Torrance-Sparrow model for reflection from rough surfaces is based on the principles of geometric optics. The reflecting surface is modelled as randomly oriented mirror-like microfacets.

The Blinn and Cook models are very similar, however, while Blinn simply supposed that the specular reflection has the colour of the incident light, Cook integrated the wavelength dependence of the specular reflectance coefficient into the model. For most materials, the colour of the specular highlight comes from the material colour for small incident angles. As the incident angle
approaches \( \pi/2 \), mirror-reflection will occur; all incident light is reflected, and the specular reflection approaches the colour of the light source.

In the Cook and Torrance model, the specular reflection is given as:

\[
rs = \frac{F \cdot D \cdot G}{\pi \cdot (n \cdot L) \cdot (n \cdot S)}
\]

where \( n \), \( L \), \( S \) are the surface unit normal, the unit light vector, and the unit eye vector, respectively. \( F \) is the Fresnel term which for a given material is a function of the angle of incidence and the wavelength of the light. \( D \) is the distribution function for microfacets in the surface and \( G \) is a geometric attenuation factor due to shadowing and masking between microfacets.

### 3.7.8. Global Illumination Models

Global illumination models were introduced in connection with ray tracing renderers [WHITTLED80]. The intensity reaching the observer consists of an ambient component (randomly scattered light), a diffuse component and a specular reflection component summed over all light sources, a specular reflection component (for reflecting other objects), and a transmission component (for transparent objects). The light model with exception of the reflected and refracted component is equal to the Phong light model (see symbol description above under Phong model):

\[
I = I_a \cdot k_a + ( \sum_{j=1}^{m} I_{lj} \cdot (k_d \cdot (n \cdot L_j) + k_s \cdot (n \cdot H_j)^n)) + \ k_s \cdot I_s + \ k_l \cdot I_l
\]

This light model can be extended to better simulate specular reflection by a rough surface using the formulas of Cook and Torrance.

While all other light models have incorporated some empirically derived "ambient" light component to account for all "unexpected" light reflections in the environment, the *radiosity method* has integrated this diffuse reflection into the light model [COHEN86]. The fundamental principle of the radiosity method is that of energy equilibrium:

*The energy reflected from a surface plus the energy transported through the surface must equal the energy that illuminates the surface.*
Thus, the radiosity method must first solve a set of linear equations, finding
the energy reaching each polygon. Then the hidden-surface process can use
this information for rendering this "pre-lighted" scene from different views.


Spatial aliasing problems are due to sampling an image containing fine details
at a too low resolution. The Nyquist Theorem states that the sampling
frequency must be at least twice the highest frequency of the original signal if
aliasing problems are to be avoided. Spatial aliasing is most noticeable on
object silhouettes (edge aliasing) and when rendering regular patterns (garden
fences, checkerboards etc.). If a spatial signal with frequency \( \omega_0 \) is sampled
with a sampling frequency \( \omega_1 \), an aliased spatial (low) frequency \( \omega_0 - \omega_1 \) will
appear in the sampled image.

During animations, the image is also subject to temporal aliasing, that is the
image changes faster than the temporal sampling. The latter effect is also
known from the movies (when wagon wheels seem to rotate backwards).

The most appropriate technique for avoiding aliasing would be to limit the
image to frequencies below the Nyquist frequency \( \omega_0 \). Sampling theory states
that the intensity function of the original image \( I(x, y) \) can be low-pass filtered
to a new function \( \Gamma(x, y) \) by a convolution:

\[
\Gamma(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(\alpha, \beta) \ H(x - \alpha, y - \beta) \ \text{d}\alpha \ \text{d}\beta
\]

with the sampling (sinc) filter \( H \):

\[
H(u) = \frac{\sin(\omega_0 u)}{\pi u}
\]

However, this optimal sampling filter function makes it necessary to evaluate
an infinite double integral for each pixel. Thus, we will have to use some
approximations which have limited spatial dimensions. Filter functions being
used in graphics are the box filter, the triangle filter (truncated Gaussian filters), and simplified versions of the sinc filter discussed above [BLINN89a/b]. Often, the filter is limited to an area not greater than 3x3 pixels.

Fig. 3.14. Various filter functions and their Fourier Transform (from [BLINN89b]).
Various filtering functions and their Fourier Transforms are shown in the figure above.

3.8.1. Supersampling

Spatial anti-aliasing can be carried out in two principally different ways. The normal way is to increase the sampling frequency (supersampling). Typically the image for a 512x512 screen is generated as a 1024x1024 image or a 2048x2048 image and then averaged down to the final resolution using some weighting function. Experience shows that the image must be computed to 4 times the final resolution for good anti-aliasing results. However, this gives a 16-fold increase in the execution time.

To reduce the execution time, anti-aliasing can be limited to the critical sections of the display. That is, inside a large surface, where intensity changes smoothly, there is no use for supersampling and anti-aliasing. Anti-aliasing is necessary where the image changes abruptly; at polygon edges, at sharp creases of a surface or at specular reflections. This "adaptive" supersampling is implemented in the so-called A-buffer algorithms where a pixel is subdivided into 4x4 or 4x8 subpixels [CARPENTER84] at critical parts of the scene.

3.8.2. Adaptive Supersampling

Another "adaptive" supersampling technique is implemented with many ray tracing algorithms [WHITTLED80] where rays are cast at the four corners of a pixel. If the computed intensity varies significantly, the pixel is recursively subdivided into four subpixels and new rays are cast at the new subpixel corners (see Fig. 3.7).

However, since all the above methods uses point-sampling, scenes can always be constructed with such repetitive details that aliasing frequencies can arise.
3.8.3. Anti-Aliasing by Area Integration

The other way of spatial anti-aliasing is to treat each pixel of the display as an area rather than as a point. Thus, for a polygon which intersects a pixel, the actual area of the polygon which is inside the pixel is computed. Catmull [CATMULL78] incorporated area-computations in a 2 1/2 D hidden surface algorithm. Catmull used a special clipping algorithm, clipping polygons to the pixel borders. His simple area computations amounts to filtering with a box filter, extending just to the pixel's border. However, the technique is too expensive to find extensive use. Area sampling has also been enhanced by using a Gaussian filter [FEIBUSH80].

Because the most objectionable aliasing seems to be due to the regular point-sampling rate, it might be better to sample the image at irregularly spaced points. The technique of stochastic sampling (both spatially and temporally) have been applied in connection with ray tracing renderers [COOK84, PURGATHOFER86].†

3.8.4. Separable Two-Dimensional Filters

Frank Crow [ROGERS85] has suggested the triangle filters shown on the figure below for filtering for two and four times reduction of resolution.

```
 1 2 3 4 3 2 1
 2 4 6 8 6 4 2
 3 6 9 12 9 6 3
 4 8 12 16 12 8 4
 3 6 9 12 9 6 3
 2 4 6 8 6 4 2
 1 2 3 4 3 2 1
```

Fig. 3.15.

3x3 "Gaussian" filter and 7x7 "Gaussian" filter for reducing resolution by two and four, respectively.

† Because most ray tracing algorithms do not rely on scan line coherence and incremental methods, they are easier to adapt to irregular sampling than other hidden surface algorithms.
One nice thing with these "Gaussian" two-dimensional filters is that they are separable into one-dimensional filters [NIBLACK85]. The ability to convolve the image with a one-dimensional n-element filter horizontally and then vertically, rather than having to perform a two dimensional convolution with a nxn filter, reduces the computations from $O(n^2)$ to $O(2n)$.

\[
\begin{array}{cccccc}
1 & 2 & 1 & 1 & 2 & 3 & 4 & 3 & 2 & 1 \\
\end{array}
\]

Fig. 3.16.
One-dimensional components of the 3x3 and 7x7 filters above.

Triangle filters can be efficiently implemented by observing the fact that the weighting coefficient applying to a cell in the display is either increased by 1 (on the right edge of the triangle) or decreased by 1 (on the left edge) as the filter is moved one cell to the right. Thus, for implementing an $n$-element filter, $n$ additions / subtractions are necessary plus one bit-shift or division by the sum of the coefficients.

3.8.5. Anti-Aliasing in Non-RGB Colour Spaces

Most implementations of anti-aliasing renderers use simple supersampling and filtering techniques [RIESENFELD84, CHRISTIANSEN87] in regular RGB colour spaces. In these, anti-aliasing is accomplished by linear interpolation of each colour component.

In the Moviebox system, supersampling by a factor of two is possible, and resolution is reduced back to normal by employing either a simple 2x2 box filter or the 3x3 "Gaussian" shown above. This applies only to the "RGB mode" of Moviebox, that is when Moviebox generates images with 8 bits of red, green, and blue. For workstations with fewer colours available, the 256x256x256 colour space is normally mapped into a regular colour space with a coarser quantization (8x8x4 or 6x6x6 for an 8 bits indexed display) with a dithering technique (see Ch. 3.10).

However, the default working mode of Moviebox is "indexed mode", that is when Moviebox partitions the available set of colour indices into "slices" which are used for shading each type of material. The image is then generated
directly for this colour space. This gives a much smoother appearance than
RGB images dithered to a coarse, regular colour space.

The colour space used in "indexed mode", is illustrated on colour plates 1, 2,
and 3. Colour plate 1 shows the palette on a 256 colour system, when set up
for 8x8x4 colour space (RGB mode). Each colour is shown in the RGB
coordinate system, showing the quantization. Colour plate 2 shows the
topology of the colour space when rendering the "robot" in Colour plate 3 in
"indexed mode" on the same 256 colour system. Ramps for four different
materials (red, gray, green) are shown. Because the material colours are
linearly indexed with the light-normal dot product, specular reflection causes
longer distance between colours towards the end of each ramp. The green and
blue materials have been specified with plastic highlights, causing the green
and blue material ramps to curve towards white. Colour plate 3 shows the
actual display of a "Cheap Phong" shaded robot.

However, the shape of the "indexed mode" colour space makes anti-aliasing
much more complex. Turkowski discusses the problem of anti-aliasing in
colour spaces of different topology [TURKOWSKI186]. Though some
comments apply to shaded imagery, the suggested colour space topologies,
for instance his Major-minor colour space, are primarily suited for anti-
aliasing of constant colour drawings.

The optimal colour space for shaded images would assign most colours to the
primary material ramps, and fewer colours to allow for anti-aliasing of the
object edges against the background, and yet fewer colours to allow anti-
aliasing where one object silhouette overlaps another object. To my
knowledge, there exists no good implementations of shaded imagery with
anti-aliasing for indexed colour displays.

Colour plate 4 shows a closeup of the robot rendered in RGB mode and
dithered to 8x8x4 colour levels. Colour plate 5 shows the same object after
being supersampled with double resolution and filtered by a 3x3 "Gaussian"
filter. Also this image was dithered to 8x8x4 levels in order to display on the
workstation frame buffer. Though the dithering will camouflage this, the anti-
aliasing effect is particularly noticeable on the near-vertical edges (see blue
part).
3.9. Texture Mapping, Transparency, Reflectance

Texturing is mainly a technique for efficiently handling picture detail and increasing the apparent scene complexity without swamping the general geometry handling of a rendering system. What is represented as texture and what is given a general geometric representation really depends on the scale of display.

In computer graphics, the term texture mapping is used both for applying a repeated pattern (brick, gravel, grass) to a surface and for the mapping of a non-repeating two-dimensional image to an object surface (carpet, painting).

Textured objects, rich in high frequencies, are particularly susceptible to aliasing problems. Thus, a texture mapping program will have to contain some filtering method.

Textures are often digitized photographic images, but can also be manually drawn images, or fully synthesized images (environment maps). Angerhofer describes how stochastic texture maps can be generated, either by directly specifying the power spectrum, or from pictures of the material [ANGERHOFER85]. One of the nice features of Angerhofer's textures are that they are periodic, which means that they can be used to tile a surface without discontinuities at tile borders.

3.9.1. Texture Model

Texture is most frequently represented as a two-dimensional array of colour values. This array is somehow related to a graphic object being rendered. For instance, each polygon vertex can contain (u, v) coordinates relating to the texture map.

Sometimes, it is hard to specify the mapping for each vertex to the texture space. In this case a two-phase technique can be useful [BIER86]. The object is surrounded by an intermediate object of simple geometric shape (box, sphere, cylinder). The texture is logically mapped onto the intermediate object (phase 1), then the texture is projected from the intermediate object onto the actual object (phase 2).
Two-phase texture mapping has been added to Moviebox in a trial implementation, using simple point sampling (no filtering) [BIRKELAND88]. The implementation was done without considerations for an efficient implementation, and showed a performance ratio compared with Gouraud shaded polygons of approximately 5:1.

Instead of just "wall-papering" surfaces, textures can also be used to model three-dimensional "substance". This is very convincing when rendering objects made of strongly textured material such as wood or marble [PEACHEY85, BREMER86]. A procedural model is often used for representation instead of a large three-dimensional texture array.

3.9.2. Bump Mapping

Though texture mapping normally applies to the colour of a surface, it is possible to let a texture map attenuate surface normals instead. Blinn [BLINN78] uses this technique for simulating "wrinkled" surfaces (doughnuts).

The ability to represent transparency along with the texture map also gives us the possibility of representing apparently complex objects such as the leaves of a tree as texture.

3.9.3. Environment Mapping

When objects in the scene are shiny, other objects in the scene should be visible in these reflections. This is implicit in the normal workings of a ray tracing renderer. Such reflections are present when rendering buildings with reflections in the glass windows, or when trying to give an impression of a polished car in city surroundings.

Modelling everything in the surroundings as objects would be overly expensive. Instead, we project an approximation of "the rest of the world" onto a sphere or box and store the projection as a texture map. When rendering, this map is sampled in the reflected direction [BLINN76].
The environment mapping can be divided into two parts: a static map (of the sky or other imagery outside the geometry model) and a dynamic map which contains objects in the scene and which are dependent on the current choice of light sources.

3.9.4. Contour Mapping

Rather than enhancing the realism of the scene, texture mapping can also be used for exploiting properties of the scene not visible in a natural world. Properties of the geometric objects like stress, temperature, velocity etc. can be used as indices to colour maps (one- two- or three-dimensional).

In MOVIE-BYU and Moviebox, this technique is used for presenting an optional scalar property (specified as a floating point variable at each vertex).

An example of mapping FEM-analysis data onto the surface is shown on colour plate 10.

3.9.5. Fast Texture Mapping

For our purpose, scene objects consist of planar polygons, and the relation to a texture is described as a pair of indices \((u, v)\) at each vertex. Texture mapping then consists of two subproblems:

- Mapping each pixel area onto the texture map.
- Filtering the texture to avoid aliasing.

A naive method for geometry mapping is to do linear interpolation of texture indices across the polygon surface. However, when general perspective projection is in use, the linear interpolation will not give the appropriate foreshortening of the texture. We must rather apply the full perspective projection. The perspective mapping of a planar texture can be expressed by the homogeneous matrix notation [HECKBERT86]:

---

\[\text{The linear interpolation in screen space used for Gouraud and Phong shading both fail to take the nonlinear foreshortening into account. However, for most scenes this effect is not noticeable.}\]
\[ [xw \ yw \ w] = [u \ v \ 1] \begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix} \]

The inverse of this mapping is of the same form:

\[ [uq \ vq \ q] = [x \ y \ 1] \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \]

, or in terms of the original coefficients:

\[ [uq \ vq \ q] = [x \ y \ 1] \begin{bmatrix} EI-FH & FG-DI & DH-EG \\ CH-BI & AI-CG & BG-AH \\ BF-CE & CD-AF & AE-BD \end{bmatrix} \]

Moreover, the composition of two perspective mappings is also a perspective mapping. The latter is interesting if we would like to compute a mapping from a photograph onto a polygon shown in perspective. For instance, it can be useful to map photographs of building facades onto stylized building bodies.

Since the 3x3 matrix above relates homogeneous coordinate points, all scalar multiples of the matrix are equivalent, which leaves eight degrees of freedom. Thus, to compute the 3x3 matrix, the mappings of four linearly independent points in the plane must be known.

If high-frequency textures are point sampled using the mappings above, problems occur. When polygons spanning much of the scene depth is textured, the texture can be greatly expanded near the observer (fine sampling) while greatly condensed towards the rear (coarse sampling). To avoid aliasing problems, filtering is necessary. However, because the filtering requirements vary across a surface, uniform super-sampling as often used for simple anti-aliasing may prove either insufficient or inefficient, depending on the sampling rate.

In principle, each pixel area should be mapped onto texture space, and the part of the texture inside the pixel area should be weighted by a filtering function to compute the output pixel colour [BLINN76, FEIBUSH80, HECKBERT86]. As the mapped image of a pixel can span a lot of cells in the texture array, the direct filtering can prove to be very time-consuming.
To speed up the texture filtering, the texture map can be prefiltered into pyramid data structures at different resolutions. Williams proposes a tri-linear interpolation scheme; two adjacent pyramid levels are chosen depending on the largest extent of the pixel in texture space, bi-linear interpolation is performed on both levels of the pyramid, and a linear interpolation is performed between these values [WILLIAMS83]. As this method uses the largest pixel extent in texture space to select the pyramid level, excessive blurring can occur when pixel extent in texture space is non-square. The summed-area table technique suggested by Crow improves on the latter and allows better filtering than the pyramid, but at significantly higher memory cost [CROW84].

The computational cost of Williams' prefiltering method is eight pixel accesses and seven multiplies per screen pixel. The memory cost for pyramidal storage is 4/3 that of the original texture map.

3.9.6. Transparency and Reflectance

The full implementation of transparency and reflectance effects is only possible in a global illumination environment, like in a ray tracing renderer. However, reasonable implementations of both effects are also possible in scan line renderers by using environment maps.

Transparency effects, but not proper refraction, can also easily be handled inside scan line and z-buffer renderers. For scan line z-buffer algorithms, transparent objects can be saved on a list and applied to the intensity buffer after all oblique polygons have been output. The technique requires a separate intensity buffer and a transparency coefficient buffer in addition to the z-buffer and intensity buffer [ROGERS85].

In a restricted colour environment, like an 8 plane frame buffer, the attenuation of the colours by transparency coefficients is a problem. A simple solution to this, also used in the GAS system, is to implement transparent surfaces using transparency texture maps. Thus for a 50% transparent surface, 50% of the cells are just ignored by the z-buffer algorithm, leaving holes for showing the innards.
3.10. **Adaption to a Display**

Computer Graphics is all about cheating and getting away with it. By knowing the limitations of human vision, the chance of not being caught is better.

3.10.1. **Physiological Limitations**

The human eye has its limitations. As the human perception of the image is our final goal, we are interested in making an image quality that matches the capabilities of the human visual system, but does not surpass it.

The eye has a limited achromatic (black/white) spatial resolution determined by the eye's rods on the retina and an even more limited chromatic resolution due to the density of cones [LEVINE81].

![Graph showing chromatic and achromatic response](image)

Fig. 3.17.
a) Chromatic and b) achromatic spatial frequency response measurements of human visual system [PRATT78].

It can be noted that the achromatic response curve has a band-pass characteristic with highest response in the range 3 cycles/degree to 10 cycles/degree, whereas the chromatic response curve has a low-pass characteristic up to 2 cycles/degree. Related to a graphic monitor viewed from a normal viewing position (50 cm), the low-pass roll-off frequencies correspond to 23 dots/cm\(^\dagger\) (58 dots/inch) and 4.5 dots/cm (12 dots/inch) for luminance and colour vision, respectively.

\(^\dagger\) The unit dots/cm corresponds to 0.5 lines/cm (as used in optics).
Fig. 3.18.
Response characteristics of the different cones in the human eye [PRATT78]. The highest sensitivity is in the green area (=530 nm), with slightly lower sensitivity in the red area (=650 nm), and with very low sensitivity in the blue area (=425 nm).

The frequency dependency of the visual system is difficult to take advantage of with traditional symmetric RGB-systems. However, it is possible to transform the RGB system into another colour system to make use of this limitation. For example, the television industry takes advantage of the limited colour vision capabilities in their NTSC encoding of colour TV signals [CONRAC80]. Likewise, Stockham [STOCKHAM72] suggests the exploitation of the human visual model for image analysis and synthesis. In NTSC colour transmission, the colour image is represented by a luminance signal Y, and the two chrominance signals I, and Q. The formulas for computing the YIQ components from the RGB colour coordinates are as follows:

\[
Y = 0.299 \, R + 0.587 \, G + 0.114 \, B \\
I = 0.596 \, R - 0.274 \, G - 0.322 \, B \\
Q = 0.211 \, R + 0.523 \, G + 0.312 \, B
\]

It can be noted that the luminance signal Y, is computed from the RGB components with coefficients which weight the components according to the eyes spectral sensitivity curves on Fig.3.18.

For transmission, the luminance signal is band limited to 4 MHz before modulation, while I (cyan-orange colour axis) and Q (magenta-green colour axis) chrominance signals are limited to 1.3 MHz and 0.5 MHz, respectively. The maximum horizontal resolution reproducible on a TV
monitor is thus limited to 320 lines (Y channel), 120 lines (I channel), and 40 lines (Q channel) [JOHNSTON86]. At the 50 cm viewing distance, a 30 cm wide monitor matches the eye's frequency limitations.

Other research indicate that the eye is capable of distinguishing between 32 and 64 different graylevels [FOLEY82]. In colour vision experiments indicate that the human eye can distinguish 128 different hues and between 16 and 23 saturation levels [FOLEY82].

3.10.2. Direct Colour Frame Buffers

So-called direct colour frame buffers have separate cells for each primary colour (RGB). For good quality, at least 6-8 bits per primary are needed. Direct colour frame buffers create a large colour space, which simplifies rendering operations that require some blending of different colour; transparency, anti-aliasing etc.

Most shading algorithms assume that output colour quantization is linear. However, the intensity displayed on a monitor is a power of the voltage supplied to the electron gun:

\[ I = k \times V^\gamma \]

where \( \gamma \) is typically in the range 2.3-2.7.

The rendering system must compensate for this non-linearity, by applying the inverse function; gamma correction:

\[ V = (I/k)^{1/\gamma}. \]

The gamma correction can either be applied in the colour look-up table of the display, or better, before final quantization in the rendering algorithm.

3.10.3. Monochrome Frame Buffers

In order to present images specified with RGB colours on a monochrome monitor, the coefficients used for generating the luminance signal (Y) in the
NTSC colour television encoding can be used. As mentioned above, these are related to the eye's spectral response.

3.10.4. Frame Buffers with Indexed Colour

The problem with rendering on lower-priced workstations is to map the colours output from the renderer into the 256 or 16 colours available in the workstation. One strategy is to let each differently coloured object have a slice of the available colour table, using a reduced light model as described in Ch. 3.7.5. Another strategy, more expensive and often with inferior result, is to let the renderer generate images for a virtual direct colour frame buffer, and then simulate this RGB device by setting the look-up table to implement the Cartesian product of the primaries. For eight bit frame buffers, it is customary to assign 8, 8 and 4, or 6, 6 and 6 levels to the RGB primaries.

The basic strategy of dithering is to trade intensity resolution for spatial resolution. With high resolution monitors, the eye will average colours of neighbouring pixels, thus gaining the impression of colours not even represented in the frame buffer. This effect is well-known from half-tone colour printing, where four colours; yellow, cyan, magenta, and black are used for reproducing general colour images. As discussed above, the eye will effectively low-pass filter chromatic information with a roll-off frequency of 2 cycles/degree.

3.10.5. Uniform Quantization and Ordered Dithering

When an image is quantized and the distance between levels is d, the technique called ordered dithering will add a random noise signal with amplitude equal to d/2 before doing the quantization. This will make a slowly changing signal inside the interval to be quantized partly to the level below, partly to the level above, giving an impression of an in-between intensity level, and also avoiding contouring effects. Normally, the random noise signal is approximated with a "pseudo-random" dithering matrix which tiles the whole screen.
An nxn dithering matrix, where n is a power of 2, can be generated using the recursion relation [JARVIS76]:

\[
D_n = \left[ \begin{array}{cc} 4D_{n/2} & 4D_{n/2} + 2U_{n/2} \\ 4D_{n/2} + 3U_{n/2} & 4D_{n/2} + U_{n/2} \end{array} \right]
\]

for \( n \geq 4 \), where \( U_n \) is the \( n \times n \) matrix with all elements = 1.

\[
D_2 = \left[ \begin{array}{cc} 0 & 2 \\ 3 & 1 \end{array} \right]
\]

The nxn dithering matrix \( D_n \) generates \( n^2 \) intensity levels.

In contrast to the Floyd Steinberg dithering technique presented below, the ordered dithering must be used with equally spaced output colours.

### 3.10.6. Tapered Quantization and Error Dithering

With regular quantization, some of the available output colours will tend to be used a lot while others may not be used at all. In image processing literature tapered quantization is given much treatment, but mostly for achromatic image representation [ROSENFELD76].

Heckbert describes algorithms for finding a colour image quantization [HECKBERT82]. Heckbert uses a simple error metric; the square Euclidean distance in RGB space:

\[
D = \sum_{i,j} d(c_{i,j}, q(c_{i,j}))
\]

where \( d(x,y) \) measures the distortion between the original cell value \( x \) and the quantized value \( y \):

\[
d(x,y) = (x_r - y_r)^2 + (x_g - y_g)^2 + (x_b - y_b)^2
\]

Heckbert argues that a better error metric would be based on a perceptual measure, for instance a colour metric based on YIQ colour space.
To determine the colour distribution, Heckbert pre-quantizes the image to 15 bits, 5 bits red, 5 bits green, 5 bits blue to make a frequency histogram. The principle of the colour selection is to have each colour represent an equal number of cells in the original image. To select the colour representatives, a "median cut" algorithm is proposed, which recursively divides the colour space into "boxes". A box is split at the median point along the longest dimension of the box. After "k" boxes have been generated, we choose a representative for each box by averaging the colours in the box. This will generate an "inverse colour look-up table", where colours represented by RGB (5 bits each) can be converted to one of "k" colour indices.

Heckbert reports that a 512x486 x 24 bits image can be quantized to 256 levels in under one minute on a VAX 11/780.

Wan, Wong, and Prusinkiewicz have recently proposed a divisive algorithm for multidimensional data clustering which produces smaller quantization errors than the "median-cut" algorithm proposed by Heckbert [WAN88].

Floyd-Steinberg dithering [NEWMAN79] (also called error-dithering) is effective with tapered colour quantization. Their algorithm compensates for the quantization error introduced at each pixel by propagating it to the neighbour cells to the right and under. This scheme makes it possible to do the dithering in a single pass. If $c_{i,j}$ is the original colour in cell i,j and the nearest representation index is found by the function $k = p(x)$, which represents the colour $y_k$, the Floyd-Steinberg algorithm distributes the error $\varepsilon$ as follows:

$k = p(c_{i,j});$ /* Find nearest repr. */
$f_{i,j} = k;$ /* Store in frame buffer */
$\varepsilon = c_{i,j} - y_k;$ /* Quantization error */
$c_{i,j+1} = c_{i,j+1} + \varepsilon * 3/8;$ /* To right neighbour */
$c_{i+1,j} = c_{i+1,j} + \varepsilon * 3/8;$ /* To below neighbour */
$c_{i+1,j+1} = c_{i+1,j+1} + \varepsilon * 2/8;$ /* To right below nb.*/

The Floyd-Steinberg algorithm is capable of generating nice images even when having very few colours available. However, for three-dimensional images consisting of a shaded object on a constant colour background, the algorithm tends to produce "false contours" of the object silhouette edges.
Though mathematical methods are used to select optimal quantization colours minimizing the error metric, this does not guarantee that this gives the most pleasing visual impression on the human eye. While the error metric will try to minimize the error over the shaded surfaces of the image, the eye seems to be most sensitive to the edges.

When selecting the quantization colours, one would like to guarantee that all colours in the original image can be represented. This is only true when all the original colours lie inside the convex hull of the representative colours.

3.11. Survey Conclusions

This section tries to conclude what sort of algorithms are feasible for implementing an "interactive" software-based rendering system on a workstation.

The models to support should at least include polygon meshes. NURBS, for instance, could be rendered directly or approximated by polygon meshes. CSG is a very user-friendly approach to building complex geometry, and it would be useful at least to provide some support for the primitives.

Viewing operations and transformations can be supported without any worries about the computational cost.

Extent testing and clipping software should provide the possibility to efficiently scan complex models.

The best candidate for a hidden surface algorithm is the scan line Z-buffer algorithm. It is efficient both with small and large models and is flexible enough to support sophisticated rendering.

Though global illumination models can be used to generate images with thrilling realism and lighting effects, they are currently too computationally expensive for interactive use with simple workstations. So, the light and shading models to consider are primarily Gouraud and Phong.
Some support for environment mapping (and other sorts of texture mapping) is interesting with the scan line renderer to give effects of reflections etc. without using ray tracing. This should be within the computational limits.

There should be provisions for anti-aliasing; at least two times (but better: four times) supersampling with a "Gaussian" filter. A computationally even better solution would be to implement an A-buffer algorithm.

There should be efficient dithering software available to support workstations with few colour planes and colour plotters (with only on/off colour component control).
4. GAS - AN EXPERIENCE WITH A LARGE OBJECT ORIENTED RENDERING SYSTEM.

"What is object oriented programming? My guess is that object oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favour of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is."

Tim Rentsch [RENTSCH82].

GAS ("Graphics for All Seasons") is a collection of computer graphics algorithms represented in an object oriented framework. GAS was developed as a rendering toolkit and as an experiment with object oriented programming for computer graphics rendering. It is also the framework used for experimenting with different rendering algorithms. Some of these are based on so-called raster operations, and are described in a separate volume [ZACHRISEN89b].

I have found it to be too consuming of time and space to give a thorough description of the GAS classes. Instead I will just give a brief overview and a discussion of some of the experience with an object oriented rendering system. Again, I will focus somewhat on efficiency.

The world of three-dimensional rendering is a diverse world, containing various different model representations along with a large set of different rendering techniques. For advanced rendering, it is often desired to have many different types of geometry contributing to the final image, using a variety of rendering techniques for different effects.

Object oriented programming has proven to be effective for building user interfaces [LIPKIE82] but also as a conceptual basis for whole computer systems [GOLDBERG83]. Three-dimensional rendering systems have
hitherto been carried out mostly in procedurally oriented languages such as Fortran and C. Haugen and Skjfell report on Class Graphics, a basic three-dimensional graphic package for SIMULA [HAUGEN83], much patterned after the early "Core"-proposal [GPSP78]. In a later article, Grant, Amburn and Whitted report on building a rendering environment in C++ [GRANT86]. Wisskirchen discusses how PHIGS should be respecified to be effective in an object oriented environment [WISSKIRCHEN85].

For fast three-dimensional rendering engines, speed is essential and interpretive languages such as Smalltalk will be at a disadvantage. The SIMULA language was not available on the actual workstations. The natural choice for this experiment when it was started in 1984 was a new object oriented dialect of "C", ("C with Classes") invented by Bjarne Stroustrup [STROUSTRUP83, STROUSTRUP84]. The new language was a strict superset of the original "C" language [KERNIGHAM78] with the class concept borrowed from SIMULA [DAHL70]. The language later evolved into what is known as "C++" [STROUSTRUP86].

4.1. Components of Object Oriented Programming

Before going into detail about how C++ was used in the GAS system, I shall give a short description of some of the basic concepts of object oriented programming.

4.1.1. Objects.

Objects are a combination of the conventional programming notions of data and procedures. Unlike traditional programming languages, which leave it to the programmer to sort out the interaction between data and the procedures operating on that data, object oriented programming languages combine data and the procedures to go with that data into objects.

Following the principles of abstract data types [LISKOV77], object data representation and how procedures accomplish their tasks are hidden from the

† Though the early implementation had a slightly different syntax and missed some of the features of C++, I will in this chapter refer to the language as C++. 
view of the outside world. The only interaction with an object from the outside world is a reference or handle to the object and some specific methods or procedure declarations that are explicitly exported to the outside world.

4.1.2. Classes.

Objects with the same data and operations on the data are said to belong to the same class or object type. They are described once and for all by describing the class. Objects come into existence by creating an instance of the class; this allocates storage space for the object and initializes the internal variables.

4.1.3. Methods.

Classes define procedures that "know" how to operate on the instance variables. These procedures are called the object's methods. The methods that are exported to the outside world can be accessed as method calls to the object (SIMULA and C++) or by specifying the method in a message to the object (Smalltalk and Objective C).

4.1.4. Inheritance.

One of the most powerful concepts in object oriented programming is the possibility to let one object class (subclass) inherit the properties of a parent class (superclass). Subclasses and superclasses give us the ability to model data abstractions and specializations [TSCHRITZIS82]. This is the key to rationalize coding. When we have two object classes with large similarities but some differences, we create an abstraction (a superclass) with the common properties of the two classes, and then we create the two specializations (subclasses) where only the behaviour that differs between the object classes is described.

Most object oriented programming languages implement single inheritance. This means that a class can inherit the properties of only one class, providing a simple hierarchic class structure. A few object oriented programming languages support inheritance from several different classes (later Smalltalk versions, Object Logo, ExperCommon LISP). This is called multiple inheritance. Multiple inheritance is a more powerful concept than single
inheritance. For some data modelling it is of great value. However, multiple inheritance makes the language syntax and semantics more complex, and the language implementation very much more complex.

A subclass normally adds variables and methods to its superclass. However, a subclass also has the possibility to override the behaviour of an inherited method. Moreover, a superclass can define methods that are common for all its subclasses, but leave the description of the methods to the subclasses. This lets the superclass make use of this property of the object in its own methods, making it possible to move more of the object implementation to the common superclass. The latter mechanism is often referred to as virtual methods or procedures (SIMULA, C++).

4.1.5. Metaclasses and Factories

In some of the object oriented languages, a class is just a user-defined data type describing the common properties of all objects belonging to the class. A class will normally have an implicit method (new) for creating instances. An additional aspect is that classes are not just data types but rather special objects that respond to method calls or messages. (In Objective C [COX83] classes are referred to as factory objects.) Methods belonging to the class are referred to as class methods (in contrast to the instance methods discussed above). In addition to the mandatory new method, a class object can naturally implement methods for creating its instance objects in alternate ways. (Create circle arc objects by specifying centre, radius, and angle, or by specifying three points in the plane.)

When a class is also an object, it can of course also carry its own class variables. Moreover, class objects can naturally be organized into "classclasses" or metaclasses. While languages such as SIMULA 67 and C++ only support simple instance objects, Smalltalk is an example of a language taking the object oriented approach to a great length supporting instance objects, class objects and even metaclass objects.

4.2. GAS Infrastructure

The GAS system was designed to give some of the flexibility advantages of the Smalltalk system, while retaining the efficiency of the compiled C++ and
SIMULA languages. Thus, for every instance class in GAS there is also a metaclass (or classclass). The metaclass is normally instanciated only once into one "factory" object, which among other properties, can generate the instances belonging to the associated instance class.

Sometimes, when instances are frequently generated, the metaclass can provide a special object pool for the instance objects, pre-allocating objects from the heap in bigger chunks.

In GAS, the metaclass structure keeps the whole class hierarchy (and names) available to the runtime environment. At least this is desirable for debugging etc.

Deciding whether to make an attribute part of the instance or part of the class object, is a question of flexibility or economy. By allowing several instances of the "factory", instances differing only on their methods can be handled as a single object class but grouped under different factory objects. Virtual methods, that is, methods declared on generic objects, but defined for each specific subclass, are handled as procedures belonging to the "factory" object. This basic structure makes each instance relatively small while retaining flexibility. An example of an object-oriented system with this dual class structure is Smalltalk-80, where much of the same infrastructure is provided by pre-programmed classes.

The super-class of any other class in the system is class Object, which defines attributes and operations inherited by all other classes in the system.
**** Instance class declaration ****/
class Object {
    friend C1Object;

    /*********** Variables ***********/
class C1Object *obj_classof; /* Pointer to class object */

    /*********** Methods ***********/
public:
    class C1Object *classOf(); /* Get class object */
    Bool isKindOf(class C1Object *);
        /* Test if belong to subclass */
    Bool isA(class C1Object *);
        /* Test if belong to class */
    class Object *newCopy(); /* Make copy of object */
    Void printObj(); /* Print object contents */
    Void deleteObj(); /* Delete object */
    int objSize(); /* Size of object in bytes */
};

/***** Meta class declaration *****/
class C1Object : public Object { /* NOTE: subclass of Object */
    friend Object;

    /*********** Variables ***********/
    char cobj_className[12]; /* Name of class */
    int cobj_sizeofinstance; /* Size of instance (bytes) */
    class C1Object *cobj_superclass; /* Superclass pointer */

    /***** Virtual methods *****
    class Object *(*o_create)(); /* Virtual creation method */
    Void (*o_delete)(); /* Virtual deletion method */
    Void (*o_copy)(); /* Virtual copying method */
    Void (*o_print)(); /* Virtual printing method */

    /*********** Methods ***********/
public:
    class Object *newObj(); /* Instanciate object */
    char *className(); /* Return name of class */
    class C1Object *superClass(); /* Returns superclass */
};

Fig. 4.1.

Declaration of Object class (and metaclass).
74 Volume 1 - Survey and Integration

/***/ Instance class declaration ***/
class Link : Object { /* Subclass of Object */
    friend C1Link;

    /********** Variables **********/
    class Link *li_succ; /* Pointer to successor */
    class Link *li_pred; /* Pointer to predecessor */

    /********** Methods **********/
    public:
    class Link *prev(); /* Get predecessor */
    class Link *succ(); /* Get successor */
};

/***** Meta class declaration *****
class C1Link : public C1Object { /* NOTE: subclass of C1Object */
    friend Object;

    /********** Methods **********/
    public:
    class Link *newLink(); /* Instanciate object */
};

Fig. 4.2.
Declaration of Link class (and metaclass) which is a subclass of Object.

The declarations above declare a class "Link" for doubly linked objects similar to the SIMSET class of SIMULA 67 [DAHL70]. Note how the Object class is the superclass of all instance classes and metaclasses. Note also how the methods (procedures) are explicitly exported to public view, while all other parts of an object are only accessible from the class methods (and the metaclass methods due to the friend declaration).
4.3. Efficient Representation of Objects vs. Flexibility

As can be seen from the example above, GAS is very economic with the memory space requirements for instances. A basic instance of class Object (which is more interesting as part of subclasses) occupies only four bytes. In contrast, the basic object (core) of the X Toolkit [McCORMACK88], which is also aimed at handling high numbers of instances, amounts to 110 bytes.
shown in the diagram below. Totally, GAS has implemented approximately 90 classes.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GeomEngine</td>
<td>a)</td>
<td></td>
</tr>
<tr>
<td>GraphDev</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RastDev</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chromatics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IconRaster</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Link</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GElemnt</td>
<td>b)</td>
<td></td>
</tr>
<tr>
<td>GInstance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GComposite</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ObjTree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PolyTree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VolumeObj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConvexVol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cube</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cylinder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sphere</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wedge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PolygonVol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HalfSpace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BooleanVol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PointSet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polyline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polygon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point4</td>
<td>c)</td>
<td></td>
</tr>
<tr>
<td>PointNorm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Light</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transformation</td>
<td>d)</td>
<td></td>
</tr>
<tr>
<td>Viewing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4.4.

Excerpt of the class hierarchy of GAS showing a) device drivers, b) geometry objects, c) point objects, and d) transformation/viewing objects.

Normally, what is modelled as an object in GAS has its more or less concrete counterpart, like the geometric objects. However, some other objects are more on the borderline between physical objects and processes. Sometimes it is difficult to decide which methods belong to which objects.

*Is the graphic object rendered if passed to the graphic device object, or is the ability to be rendered part of each geometric object?*
In the Smalltalk-80 system, most objects represent more or less physical objects like a graphic object, a file, or a collection. However, there is also an object for the "BitBlt", or raster operation, which typically represents a process. The reason for representing this as an object and not a method in another object, is that "BitBlt" operations have a lot of options and can thus be handled more easily as an object with its own state and default values.

In GAS there are classes for some of the more complex processes (like renderers).

4.4.1. The Geometry Classes

As described in the previous chapter, the geometric models used in three-dimensional computer graphics have become so diverse that they are hard to integrate into a simple framework. However, the ability to model in terms of hierarchically structured classes certainly helps. The GAS geometry class hierarchy handles simple three-dimensional polygons and lines along with solid objects.

The Object class provides the basic functionality of all objects; copying, class queries and simple printing.

The Link subclass adds the ability to chain geometric objects together so that more complex (structured display files) structures can be built.

The GElement subclass contains common methods for all graphic elements. Typically, the extent can be queried, and a virtual display method is declared.

The VolumeObj subclass adds the abstraction of volume objects (in contrast to sheet objects), and contains facilities for managing material properties.

The ConvexVol subclass contains methods that are limited to convex solid objects. For instance, the special CSG display algorithm is contained in this class definition.

Then there are several different subclasses of ConvexVol; Cone, Cube, Cylinder etc. which each contains a method for expanding their geometry into
a mesh of polygons for rendering. Alternatively, they could provide a method for outputting depth coordinates and shade directly into the output buffers.

*BoolVol* is another subclass of *VolumeObj* which contains a volume operator along with references to a left and a right subtree.

*GInstance* is a subclass of *GElem*ent which makes it possible to make an instance of an object by applying an object transformation matrix to a graphic object.

By means of *Ginstance*, *BoolVol* and *ConvexVol* objects, complex CSG objects can be built. Solid objects, sheet objects (polygons) and lines can be integrated in object trees (*ObjTree*). When an object tree is rendered, the tree is recursively traversed, and appropriate rendering algorithms are invoked for each class of object contained in the tree.

### 4.4.2. The Device Model

The GAS device is modelled after high-performance devices with support for three-dimensional geometry. The major objective was that simple devices should be able to use software for display while more advanced devices should be able to utilize their expensive hardware matrix multipliers etc. Thus, all devices are subclasses of *GeomEngine*. This display class provides all facilities for handling three-dimensional geometry; viewing, transformation and clipping.

The *GraphDev* device class provides display facilities which are common for all types of displays; line drawings etc., while the *RastDev* subclass provides the facilities which are only available with modern raster graphic devices; shaded images etc. Each specific raster device driver is then modelled as a subclass of *RastDev* where the general software methods of the geometry engine and the raster device can be replaced with links to hardware implemented algorithms.
4.5. Experience with C++ and Object Oriented Methodology

The major point in using an object oriented programming language such as C++ for writing renderers, is that the language provides aid for structuring the programs and encourages reuse of code.

Structuring the problem into generic classes and specific subclasses, reduces the amount of code to be written. In a toolkit such as GAS, there is a menu of different objects and methods to choose from. However, many objects have attributes in common which can efficiently be generalized to a parent class.

The implementation has some good sides and some bad sides:

- The generated code is efficient, and the programmer has reasonable control over the introduced runtime operations.

- Compared to normal C (prior to ANSI C), C++'s declaration of exported methods (functions) with compile time checking of argument types is very convenient. However, nothing comes for free: the compile time of C++ is more than double that of plain C.

- In a UNIX environment it is convenient that C++ modules are compatible with C modules and C utilities.

- C++ builds new global names for object methods (functions), based on the class name and the method's name. The generated names depend on the rest of the object variables and methods, and will thus change whenever a new internal variable is added to the class. This means that all other modules using external methods to this object must also be recompiled. Describing the proper dependencies to the UNIX "make" utility will take care of this; however, regeneration of the system can cause substantial recompilation. This external effect of internal changes to the class is contradictory to the basic object oriented principle of encapsulation. The synthesized names can also be

† Though this chapter refers to C++, the specific comments are relevant for the early test version called "C with Classes" and are not necessarily relevant for later implementations of C++. 
hard to guess when debugging programs with a symbolic debugger.

- Encapsulation is otherwise one of the strong points of C++, with good control over the external interface to other classes as well as through inheritance [SNYDER86].

- The syntax of in-line methods and other methods are different, making it tedious to change from one implementation to the other.

- All allocation and freeing of objects is under control of the applications. Unlike most other object oriented languages, C++ does not provide any automatic garbage collection.

4.6. Efficiency of C++ Programs

Stroustrup claims that "a program using classes can be demonstrated to run at least as efficiently as a program using "old" C to implement the same abstraction". However, using abstractions (abstract data types) and data hiding will normally add to the processing "overhead" of the program, as this means that the attributes of an object are not directly accessible from the outside, but must be accessed through some mechanism provided by the object class. While other object oriented languages require a message passed to the object (Smalltalk and Objective-C [COX83]) or a procedure call (SIMULA) to get at object attributes, C++ provides in-line code for these simple functions. Without these fast access methods, programmers would sacrifice encapsulation and directly access the innards of objects to obtain efficiency. The in-line code methods make this security bypassing unnecessary.

This means that abstract data types with data hiding is possible in C++ with an absolutely minimal speed penalty.

The major obstacle to fast object oriented programs in C++ was found to be storage management. Normally, C++ objects are allocated from the heap storage using the customary C run-time mechanism (malloc()). As a rendering program may well handle 10000 polygons with 4000 vertex
points, the allocation (and freeing) of these objects is likely to spoil an object oriented rendering system.

In order to have high rendering performance in C++, polygon or point objects cannot be created (or transformed into other object types) during the rendering process, but rather must be passed through the rendering engine by reference.

To have reasonable performance when for instance solid objects are transformed into polygon meshes, objects were created using some special "bulk" storage allocators which allocate, say 100, objects at a time from the heap. The storage allocators are implemented in the factory objects.

4.7. The GAS Renderers

In addition to being a graphical toolkit with several object classes and methods available to the programmer, two versions of rendering programs have been built using the toolkit. One of the rendering programs is based on surface geometry, taking as input objects defined as surfaces consisting of polygons, either in a special textual graphic language or as data files in formats compatible with the Movie-BYU rendering program. The other rendering program takes as input a graphic language which defines Constructive Solid Geometry. The solid geometry can be extended with simple boundary geometry in the shape of polygons or polygon meshes. Thus the CSG definition language is a superset of the one used with the other renderer.

The surface renderer has the choice of three rendering algorithms: either a depth sorting algorithm based on BSP-trees (see Ch. 3.5.2.3.), a scan line algorithm with Phong Shading, or a RasterOp-based z-buffer algorithm (see [ZACHRISEN89b]).

The CSG renderer uses RasterOp-based algorithms for generating displays from the CSG trees [ZACHRISEN89b].
4.8. The CSG Definition Language

The CSG definition language and its parser and lexical analyzer are defined with the UNIX tools YACC [JOHNSON75] and LEX [LESK75]. The language makes it possible to construct complex objects, like the bracket on the figure above, with a simple program.

```plaintext
volume bracket = {
    translate(0.0, -0.2, 0.0)
    cube(0.8, 0.4, 0.6)
    +
    translate(0.0, 0.15, 0.0)
    rotate(-90.0 degrees, z_axis)
    {
        cylinder(0.7, 0.2)
        -
        cylinder(0.9, 0.12)
    }
    -
    translate(0.0, 0.05, 0.0)
    cube(0.4, 0.7, 0.45)
};
```

Fig. 4.6.
Program defining the bracket of Fig. 4.5.
In the grammar below, volume operators are declared as left associative. Moreover, all operators have the same precedence. Though union and intersection may be said to form sums and products, I do not feel it appropriate to apply the precedence levels of arithmetic operators. The associativity and precedence rules avoid shift/reduce parsing conflicts. For instance, in the bracket definition above the \( A + B - C \) expression is equivalent to \((A + B) - C\).

```plaintext
/* Tokens from the lexical analyzer: */
%token NAME         /* Object identifier */
%token F_VAL       /* Floating point number */
%token VOLUME      /* Volume object keyword */
%token ROTATE SCALE TRANSLATE /* Transformation keyw. */
%token CUBE CYLINDER WEDGE CONE SPHERE /* Volume keyw. */
%token X_AXIS Y_AXIS Z_AXIS /* Axis specifier */
%token DEG RAD     /* Angle unit specifier */
%left       '+', '*', '-'       /* Union, intersect, subtract */
%
volume_obj    :
    VOLUME name '" simple_volume '"
;

simple_volume :
    base_volume
    '|' '{' volume_expr '}'
;

volume_expr   :
    volume_expr '+' volume_expr
    '|' volume_expr '*' volume_expr
    '|' volume_expr '-' volume_expr
    '|' transf_list simple_volume
    '|' simple_volume
;

base_volume   :
    CUBE '(' F_L VAL ',' F_L VAL ',' F_L VAL ')''
    '|' WEDGE '(' F_L VAL ',' F_L VAL ',' F_L VAL ')''
    '|' CYLINDER '(' F_L VAL ',' F_L VAL ')' /* Along Y */
    '|' CONE '(' F_L VAL ',' F_L VAL ')' /* Along Y */
    '|' SPHERE '(' F_L VAL ')''
;

transf_list   :
    transf_list base_transform
    '|' base_transform
;

base_transform :
    TRANSLATE '(' F_L VAL ',' F_L VAL ',' F_L VAL ')''
    '|' ROTATE '(' angle ',' axis ')''
    '|' SCALE '(' F_L VAL ',' F_L VAL ',' F_L VAL ')''
;
```
angle
  : FL_VAL DEGREES
  | FL_VAL RADIANS
  ;

axis
  : X_AXIS
  | Y_AXIS
  | Z_AXIS
  ;

name
  : /* Empty (optional) */
  | NAME
  ;

Fig. 4.7.
Simplified YACC-(LALR) grammar for the CSG definition language. The semantic rules are not shown. Attribute definitions etc. are not shown. A scene definition file normally contains multiple graphic objects, attributes, and rendering directives.
5. MOVIEBOX

"Inside every big program there is a small program struggling to get out"
(Hoare's Law of Large Programs).

Moviebox is a rendering program that uses many of the algorithms developed for GAS.

The success of Moviebox is not due to any revolutionary new hidden surface algorithm, but rather to a state-of-the-art set of algorithms carefully implemented, so that performance requirements are met with today's general purpose workstations.

5.1. Moviebox Architecture

5.1.1. The Object and Polygon Database

Moviebox consists of an object and polygon database which is built in memory allocated from the C heap. The database is hierarchically structured allowing instancing of simple objects and grouping of objects into larger assemblies. The data model for the database is shown on Fig. 3.1. and Fig. 3.2.

5.1.2. The Command Processor

In order to allow the user to interact with the geometric model, a relatively simple command processor is included in Moviebox. Typically, the command processor allows the user to view an object from different angles, build new objects from old objects, and change object attributes and rendering parameters. A few "meta" commands have been added which allow command files to be built, allow repetitions of command sequences, and allow timing of rendering operations.
5.1.3. The Rendering Pipeline

The rendering pipeline is the central component in Moviebox. Views are specified with much flexibility, as was described in Ch. 3.3. When a view has been specified, the database is traversed. During traversal, attributes are bound, and object transformations are concatenated.

Using the effective extent testing algorithm described in volume 3 of this thesis [ZACHRISEN89c], parts of the model which are outside the viewing pyramid are effectively discarded. Based on the result of the extent analysis,
the Stacking Clipper algorithm will clip all polygons that are obstructing the viewing pyramid boundaries. The clipper works in four-dimensional homogeneous space. As soon as the clipping has been performed, polygons are projected to three-dimensional space and mapped to (three-dimensional) screen coordinates.

Beyond the clipper, the rendering pipeline works entirely with 32 bit integer arithmetic. The coordinates are actually transmitted as fixed-point 16 bit integers and fixed-point 32 bit integers (see figure below).

![Diagram](image)

Fig. 5.2.
Transmission formats for vertices between geometry processor and renderer. Colour can either be a single index (8 bits + 2 fractional bits) or three RGB values each 8 + 2 bits.

In the Transputer network implementation, a special slicer / distributor stage succeeds the clipper. This divides the polygons into horizontal screen "slices", such that a separate processor can handle each screen slice. The output from each parallel scan converter is collected by a "pixel collector" stage before output into the frame buffer.

In the single processor implementation, polygons are input to a bucket sorter, which sorts all polygons on ascending minimum y-coordinates. Each bucket
corresponds to a scan line on the screen. When all polygons to be displayed have been sorted into the buckets, scan conversion can start.

The scan converter implements a normal scan line z-buffer hidden surface algorithm. All polygons intersecting the current scan line are stored in an active edge list. One by one, the intersections between the active polygons and the current scan line are computed. The computed depth coordinates are compared with the z-buffer, and shaded output are generated in the draw-buffer. There are two versions of the scan line algorithm inside Moviebox: one version working on 8-bits indexed colours, and one version working on 3 x 8-bits RGB colours. Which one is used depends on the output device capabilities. The sampling technique which ensures good picture quality, and details of the DDA algorithm used for scan conversion are described in Ch. 3.6.3 and Ch. 3.6.4.

From the draw-buffer pixels are sent to the device driver. However, before the output device receives the scan line, Moviebox will optionally perform dithering, anti-aliasing filtering, and run-length coding on the scan line pixels.

5.2. Experience with C and Object Oriented Methodology

Moviebox is written in portable "C". Many of the program modules have been directly imported from GAS.

The translation of C++ programs into plain "C" is a simple process. As the C++ compiler actually outputs symbolic "C"-code, this translation could have been accomplished by the compiler. However, the translation was actually carried out in a manual editing session with the help of a flexible editor supporting substitution operations specified in terms of regular expressions etc. The main reason why automatic translation to "C" was not used, is that the C++ precompiler generates external function names for class methods that are guaranteed to be unique in the program namespace, but are not meant for human readership. Likewise, the implementation of inheritance results in an abundance of C struct pointers (objects -> structs) being typecast to its superclass struct pointers.
However, the C++ origin causes well-structured resulting code. Applying a strict programming style, many of the properties of the C++ code are retained in the C modules:

- The program is divided into modules, each defining one "abstract data type" - internal data structure, internal and external functions.
- "Objects" (structs) are still allocated (either from the heap or some specially pre-allocated pools) and initialized upon creation.
- Code will normally be re-entrant (no static variables) and able to handle multiple occurrences of each data type.

The concept of doing object oriented programming in a traditional procedure oriented programming language is in no way original. I do not argue that C is a challenger in the domain of object oriented programming. However, C programs can benefit from adapting an object-oriented programming style.

5.3. A Conceptual Model

The object-oriented origin of Moviebox has influenced the conceptual model (user model) which allows the user to handle objects of the following classes:

- Renderers attached to a device viewport.
- Views.
- Scenes (object hierarchies).
- Lighting environments.
- Material descriptions.

† The Toolkit Intrinsic for the X Window System [McCormack88] and all manufacturer supplied Widget sets are examples of this. Widgets are screen objects, implemented as instances of various screen object classes. Widget classes are structured in a class hierarchy, inheriting attributes and functionality. Though this could map into well-structured C++ programs, the toolkits are nevertheless implemented in plain C to guarantee portability of the code.
Though there are implementations of Moviebox using mouse and menu control [PHAN89], the normal way to control Moviebox rendering is by means of the command interpreter. Objects are given names when created, and are identified by these for later manipulation. The concept of the "current object" is used by the command interpreter. Thus a command will by default be sent to the "current object" with this command implemented. To direct a command to a specific object, the command is prefixed with the object name. A command can also be sent to all objects by prefixing the command with a "wildcard" character.

A renderer will be attached to one object from each of the other classes in order to generate an image. This model allows the same image to be generated on different devices with rendering parameters adapted to the hardware. Different views on the same scene can be shown in multiple screen viewports. Taken to extremes, Moviebox can handle multi-monitor set-ups for simulating front and side views in an airplane cockpit.

5.4. Benchmarks and a Performance Model

"Lies, Damned Lies, and Benchmarks".

Timing benchmarks have been carried out on a Sun-3 workstation with floating point processor M68881. Time was measured with the Sun OS operating system function "CLOCK(3C)" [SUN OS]†. As Moviebox is compatible with the geometry files formats used in the MOVIE-BYU rendering program, the choice of comparing timing results with this seemed obvious.

5.4.1. On MOVIE-BYU

As a background for the interpretation of the benchmark results, MOVIE-BYU will be described briefly.

MOVIE-BYU uses the Watkins "scan line span" hidden surface algorithm [WATKINS70], [CHRISTIANSEN87]. The virtues of this are briefly

---

† "Clock()" works on a discrete timing event, with a timestep of 16.67 milliseconds. The CPU time, sum of the user and system times of the calling process, is measured.
discussed in Ch. 3.5.3. MOVIE-BYU is a traditional Fortran implementation with statically dimensioned object arrays. As distributed, the settings are:

Max. no. of parts (objects) = 15
Max. no. of polygons = 2000
Max. no. of vertices = 2000
Max. no. of edges = 10000

It would have been interesting to regenerate the system with larger arrays, but since no documentation on this subject was available during the testing, this was not achieved. According to Fujimoto, there are major obstacles when trying to set up MOVIE-BYU for rendering more than 1200 hexahedrons. The test scenes rendered with MOVIE-BYU were thus limited to the values shown above. This is in contrast with the Moviebox system, which throughout uses dynamically modifiable data arrays. The only limitation to Moviebox scenes is when primary/virtual memory becomes exhausted.

Due to the different hidden surface algorithms used in MOVIE-BYU and Moviebox, one would expect somewhat different performance characteristics.

In the scan line z-buffer hidden surface algorithm of Moviebox, each polygon is compared with the accumulated depth-buffer. The time used for each polygon will thus be independent of the complexity of the rest of the scene.

In the scan line span hidden surface algorithm of MOVIE-BYU, running time estimates are more complex. Polygons must be sorted on their x-coordinates for each scan line. Depth sorting calculations will take place for each span encountered, and will involve a number of spans depending on the scene depth complexity.

Compared with a scan line z-buffer algorithm, one would expect that the Watkins algorithm will be at advantage with lower screen complexities, and that the scan line z-buffer will be better when scene complexity increases.
5.4.2. A Coarse Performance Model

Because the scan line z-buffer algorithm is very simple, a simple performance model can be set up for Moviebox. It can for instance be used for extrapolating performance to other scenes and processors.

The rendering scene can be described by a number of parameters:

- \( P_s \): No. of pixels on screen.
- \( F_e \): No. of polygons in environment.
- \( E_e \): No. of polygon edges in environment.
- \( V_e \): No. of vertices in environment.
- \( S_e \): No. of polygon spans in environment.
- \( P_e \): No. of polygon pixels in environment.

When doing a normal Moviebox rendering with Gouraud or "Cheap Phong" shading, the following cost factors are present:

- **Traversal.** Each polygon is traversed in the structured display file, computing attributes and concatenating transformations. (Cost: \( a * F_e \).)

- **Clipping.** Boxing is done on several levels of detail. When clipping has to be done on the polygon level, cost is proportional to number of edges. (Cost: \( b * E_e \).)

- **Transformations.** Each vertex is transformed by the concatenated view and object transformation. (Cost: \( c * V_e \).)

- **Lighting.** Lighting computations must be done at each vertex. (Cost: \( d * V_e \).)

- **Hidden surface.** Hidden surface computations are done at each polygon pixel. (Cost: \( e * P_e \).)

- **Shading.** Shading interpolation must be done at each scan line intersection with polygon edges, and on each polygon pixel. (Cost: \( f * S_e + g * P_e \).)
Dithering, anti-aliasing and pixel output are done at each pixel of the screen. (Cost: $h \times P_s$.)

The rendering cost becomes:

$$R_c = a \times F_e + b \times E_e + c \times V_e + d \times V_e + e \times P_e + f \times S_e + g \times P_e + h \times P_s$$

Supposing that the scene is isotropic (and pixels are square), the average polygon height equals the average polygon width. The number of polygon scans ($S_e$) will thus be a dependent variable of the number of polygon pixels:

$$P_e = \left(\frac{S_e}{F_e}\right)^2 \times F_e$$

$$S_e = \left(\frac{P_e \times F_e}{2}\right)^{1/2}$$

Most scenes will consist of simple polygons with three or four edges. Moreover, unless vertices shared by several polygons are transformed just once, the number of vertex transformations will equal the number of edges. Since the dominant cost is the vertex transformation, we will express the cost as a function of $V_e$ rather than of $F_e$, $E_e$, and $V_e$.

We can then express the computation cost as:

$$R_c = a \times V_e + b \times P_e + c \times F_e^{1/2} \times P_e^{1/2} + d \times P_s$$

When studying how cost grows with the detail of the model, the term \textit{depth complexity} is often convenient. Depth complexity ($D_c$) of a model gives the average number of polygon intersections for a ray penetrating a scene. If increasing the complexity of a model means that vertices become denser to better approximate a surface, depth complexity of the model will tend to be constant over this operation.

We have that:

$$D_c = \frac{P_e}{P_s}$$

and the computation cost model becomes:
\[ R_c = a \cdot V_e + b \cdot (D_c \cdot P_s) + c \cdot (D_c \cdot F_e \cdot P_s)^{1/2} + d \cdot P_s \]

In Moviebox, when solid models are rendered (in contrast to sheet models), back faces are normally culled, before entering the scan converter. This can be compensated for by setting the \( F_e \) variable to 0.5 times the number of polygons in the scene.

5.4.3. On the Test Scenes

Test scenes consist of variations of a scene of hexahedrons (cubes) parameterized to test the effect of \( xy \) or depth complexity, and one camshaft model originating from the APS (Advanced Production System) project [BJØRKE87].

The hexahedron scene is well-structured with all vertices shared between adjacent polygons, and polygons defined in a clockwise direction to define the outside of the object. The renderer can thus use back-face culling to reduce the complexity of hidden surface removal.

Though the camshaft model is also a typical solid object, the polygon database is not well-ordered such that front and back-faces are defined. This means that back-faces can not be culled. Moreover, the model is not defined such that vertices are shared between adjacent patches.

The following scenes were used for the time measurements:

1. A very simple scene consisting of 2x2x2 hexahedrons.
2. Also a simple scene consisting of 4x4x4 hexahedrons.
3. A scene with a relatively high depth complexity consisting of 4x4x14 hexahedrons. This was viewed in four different scales to vary the number of shaded pixels while keeping the geometry operations constant. This is about the most complex scene that can be tested with MOVIE-BYU with the default geometry array sizes.
4. A relatively complex scene consisting of 12x12x12 (1728) hexahedrons. This scene caused the Sun-3 with 4 Mbyte to repeatedly swap between disk and memory.

5. A scene consisting of a long and slender camshaft.

5.4.4. Test Results

The table below shows the scene parameters for the test scenes, the actual measured computation time for Moviebox and the result of the model using the estimated parameters. With the exception of experiments 4a and 4b, where the Sun-3 was busy with swapping pages to disk, the model is within 5% of the measured time for all experiments.

Table 5.1.
Scene parameters for the ten test scenes with actual running times (in seconds) for Moviebox on a Sun-3/50 with M68881 FPP, and estimated performance according to the developed performance model. Parameters are described in the text above.

<table>
<thead>
<tr>
<th></th>
<th>( F_e )</th>
<th>( V_e )</th>
<th>( P_e )</th>
<th>( D_c )</th>
<th>Moviebox</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Boxes 2x2x2</td>
<td>24</td>
<td>64</td>
<td>137945</td>
<td>0.526</td>
<td>3.1</td>
<td>3.06</td>
</tr>
<tr>
<td>2: Boxes 4x4x4</td>
<td>192</td>
<td>512</td>
<td>242472</td>
<td>0.925</td>
<td>5</td>
<td>4.89</td>
</tr>
<tr>
<td>3a: Boxes 4x4x14</td>
<td>672</td>
<td>1792</td>
<td>23915</td>
<td>0.091</td>
<td>4.4</td>
<td>4.44</td>
</tr>
<tr>
<td>3b: (zoom 2)</td>
<td>672</td>
<td>1792</td>
<td>95955</td>
<td>0.366</td>
<td>5.4</td>
<td>5.47</td>
</tr>
<tr>
<td>3c: (zoom 3,4)</td>
<td>672</td>
<td>1792</td>
<td>266005</td>
<td>1.015</td>
<td>7.3</td>
<td>7.29</td>
</tr>
<tr>
<td>3d: (zoom 4)</td>
<td>672</td>
<td>1792</td>
<td>383621</td>
<td>1.463</td>
<td>8.3</td>
<td>8.4</td>
</tr>
<tr>
<td>4a: Boxes 12x12x12</td>
<td>5184</td>
<td>13824</td>
<td>383968</td>
<td>1.465</td>
<td>28.5</td>
<td>24.7</td>
</tr>
<tr>
<td>4b: (other view)</td>
<td>5184</td>
<td>13824</td>
<td>540060</td>
<td>2.06</td>
<td>33.1</td>
<td>26.9</td>
</tr>
<tr>
<td>5a: Camshaft (w/o. cull)</td>
<td>574</td>
<td>1806</td>
<td>66751</td>
<td>0.255</td>
<td>5.3</td>
<td>5.03</td>
</tr>
<tr>
<td>5b: (zoomed 3)</td>
<td>574</td>
<td>1806</td>
<td>343356</td>
<td>1.31</td>
<td>8</td>
<td>7.87</td>
</tr>
</tbody>
</table>

For a Sun-3/50 with floating point processor 68881 and 4Mb RAM, the model coefficients were estimated as:

\( a = 1000 \mu s \) (per vertex)

\( b = 6.0 \mu s \) (per polygon pixel)
\[ c = 150 \, \mu s \quad \text{(per polygon scan line)} \]
\[ d = 7.0 \, \mu s \quad \text{(per screen pixel)} \]

The geometry coefficient will mostly be influenced from floating point calculations for transformations and lighting calculations. The pixel coefficients will mostly be affected by simple, integer computations on the horizontal segment span. As the measured time includes output of image to disk, the system's disk input/output capacity will influence the screen pixel coefficient.

To test this, some of the tests were also run on different machine configurations: a Sun-4/110 (SPARC/RISC) processor equipped with 32Mbyte RAM, and a Sun-3/470 (68030/68882) processor with 16 Mbyte RAM. The large RAM allowed the larger models to be handled without causing swapping of pages to disk. The measured computation times are shown in the figure below.

The coefficients for the Sun-4/110 were estimated as:

\[ a = 200 \, \mu s \quad \text{(per vertex)} \]
\[ b = 3.1 \, \mu s \quad \text{(per polygon pixel)} \]
\[ c = 110 \, \mu s \quad \text{(per polygon scan line)} \]
\[ d = 5.0 \, \mu s \quad \text{(per screen pixel)} \]

The coefficients for the Sun-3/470 were estimated as:

\[ a = 130 \, \mu s \quad \text{(per vertex)} \]
\[ b = 4.0 \, \mu s \quad \text{(per polygon pixel)} \]
\[ c = 55 \, \mu s \quad \text{(per polygon scan line)} \]
\[ d = 2.9 \, \mu s \quad \text{(per screen pixel)} \]

As seen from the measurements, the Sun-4, and the Sun-3/470 (with a huge memory) perform extremely well on complex models, where the processing time is more influenced by geometry operations than pixel operations. The coordinate transformation processing and the screen pixel output (to disk) on the Sun-3/470 is significantly faster than on the Sun-4/110, while the speed of pixel shading operations is slightly higher on the SPARC processor.
Table 5.2.

Comparable timing data (in seconds) for Moviebox on a Sun-3/50 with 68881 FPP, on a
Sun-4/110, and on a Sun-3/470 with 68882 FPP.

<table>
<thead>
<tr>
<th>#polygons</th>
<th>depth</th>
<th>Moviebox Sun-3/50</th>
<th>Moviebox Sun-4/110</th>
<th>Moviebox Sun-3/470</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Boxes 2x2x2</td>
<td>192</td>
<td>0.5262</td>
<td>3.1</td>
<td></td>
</tr>
<tr>
<td>2: Boxes 4x4x4</td>
<td>1536</td>
<td>0.925</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3a: Boxes 4x4x14</td>
<td>5376</td>
<td>0.0912</td>
<td>4.4</td>
<td>2.24</td>
</tr>
<tr>
<td>3b: (zoom 2)</td>
<td>5376</td>
<td>0.366</td>
<td>5.4</td>
<td>2.85</td>
</tr>
<tr>
<td>3c: (zoom 3.4)</td>
<td>5376</td>
<td>1.0147</td>
<td>7.3</td>
<td>3.94</td>
</tr>
<tr>
<td>3d: (zoom 4)</td>
<td>5376</td>
<td>1.4634</td>
<td>8.3</td>
<td>4.59</td>
</tr>
<tr>
<td>4a: Boxes 12x12x12</td>
<td>41472</td>
<td>1.4647</td>
<td>28.5</td>
<td>9.83</td>
</tr>
<tr>
<td>4b: (other view)</td>
<td>41472</td>
<td>2.0602</td>
<td>33.1</td>
<td>12.58</td>
</tr>
<tr>
<td>5a: Camshaft (w/o. cull)</td>
<td>1806</td>
<td>0.2546</td>
<td>5.3</td>
<td>3.1</td>
</tr>
<tr>
<td>5b: (zoomed 3)</td>
<td>1806</td>
<td>1.3098</td>
<td>8</td>
<td>4.76</td>
</tr>
</tbody>
</table>

The timing measurements of MOVIE-BYU were not as comprehensive as
those for Moviebox. The scenes used for testing Moviebox and MOVIE-BYU
were identical, but the way of specifying viewing parameters for the two
renderers were different, so viewing parameters were set such as to give
approximately the same view. The results below show the expected increase
in running time when depth complexity in the scene increases. For simpler
scenes, one would expect a better performance of the scan line span
algorithm. However, the faster operations of Moviebox on simpler scenes is
probably due to the efficiency of C programs in handling pixels.
Table 5.3.

Comparable timing data (in seconds) for MOVIE-BYU and Moviebox on a Sun-3/50 with 68881 FPP.

<table>
<thead>
<tr>
<th></th>
<th>#polygons</th>
<th>depth</th>
<th>MOVIE-BYU</th>
<th>Moviebox</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>complexity</td>
<td>Sun-3</td>
<td></td>
<td>Sun-3</td>
</tr>
<tr>
<td>1</td>
<td>Boxes 2x2x2</td>
<td>192</td>
<td>0.5262</td>
<td>8.8</td>
</tr>
<tr>
<td>2</td>
<td>Boxes 4x4x4</td>
<td>1536</td>
<td>0.925</td>
<td>29</td>
</tr>
<tr>
<td>3a</td>
<td>Boxes 4x4x14</td>
<td>5376</td>
<td>0.0912</td>
<td></td>
</tr>
<tr>
<td>3b</td>
<td>(zoom 2)</td>
<td>5376</td>
<td>0.366</td>
<td></td>
</tr>
<tr>
<td>3c</td>
<td>(zoom 3.4)</td>
<td>5376</td>
<td>1.0147</td>
<td>70</td>
</tr>
<tr>
<td>3d</td>
<td>(zoom 4)</td>
<td>5376</td>
<td>1.4634</td>
<td></td>
</tr>
<tr>
<td>5a</td>
<td>Camshaft (w/o.cull)</td>
<td>1806</td>
<td>0.2546</td>
<td>25</td>
</tr>
<tr>
<td>5b</td>
<td>(zoomed 3)</td>
<td>1806</td>
<td>1.3098</td>
<td>29</td>
</tr>
</tbody>
</table>

All measurements for Moviebox and MOVIE-BYU include writing pixel runs to a file. MOVIE-BYU will output RGB-runs, while Moviebox will output runs of colour indices.

In order to generate pictures for simple workstations with 256 colours or less, MOVIE-BYU must use dithering to produce reasonable picture quality. MOVIE-BYU has built-in Floyd-Steinberg dithering (see Ch. 3.10.6), using floating point arithmetic. Dithering increased the running time of test 3c from 70 seconds to 210 seconds.

By default, Moviebox will do "Cheap Phong" shading. This requires a linear interpolation during shading operations. With flat shading, rendering was measured to be 15% faster. When full 3 x 8 bits RGB is generated instead of just 8 bits indexed colours, the polygon pixel times increase, so that the total running times are approximately doubled. With RGB output, 2x2 supersampling and 3x3 anti-aliasing filtering, the times measured for RGB are approximately doubled.

5.4.5. Other Timing Results

Fujimoto, Tanaka, and Iwata [FUJIMOTO86] report on comparative timing between MOVIE-BYU and their so-called "ARTS - Accelerated Ray Tracing System". On a VAX 11/785, generating images for a 512x512 frame buffer,
MOVIE-BYU spends approx. 2.5 minutes on an image with 200 hexahedrons and approx. 10 minutes on an image with 1200 hexahedrons. For ARTS, the respective times spent were 11.5 minutes and 12.5 minutes. The conclusion is that for simple scenes (<1500 hexahedrons), scan line renderers are superior to ray tracers. However, for more complex scenes, ray tracers using speed-up techniques like ARTS may prove to be faster.

Hall [HALL86] reports on timing a rendering algorithm based on a "modified Watkins" scan line algorithm with different shading techniques and with ray tracing used for reflectance and refraction (based on visible surfaces from the Watkins algorithm). Hall's performance ratios are given as follows:

<table>
<thead>
<tr>
<th>Shading Method</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant colour hidden surface algorithm</td>
<td>0.47</td>
</tr>
<tr>
<td>Gouraud shading</td>
<td>0.48</td>
</tr>
<tr>
<td>Phong shading</td>
<td>1.00 (ref.)</td>
</tr>
<tr>
<td>Texture and reflection mapping</td>
<td>1.88</td>
</tr>
<tr>
<td>Recursive ray tracing (Whitted model)</td>
<td>4.16</td>
</tr>
<tr>
<td>Distributed ray tracing (Hall model)</td>
<td>4.83</td>
</tr>
</tbody>
</table>

Rogers [ROGERS85] gives performance ratios for rendering a very simple scene (two objects) with different hidden surface renderers as implemented by him:

<table>
<thead>
<tr>
<th>Shading Method</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z-buffer (full screen)</td>
<td>1.0 (ref.)</td>
</tr>
<tr>
<td>Scan line z-buffer</td>
<td>1.9</td>
</tr>
<tr>
<td>Watkins algorithm</td>
<td>2.1</td>
</tr>
<tr>
<td>Warnock algorithm</td>
<td>6.2</td>
</tr>
<tr>
<td>Ray tracing</td>
<td>9.2</td>
</tr>
</tbody>
</table>

There also exist a set of scenes proposed as "standard benchmarks" by their author [HAINES87]. However, these environments are mainly constructed for ray tracing renderers.

5.4.6. Performance Test Conclusions

The performance measurements show that Moviebox is much faster than MOVIE-BYU. For simple scenes, Moviebox is nearly three times faster. For
more complex scenes, Moviebox is 10 times faster. If even more complex scenes could have been tested on MOVIE-BYU, the algorithm discussion above indicates that Moviebox would be even more superior.

The MOVIE-BYU measurements compare well with the tests done by Fujimoto for simple scenes (200 hexahedrons). For more complex scenes, Fujimoto reports a rendering time for 1200 hexahedrons of 11.5 minutes (on a VAX), while the tests of Moviebox show a rendering time of 30 seconds for 1700 hexahedrons (on a Sun-3).

From the tests of Moviebox the following performance ratios can be set up:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant colour scan line z-buffer algorithm</td>
<td>0.85</td>
</tr>
<tr>
<td>Gouraud shading (or &quot;Cheap Phong&quot;) scan line z-buffer</td>
<td>1.00 (ref.)</td>
</tr>
<tr>
<td>Gouraud shading with full RGB interpolation</td>
<td>2.0</td>
</tr>
<tr>
<td>Gouraud shading and 3x3 filter anti-aliasing</td>
<td>4.0</td>
</tr>
</tbody>
</table>

The comparisons between the Motorola 68020/68881 in Sun-3 and the SPARC processor in Sun-4 go naturally in favour of the Sun-4. For geometry operations, the Sun-4 tends to be approximately five times faster. For pixel operations the Sun-4 is 30%-90% faster.

The Moviebox program size (code), was 180 Kbyte on the Sun-3 (M68020), and 131Kbyte on the Sun-4 (SPARC).

### 5.5. Shortcomings and Extensions

"Simple things should be fast, complex things should be possible,"

freely after Alan Kay.

Moviebox lacks some of the brilliant lighting effects seen on images produced by ray tracing renderers. Most of the features below can easily be incorporated in renderers that generate images for full-colour (RGB) frame buffers and that use some ray tracing technique to cope with global lighting. However, the latter type of renderer can hardly compete with Moviebox on image generation time.
Adding other renderers to the Moviebox framework is rather simple. The bulk of code is concerned with other parts of the scene building and rendering process than just the hidden surface algorithm.

A desirable situation would be to have four algorithms available to the user:

- Depth sorting algorithm for "GKS-engines" (terminals with fast polygon drawing).
- The implemented Z-buffer algorithm as the workhorse for the average workstation user.
- An A-buffer algorithm with simple environment texture mapping for high-quality image production.
- A ray tracer for special lighting effects.

The existing Moviebox provides a means for quickly getting the scene and rendering parameters right. Sophisticated copies with fancy lighting effects could then be made in batch with one of the more time-consuming algorithms.

Images from the test examples along with samples from various applications of Moviebox are shown on the colour plates (Appendix A).
APPENDIX A. COLOUR PLATES

The colour illustrations in this volume are all produced as photographs taken from the face of the Hitachi monitor of a Sun-4/110 workstation with CG4 frame buffer. The frame buffer is limited to 256 simultaneous colours.

Plates 1, 2, 4, and 5 have been produced as 3x8 bits RGB images and dithered to 8x8x4 levels (ordered dithering). Moviebox normally divides the colour table into slices for each material to be rendered in order to provide smooth shading. This technique has been used for the rest of the plates.

All images have been rendered by the author using Moviebox in 1050x800 pixels resolution.

Some of the figures have "smoothed" their (polygon) geometry. This is performed by generating vertex normals averaging surface normals of adjacent polygons and using these for lighting computations. "Cheap Phong" shading will interpolate this shade across the polygon and create a "smoothly rounded" object even though the actual geometry is a simple rectangular box. For instance, the gray part of the robot arm on plate 3 have been "smoothed" this way.

The photographs were taken using 100 ASA colour print film, using a single-lens reflex camera with 50 mm lens, 1/2 second, f 8. In the final printed report, photos have been copied on a Canon Colour Copier.

Plate 1.
Colour palette setup for 8x8x4 RGB display. Colours have been mapped to RGB coordinate system.

Plate 2.
Colour palette setup for rendering four materials (metallic red, metallic gray, plastic green, and plastic blue) with "cheap Phong" shading. Note how specular highlights cause longer distance between colours towards end of each ramp, and how the white "plastic" highlights of the green and blue materials cause these ramps to curve towards white.
Plate 3.
Shaded display of robot using the colour space from 2 (robot model courtesy of G. Senneset).

Plate 4.
Shaded display of robot rendered in full RGB and dithered to 8x8x4 levels of RGB (see colour space on Colour plate 1).

Plate 5.
Shaded display of robot rendered in full RGB, supersampled and filtered by 3x3 "Gaussian", and dithered to 8x8x4 levels of RGB. It can be noted how the "jaggies" on the near-vertical edges of the robot (blue box) have been smoothed.

Plate 6.
The complex 12x12x12 hexahedron array used in the test above, consisting of 6412 polygons.

Plate 7.
The camshaft used in the test (courtesy of APS project).

Plate 8.
Passenger ship from a Singapore harbour scene (courtesy of Norcontrol Simulations).

Plate 9.
Off-shore construction being inspected by submarine (courtesy of SINTEF dep. 71).

Plate 10.
Three-link mechanism with results from finite-element-analysis mapped onto surface (courtesy of SINTEF dep. 20). This is an image from a video tape generated for a showing at ESA (European Space Agency).
Plate 11.
Turbine blade (courtesy of Kvaerner Brug and SINTEF dep. 20).
APPENDIX B. REFERENCES.


CHRISTIANSEN87 Christiansen, H., M. Stephenson, B. Nay, A. Grimsrud MOVIE.BYU Training Text - Graphics Utah Style, 1980 North 1450 East, Provo, Utah 84604, USA.


MELEN88a Melén, T., A. Sommerfelt, "PARAGRAF - Parallell algoritmer og arkitekturer for grafisk databehandling,"
Computer Science Dept. NTH / RUNIT-SINTEF report STF14 A88003, ELAB-RUNIT, N-7034 Trondheim.


Phong, B. T., "Illumination for Computer Generated Images," Communications of ACM 18, June 1975, pp. 311-317


RIESENFELD84 Riesenfeld et al., *Alpha_1 Manual*, Dept. of Computer Science, University of Utah, Salt Lake City, Utah 84112, 1984.


WHITTLED86 Whitted, T., "Introduction to Ray Tracing," In *Developments in Ray Tracing*, Tutorial at SIGGRAPH '86.


Computer Graphics Rendering Techniques
with an Emphasis on Performance Issues

Volume 2

Rendering on a RasterOp Processor
Picture Generation Algorithms for Modern Engineering Workstations.
ABSTRACT OF VOLUME 2

This paper focuses on presentation of three-dimensional polygon data on modern engineering workstations. In particular, near-real-time generation of shaded raster images is discussed.

Using a large frame buffer and an instruction set based on raster operations, images are rendered with hidden surface removal. The strength of this approach is that of providing flexibility in the rendering. Uniform cosine shading and smooth shading with highlights and translucent surfaces are supported along with depth-cued "hidden-line" displays.

More elaborate use of the same basic operations allows shadow casting from multiple light sources and direct rendering of objects represented by CSG (Constructive Solid Geometry).

The price to pay for this flexibility, compared with specialized microcoded z-buffer-based hidden surface removal algorithms, is a slightly lower speed. However, as the raster operations are well suited for implementations with many parallel processors, we can still win.
PREFACE TO VOLUME 2

The following paper is based on an implementation done during 1985/1986 on an ICAN Raster Workstation. The work has been partly financed by the NTNF project, "High-End Raster" (project management by ICAN a/s).

The hidden-surface renderer was first demonstrated publicly at DAK/DAP-86, November 1986. The shadow casting algorithm and the CSG-rendering algorithm were completed during the spring of 1987.

The road to completion of this report has been long and thorny. The disk and CPU of the ICAN Raster Workstation broke down before this research was completed. The primary problem resulting from this was production of the illustrations and timing measurements for this report. In order to produce these, the C++ compiler and the whole GAS system was ported onto Sun-3 workstations during the autumn of 1989.

Thanks to R. D. Bergeron and A. Kaufmann for their comments on an earlier version of this manuscript.
# TABLE OF CONTENTS

ABSTRACT OF VOLUME 2 ......................................................... I
PREFACE TO VOLUME 2 ......................................................... II
TABLE OF CONTENTS ......................................................... III

1. INTRODUCTION TO RASTEROPS ........................................... 1
  1.1. RasterOp Basics ..................................................... 1
  1.2. Our Drawing Primitives ............................................ 2

2. A HIDDEN SURFACE REMOVAL ALGORITHM ............................. 4
  2.1. Basic Hidden Surface Algorithm .................................. 5
  2.2. Smooth Gouraud Shading ......................................... 9
  2.3. "Cheap Phong" Shading ........................................... 9
  2.4. Hidden Line Display with Depth Cue ............................ 10
  2.5. Transparency Simulation ........................................ 11

3. SHADOW CASTING ......................................................... 12

4. DIRECT RENDERING OF CSG .............................................. 15
  4.1. Rendering CSG Objects ......................................... 15
  4.2. Algorithm for Drawing Convex CSG Objects .................... 21
  4.3. Handling Non-Convex Primitives ................................. 23
  4.4. Algorithm for Drawing Concave CSG Objects ................... 25
  4.5. Touching Objects ............................................... 26

5. ADAPTABILITY ........................................................... 28
  5.1. An Improved Set of RasterOps .................................. 28
  5.2. Z-Buffer Resolution Problems .................................. 28
  5.3. A Scan Line Algorithm .......................................... 28
  5.4. RasterOp Software Architecture ................................ 29
  5.5. A RasterOp Server ............................................... 29

6. THE RENDERING ENVIRONMENT ......................................... 31

7. SOME PERFORMANCE MEASUREMENTS ................................. 32
  7.1. Measurements on the ICAN Raster Workstation .................. 32
  7.2. Measurements on the Sun-3/470 ................................ 32
  7.2.1. Z-Buffer Algorithm .......................................... 33
  7.2.2. Shadow Casting Algorithm .................................. 34
  7.2.3. CSG Algorithm ............................................... 35

8. RELATED WORK ......................................................... 37

9. CONCLUSIONS .......................................................... 39

APPENDIX A. COLOUR PLATES ............................................. 1
APPENDIX B. REFERENCES .................................................. 1
IV Volume 2 - Rendering on a RasterOp Processor
1. INTRODUCTION TO RASTEROPS

Raster operations (or RasterOps [NEWMAN79], or BitBlts [GOLDBERG83]) are operations carried out on each pixel inside a rectangular area in the frame buffer. These are common among modern raster-screen equipped microcomputers and engineering workstations. Their typical use is for document preparation with multi-font text generation (as in this paper), various feedback techniques, and window management.

For brevity, I shall in the following refer to these as RasterOps.

In the following paper, a RasterOp-based instruction set is used in a general purpose engineering workstation for various types of three-dimensional object presentation including hidden-surface removal, smooth shading and shadow casting.

1.1. RasterOp Basics

"RasterOps" is here used as a common name for operations modifying each pixel inside a rectangular raster (the destination raster). The value of the destination raster pixel can be computed as a function of the previous pixel value \( d \), and the value of the corresponding pixel \( s \) in a source raster of the same size. In some systems, the destination raster is also function of a third area which may be a texture raster.

In monochrome systems, 16 different raster operations can be defined for the two-operand version (combinations of destination and source), while 256 different raster operations can be defined for the three-operand version (combinations of destination, source, and texture).

For colour systems, there are many degrees of freedom. A choice of six of the most commonly used operations are described in [NEWMAN79]. Here a constant colour \( c \) is added to the parameter list of the RasterOp function \( g(d, s, c) \). The six operations are described as follows:
2 Volume 2 - Rendering on a RasterOp Processor

1. write_rectangle: \( g(d, s, c) = c \). The new destination value is constant throughout the raster.

2. write_mask: \( g(d, s, c) = (\text{if } s \neq 0 \text{ then } c \text{ else } d) \). The mask tells where to apply the specified colour.

3. write_colour: \( g(d, s, c) = (\text{if } s \neq \text{ transparent} \text{ then } s \text{ else } d) \). Copy source to destination, but omit "background".

4. copy_raster: \( g(d, s, c) = s \). Copy source to destination.

5. invert_mask: \( g(d, s, c) = (\text{if } s \neq 0 \text{ then } f(d) \text{ else } d) \). The mask tells where to invert.

6. invert_rectangle: \( g(d, s, c) = f(d) \). Invert destination raster.

Guibas and Stolfi have developed a good formal representation for RasterOp representation by means of their MUMBLE language [GUIBAS82]. For shortness and readability, I shall describe algorithms in an informal manner, using Newman and Sproull's RasterOps as elementary operations.

### 1.2. Our Drawing Primitives

Though implemented as four basic operations (write_rectangle, write_colour, write_mask, write_texture) with four output modes (write, and_not, or, xor), our RasterOps [ZACHERISEN81] are similar to the ones described above. The major difference is the addition of the write_texture operation which repeatedly copies pixels from a texture rectangle into the destination area where a source mask area is non-zero.

A fifth RasterOp (compare) does simple arithmetic comparisons (=, ≠, >, <, ≥, ≤) between pixel values in two source rectangles, outputting a mask (=1 means true, =0 means false) in the destination rectangle.

For scan conversion the following instructions complements the RasterOps:

1. Draw line and interpolate pixel value between endpoints (line_interpolate).
2. Draw convex polygon and interpolate pixel value across interior (\textit{polygon\_fill}).

The scheme used for interpolating the pixel value across the polygon is the one used in so-called "Gouraud shading"; specify colour at vertices, interpolate colour along edges between vertices, interpolate colour along scanline from left edge to right edge [GOURAUD71].

The implementation of these instructions is clever enough to use a faster algorithm if colour is constant across the polygon/line$^\dagger$.

The primitives described above are implemented using microcode on a 16-bit bit-slice raster processor which is used with a 68000 processor running the UNIX operating system for an engineering workstation [ICAN81]. The raster processor handles a frame buffer normally of size 1024x1024, but extendible.

More details on the rendering environment are found in Ch. 6.

\footnotesize{$^\dagger$ Some graphic processors include a "seed-fill" operation for drawing polygons. Though able to handle concave polygons, their speed limits their use to interactive painting programs. Moreover, most solid modelling packages will break down their surface primitives to polygons with three or four vertices. Thus the presence of a restricted triangle-filling primitive is far superior to the seed\_fill for fast three-dimensional surface drawing.}$
2. A HIDDEN SURFACE REMOVAL ALGORITHM

A so-called z-buffer hidden surface removal algorithm has for each pixel in the frame buffer, a corresponding cell in a z-buffer (also called depth buffer). As the algorithm draws polygons on the display, the maximum z-coordinate of each polygon pixel is accumulated in the depth-buffer. If at a pixel, the z-coordinate of the polygon is larger than the one in the z-buffer, the z-buffer is set to the new value, and the frame buffer is set to the computed shade.

Software-based z-buffer algorithms have gained popularity for scan line rendering due to their simplicity and their ability to perform without imposing constraints on the graphic objects to be rendered. Hardware/firmware-based z-buffer algorithms in "solid-modelling engines" have gained popularity due to their simple implementation, speed, and the ever-decreasing cost of memory chips.

In such "solid-modelling engines", the time needed for a z-buffer display of a scene increases linearly with the number of polygons to be rendered (and with the number of pixels in each polygon). This is also the case with the implementation described here.

The goal aimed at with many of these systems is to provide real-time or near-real-time presentation of animated, shaded, raster images. The application area may be robotics or mechanical engineering.

Modern workstations are often generously provided with megabytes of memory, and in many cases this is directly accessible to the rendering processor, as general program memory or as invisible parts of a dedicated frame buffer. In our case, we have such space available for saving obscured parts of windows, various text fonts, texture patterns and z-buffer usage.

On our system, we can subdivide the 1024x1024 frame buffer by zooming/panning with a factor of two. This makes three invisible "screenfuls" available for computations:
Fig. 1a
Frame buffer divided into working areas.

1. **Z-buffer**: accumulates maximum z-coordinates as polygons are drawn.

2. **Scratch-buffer**: used for filling polygon (shape) and interpolating z-values (and colour values) across polygon.

3. **Draw-buffer**: used for final output of visible parts of polygon.

In the Scratch-buffer and in the Z-buffer the top-most plane is used for masks, while the rest of the bit-planes are used for the actual Z-buffer.

### 2.1 Basic Hidden Surface Algorithm

**Prerequisites:**
Incoming polygons must have been clipped to the size of the Draw-buffer. Z-coordinate range must have been mapped to the available Z-buffer depth (with 11 planes available for the Z-buffer, that is to the range 0 - 2047). For each polygon the extent (smallest rectangle containing the polygon) is computed. This extent is used to limit the number of pixels affected by RasterOps in the algorithm.

At initialization, the Z-buffer is cleared to zero (minimum z-coordinate).
Accumulated polygons in Draw-buffer and Z-buffer before triangle output.

For each polygon, the following procedure is carried out:

0. The Scratch-buffer (and mask) is cleared to 0 (write_rectangle).

1. Polygon z-coordinates are interpolated in the Scratch-buffer (polygon_fill) with the high mask bit set. Thus, the polygon mask is formed in the mask plane while the z-coordinates are interpolated in the other Scratch-buffer planes.

Triangle's z-coordinates have been interpolated in the Scratch-buffer.
2. Excluding the mask plane, the Scratch-buffer (only the polygon extent), is compared to the previous contents of the Z-buffer (compare). Where the Scratch-buffer is smaller than the Z-buffer, a bit is set in the mask plane of the Z-buffer.

3. Using this (polygon-obscured) mask, the Scratch-buffer z-coordinates and (polygon-) mask is cleared (write_mask). Thus, the Scratch-buffer mask now designates the visible part of the polygon.

![Diagram](image)

Fig. 1d

Obscured parts of triangle have been masked away from the Scratch-buffer.

4. The Z-buffer is now updated. All non-cleared cells in the Scratch-buffer are copied to the z-buffer (write_colour).
Fig. 1e
Triangle's z-coordinates have been added to the Z-buffer.

5a. For polygons with uniform colour, the Scratch-buffer mask is projected into the Draw-buffer with the colour resulting from the shading computation (write_mask).

Alternatively, to increase the colour resolution, the Draw buffer designated by the Scratch-buffer mask could be tiled with a precomputed texture representing the desired shade [JARVIS76] (write_texture).

Fig. 1f.
Unobscured parts of the triangle have been projected into the Draw-buffer with the desired colour (or texture).
With the frame buffer divided into four equally sized areas, the Draw-buffer can be kept invisible during image generation. Moving the screen focus to the Draw-buffer after generation is complete gives us smooth transitions between images ("double-buffering").

2.2. Smooth Gouraud Shading

Instead of a simple uniform cosine shading of the polygon, we can achieve a smooth Gouraud shading of the polygon by adding some more RasterOps.

5b.1. The Scratch-buffer is cleared (write_rectangle).

5b.2. The polygon is again drawn in the Scratch-buffer interpolating the specified vertex colours across the interior (polygon_fill). For full colour images in RGB-space, this is carried out for all three colour channels using an appropriate output mask for each channel (see discussion below).

5b.3. The invisible part of the polygon is cleared using the mask previously generated in the Z-buffer (write_mask).

5b.4. The remaining part of the polygon is then copied to the Draw-buffer (write_colour).

2.3 "Cheap Phong" Shading

As long as we don't cast shadows, the most "natural" choice for a single light-source is to coincide with the eye's position. Adhering to this limitation, and adapting a lighting model similar to Phong [PHONG75], the shaded pixel values for a display object are limited to being a simple function of the dot-product between the surface normal and the eye/light vector \((N \cdot L)\)†:

\[
I = I_a \cdot k_a + I_p \cdot k_d \cdot (N \cdot L) + I_p \cdot k_s \cdot (N \cdot L)^n
\]

† We may differentiate between the lighting model proposed by Phong which includes the specular reflection term, and Phong's shading method which involves linear interpolation of surface normals across the scanline.
If the number of differently coloured objects is small, it is advantageous to store the dot-product \((N \cdot L)\) in the frame buffer and tabulate the shading function in segments of the video lookup-table, one segment for each differently coloured object (see palette on colour plate 2 in Volume 1). With this approach, shaded drawing is done with a linear interpolation of the dot-product, rather than being a three-component interpolation on RGB-values.

The major drawback with this approach is that the intensity function becomes very steep for shiny materials when \((N \cdot L)\) approaches 1. This will result in large steps in the colour value around the highlights, visible as contours.

Applying the lighting model of Phong, together with a linear interpolation of the dot-product, is sometimes referred to as "cheap Phong" shading.

### 2.4. Hidden Line Display with Depth Cue

If the desired output instead should be line drawings, an approximation can be generated as follows:

5c.1. The mask representing the visible part of the polygon is projected into the Draw-buffer with the background colour, erasing anything drawn previously which is obscured by this polygon (write_mask).

5c.2. The Scratch-buffer is cleared (write_rectangle).

5c.3. The polygon edge is drawn in the Scratch-buffer (line_Interpolate). To achieve a depth cue effect, the line intensity can be computed as a function of the vertex z-coordinate.

5c.4. The invisible part of the polygon edge is cleared using the mask previously generated in the Z-buffer (write_mask).

5c.5. The remaining part of the polygon edge is copied to the Draw-buffer (write_colour).
2.5. Transparency Simulation

A high-quality display of transparent and translucent surfaces is difficult with this approach. The best we can do is probably a simulation using a texture consisting of visible pixels with the desired colour, and transparent holes for seeing through.

This texture can mask away the transparent pixels after the initial (stage 1 above) z-coordinate interpolation in the Scratch-buffer (write_texture). This will shield the transparent part of the polygon both from updating the Z-buffer and from the Draw-buffer.
3 SHADOW CASTING

Shadow casting has normally been considered too complicated a phenomenon to include in fast image generation systems. However, having a z-buffer and a flexible set of RasterOps, this is not necessarily so.

The algorithm below uses a "shadow volume" approach [CROW77, BERGERON86]. After computing the z-coordinates of the scene in the Z-buffer, it computes for each light source which parts of the visible scene are inside the "shadow volume" cast by each polygon, and thus not affected by this light source.

![Diagram of shadow casting](image)

**Fig. 3.1.**

The shadow volume cast by a polygon A is clipped by the hither (Zh) and yon (Zy) boundaries, and intersects a polygon B, casting a shadow A' on B.

The shadow casting algorithm goes as follows:

1. Run all polygons of the scene through stages 1-4 of the hidden-surface algorithm described above, accumulating the z-coordinates of the visible polygons in the Z-buffer.
2. For each light source do:

   2.1. For each polygon do:

      2.1.1. Compute the shadow volume cast by this polygon from the
current light source and intersect this with the clipping
volume (to avoid range problems).

      2.1.2. For each face of the clipped shadow volume do:

         2.1.2.1. Interpolate the z-coordinates across the polygon in
the Scratch-buffer (polygon_fill).

         2.1.2.2. The z-coordinates are then compared to the z-
coordinates in the Z-buffer (compare). "Shadow
volume"-faces facing away from the eye should have
z-coordinates smaller than the Z-buffer value, and
faces facing towards the eye should have z-
coordinates larger than the Z-buffer value in order for
the polygon to cast a shadow on the visible portion of
the scene.

         2.1.2.3. Masks for where the visible scene is outside the
"shadow-volume" for each "shadow volume"-face are
or'ed together (write_mask) and then inverted
(write_rectangle).

      2.1.3. This "obsured mask" is then or'ed together with the
"obsured masks" caused by shadow volumes from the other
polygons (write_mask).

2.2. The accumulated "obsured mask" can then be used in a third pass
over the polygon, where the visible polygons seen from the current
light source are redisplayed with a computed (higher) intensity.

If it is desired to have multiple light sources, it would be necessary to
compute the "obsured mask" for each and store them in separate planes, and
then regenerate the scene taking into account all combinations of the
"obsured masks". As this operations is O(n!), for n light sources, this is
feasible only for a very limited number of light sources (two - three).
The shadow casting process is illustrated on colour plates 2a-e.

If polygon adjacencies are known during rendering, the computation of shadow volumes can be done more effectively for entire objects, instead of per polygon. Only polygon (silhouette) edges where the adjacent polygons change from front-facing to back-facing, when seen from the light source, will imply a face of the shadow volume.

If the coefficient vector for a polygon is \( \gamma \) and the light source is situated at \( L (x, y, z) \), the sign of the dot-product

\[
d = [x, y, z, 1] \cdot \gamma
\]

will determine if the light source sees the front or back face of the polygon.

For light sources infinitely far away along a vector \( n = [x, y, z] \), the dot-product becomes:

\[
d = [x, y, z, 0] \cdot \gamma
\]
4 DIRECT RENDERING OF CSG

Inspired with the success of hidden surface rendering and shadow casting, the next obvious challenge for RasterOp algorithms was Constructive Solid Geometry (CSG).

CSG has been treated in-depth in many papers in recent years, and is a popular representation for ray-tracing scenes etc. [REQUICHA80, MUUSS86]

A CSG object is built from primitive solids like cubes, wedges, or cylinders. "Composite solids" can be built by adding, subtracting, or intersecting other solids. Composite and primitive solids can be translated, scaled and rotated.

Building a CSG object corresponds well with the traditional construction of objects in the mechanical industry, and is simpler than defining objects in terms of their limiting edges and faces.

4.1 Rendering CSG Objects

When considering the display of the objects resulting from these Boolean operations on a graphic display, we observe that the resulting visual surfaces must be in the set of bounding surfaces of the transformed primitive objects that take part in the Boolean operations.

A scene of CSG objects can be represented as a set of binary trees, where each tree represents a disjoint object. The binary CSG tree can be defined by (after [REQUICHA80]):

\[
<\text{CSG tree}> ::=
\begin{align*}
    & \text{<primitive leaf>} \\
    | & <\text{CSG tree}> <\text{set-operator node}> <\text{CSG tree}> \\
    | & <\text{CSG tree}> <\text{transformation node}> <\text{transformation}> \\
    | & <\text{CSG tree}> <\text{attribute node}> <\text{rendering attributes}> \\
\end{align*}
\]

For presentation, graphic attributes such as colour, surface reflection etc. can be attached to each tree or subtree. Rendering attributes are otherwise inherited from above.
The figure below shows the representation of a simple bracket formed by adding a tube on top of a cube and then subtracting another cube. (Actually, the tube is formed by subtracting a cylinder with a small radii from a cylinder with a larger radii.)

![Diagram of a bracket](image)

Fig. 4.1.

a) CSG tree representing the solid "bracket" object shown in b). (Transformations are not shown.)

Obviously, the normal z-buffer algorithm is able to handle the visual relation between disjoint objects (and also if they are actually intersecting). Moreover, a normal z-buffer algorithm will take care of forming the union of two sub-objects. What remains is then how to be able to generate the visual surface of a sub-object formed by intersection or subtraction.

The tree structure is equivalent to a mixed expression, with binary operators add/union (+), intersection (*), and subtraction (-).

The bracket would be represented as follows:

\[(\text{Cube}1 + (\text{Cyl}1 - \text{Cyl}2)) - \text{Cube}2\]
We may simplify the problem by thinking of subtraction \((A - B)\) as an intersection with a negated object \((A * !B)\), where "!" is a monadic negation operator. The bracket representation becomes:

\[(\text{Cube}1 + (\text{Cyl}1 * !\text{Cyl}2)) * !\text{Cube}2\]

Luckily, these volume operators obey the same rules as their counterparts in traditional Boolean algebra [TAUB82]. Thus, using the techniques from Boolean algebra (including De Morgan's theorem), allows us to rearrange a complicated expression into a canonical form. The canonical forms used in electronic circuit design are sums of products and products of sums. For CSG, sums of products canonical form is most appropriate, as we shall see.

So, we try to rearrange the volume expression for our bracket:

\[(\text{Cube}1 * !\text{Cube}2) + ((\text{Cyl}1 * !\text{Cyl}2) * !\text{Cube}2) \quad \{ \text{distributive \#-operator} \}\]

and:

\[(\text{Cube}1 * !\text{Cube}2) + (\text{Cyl}1 * !\text{Cyl}2 * !\text{Cube}2) \quad \{ \text{associative \#-operator} \}\]

When rendering, we are only interested in displaying the surface of the object.

If \(F(A)\) is the visual surface of \(A\) from a given viewpoint, and \(A\) is a CSG object represented as a union (sum) of two objects, \(F(A)\) can be expressed as

\[F(\text{Union}(B, C)) = Z\text{buffer}(Z\text{buffer}(\text{Empty}, F(B)), F(C))\]

that is, \(F(A)\) is generated by applying the z-buffer algorithm to an empty z-buffer and \(F(B)\) and then applying the z-buffer algorithm to the accumulated z-buffer and \(F(C)\).
Fig. 4.2.
The visual result of subtracting a box C from a box B.

If A is a CSG object represented as an intersection of B and C, F(A) can informally be expressed as:

\[
F(\text{Intersect}(B, C)) = \text{Zbuffer}(\text{Zbuffer(Empty, Csg\_intersect}(F(B), C)), \\
\text{Csg\_intersect}(F(C), B)),
\]

that is, the part of the visual surface of B which is inside C and the part of the visual surface of C which is inside B.

Similarly, if A is formed by subtracting C from B, F(A) can be expressed as:

\[
F(\text{Intersect}(B, \text{Inv}(C))) = \text{Zbuffer}(\text{Zbuffer(Empty, Csg\_intersect}(F(B), \\
\text{Inv}(C)), \text{Csg\_intersect}(F(\text{Inv}(C)), B)).
\]

This starts to look like a suitable task for a RasterOp processor with simple logic operations. What remains is to be able to find the part of F(B) which is inside/outside C (the Csg\_intersect - operation).

For simplicity, we limit all our primitive objects to be convex. This means:

a) Only surface parts facing towards the viewpoint contribute to F(B) (approximating the surface of B with plane polygons if desired).
b) For Csg_intersect(F(B), C) to output a point \((x, y, Z_b(x, y))\) on \(F(B)\), \(Z_T(x, y) < Z_b(x, y) < Z_f(x, y)\); that is, the z-coordinate of the visual candidate point must be smaller than (behind) any z-coordinates of front-facing surfaces of C at \((x, y)\), and larger than (in front of) any back-facing surfaces of C at \((x, y)\).

c) \(F(\text{Inv}(B))\) can be generated by outputting all back-facing polygons in B.

d) For Csg_intersect(F(B), Inv(C)) to output a point \((x, y, Z_b(x, y))\) on \(F(B)\), \(Z_b(x, y) > Z_f(x, y)\) or \(Z_b(x, y) < Z_T(x, y)\); that is, the z-coordinate of the visual candidate point must be larger than (in front of) any z-coordinates of front-facing surfaces of C at \((x, y)\), or smaller than (behind) any back-facing surfaces of C at \((x, y)\).
Fig. 4.3.
The basic operations when subtracting C from B (ref. Fig. 4.1); a) and c) show the surface of B with the negated volume C subtracted. e) shows the result of this when accumulated in the Z-buffer. b) and d) show the surface of the negated C with B subtracted. f) shows the result of this when accumulated into the Z-buffer†.

In our RasterOp-based system, to handle the necessary computations above, we need a buffer in addition to the Z-buffer (and Draw-buffer and Scratch-buffer) used for the z-buffer algorithm. This new buffer will temporarily store

† The Ci-function is identical to the Csg_intersect function described in the text.
z-coordinates from the primitive objects while they are compared with the front and rear limitations of the intersecting primitive objects. This buffer is called the CSG-buffer in the following. In addition, we need two single bitmasks: one for accumulating the area to be excluded by intersections and subtractions, and one for accumulating the total object area. The bitmasks are named outside-mask and coverage-mask, respectively. The coverage-mask is placed above the Scratch-buffer, to allow updates simultaneously with scratch drawing. The outside-mask is placed above the Z-buffer.

4.2 Algorithm for Drawing Convex CSG Objects

The following is an algorithm for drawing CSG objects composed of convex primitive objects:
1. Rearrange CSG expression to sum of products/union of intersections canonical form
   \[(a_1 * a_2 * ... * a_m + b_1 * b_2 * ... * b_n + ... + k_1 * k_2 * ... * k_q)\].

2. For each intersection product \((a_1 * a_2 * ... * a_n)\) do
   2.1 For each intersection member \((a_i)\) do
       Output z-coordinates of visible surface to CSG-buffer:
       2.1.1 Approximate surface with polygons.
       2.1.2 For each polygon do
           Transform it.
           if (front-facing and not negated) or (back-facing and negated)
               Interpolate z-coordinates into CSG-buffer.
       2.1.3 Clear outside-mask.
   2.1.4 For each other member of intersection \((a_1, a_2, ..., a_j-1, a_j+1, ..., a_n)\) do
       Clear coverage-mask.
       Approximate surface with polygons.
       For each polygon do
           Transform it.
           Interpolate z-coordinates into Scratch-buffer.
           and accumulate polygon area to coverage-mask.
           if (non-negated object - intersection)
               Compare \(\times\) z-coordinates in
               Scratch-buffer with z-coordinates
               in CSG-buffer and OR result to
               outside-mask plane.
           else /* back-facing */
               Compare \(<\) z-coordinates in
               Scratch-buffer with z-coordinates
               in CSG-buffer and OR result to
               outside-mask plane.
           else /* negated object - subtraction */
               Compare \(\geq\) z-coordinates in
               Scratch-buffer with z-coordinates
               in CSG-buffer and OR result to
               outside-mask plane.
           else /*back-facing */
               Compare \(\leq\) z-coordinates in
               Scratch-buffer with z-coordinates
               in CSG-buffer and OR result to
               outside-mask plane.
       if (intersection)
           Invert coverage-mask.
           Copy (or-mode) outside-mask to coverage-mask
       else /* subtraction */
           Copy (and-not-mode) outside-mask to coverage-mask.
       Clear CSG-buffer where coverage-mask is set
       (write_mask) - accumulatively reduces the visible
       part of \(a_j\).

2.1.5 The outside-mask is set to 1.
2.1.6 The Z-buffer is updated from the non-zero CSG-buffer
       (copy_colour), and the outside-mask is cleared in the same operation
       to form an "obscured"-mask where \(a_j\) is not visible.
2.1.7 The front-facing polygons of \(a_j\) can then be displayed with
       the proper shading rules, taking into account the "obscured" mask.
The reason for the $\leq$ and $\geq$ for subtraction, $<$ and $>$ for intersection, is to make sure that $F(\text{Intersect}(A, \text{inv}(A)))$ is empty, while $F(\text{Intersect}(A, A))$ equals $F(A)$.

For efficiency, object extents are heavily used to eliminate obviously empty intersections, and limit the actual areas of raster operations.

The above algorithm is robust as long as there are no holes between polygons. Pixels on edges and at vertices may be drawn once or several times without causing problems.

The process of subtracting one cube from another is illustrated on colour plates 4a-b.

4.3 Handling Non-Convex Primitives.

Certain non-convex primitives are hard to ignore for a CSG system. For instance, a torus is a good primitive for making certain types of fillets etc. A torus could be handled by decomposing it into a sum (union) of parts which can be approximated by a convex object; however, it would result in an abundance of products in the SOP (Sum Of Products).

To avoid this, we can modify the algorithm to be able to handle non-convex volumes, as well. This is equivalent to the problem of writing an algorithm to fill general polygons instead of just convex polygons. For the interior of general, possibly self-intersecting polygons, the most popular definition is based on Jordan's Curve Theorem. This involves counting the number of intersections with the polygon boundary when a line is drawn from a point $(x, y)$ towards infinity in a chosen direction; if the number is odd, the point $(x, y)$ is inside the polygon. Likewise, efficient algorithms for filling polygons are often based on similar parity techniques [ACKLAND81, DUNLAVEY83].

So, a parity-based approach is also chosen for our CSG intersections.
In the notation used above, \( \text{intersect}(F(A), B) \) can be generated by taking each single-valued\(^\dagger\) part of \( F(A) \), interpolating the \( Z \)-values into the CSG-buffer \( (Z_a(x, y)) \), and comparing the \( Z \)-values of each polygon on the surface of \( B \), and at each point \((x, y)\) keeping track of the number of times (modulo 2) the polygons of \( B \) intersect a ray from \((x, y, Z_a(x, y))\) to \((x, y, \infty)\).

Two bit-masks are needed for this algorithm along with the three buffer areas: \textit{parity-mask} which finds if a ray from the object being displayed towards the user intersects the comparing object an even or an odd number of times, and \textit{invisible-mask} which accumulates the invisible parts from each of the intersection computations.

Fig. 4.4.
Building of the \textit{parity-mask} as the concave polyhedron is scan-converted.

The figure above shows a section in the \( XZ \)-plane of a face \( F1 \) and a concave polyhedron (section 1-2-3-4-5). Below the section is shown the bitmask resulting from the depth comparison operation performed between the face \( F1 \) and each face of the intersecting polyhedron. The resulting parity mask is shown on the bottom line, indicating which parts of \( F1 \) are inside the polyhedron.

\( \dagger \) \( F(A) \) must be divided into parts which are singlevalued in \( z = f(x, y) \).
4.4 Algorithm for Drawing Concave CSG Objects

The following is an algorithm for drawing CSG objects composed of convex and concave primitive objects:

1. Rearrange CSG expression to sum of products/union of intersections canonical form \((a_1 * a_2 * ... * a_m + b_1 * b_2 * ... * b_n + ... + k_1 * k_2 * ... * k_r)\).

2. For each intersection product \((a_1 * a_2 * ... * a_n)\) do
   For each intersection member \((a_j)\) do
   Approximate surface with polygons.
   while not whole potentially visible surface drawn do
   /* Output z-coordinates of visible surface to CSG-buffer: */
   For each front-facing polygon in single-valued surface
   part do
   Transform it.
   Interpolate z-coordinates into CSG-buffer.
   Clear invisible-mask.
   For each other member of intersection
   \((a_1, a_2, ..., a_{j-1}, a_{j+1}, ..., a_n)\) do
   Set parity-mask to all 0 for intersection,
   all 1 for subtraction.
   Approximate surface with polygons.
   For each polygon do
   Transform it.
   Interpolate z-coordinates into
   Scratch-buffer.
   Compare z-coordinates in Scratch-
   buffer (<) with z-coordinates in
   CSG-buffer and XOR result to parity-mask
   plane.
   Set invisible-mask where parity-mask is set
   (write_mask) - accumulatively increasing the
   invisible part of \(a_j\).
   Compare z-coordinates in CSG with Z-buffer, OR-ing result
   to invisible-mask.
   Use invisible-mask to clear the CSG-buffer (write_mask).
   Update Z-buffer from visible part of CSG-buffer
   (copy colour).

   The front-facing polygons of \(a_j\) can then be displayed with
   the proper shading rules, taking into account
   invisible-mask.

The algorithm requires that the polygons on the intersecting object surface are drawn carefully, so that each pixel is handled an even number of times for
front and back surfaces. This is necessary to toggle the parity-mask bits correctly.

Whereas the implemented primitives are the traditional (polyhedron, cone, sphere), the algorithm should be able to cope with, for instance, objects bounded by B-spline surfaces. In this case the algorithm requires that surfaces are non-self-intersecting, and that it is possible to segment the surface into singly z-valued patches. The latter should be possible by common spline surface subdivision techniques such as the Oslo algorithm [COHEN80].

The quality of the resulting image relies on the quality of the polygon fill algorithm. The following algorithm satisfies the requirements both for z-buffer rendering and for the convex and concave versions of the CSG-display algorithms. The algorithm is limited to polygons that are convex or (weaker) that have no more than two intersections with any scan line.

Implemented in C using fractional 32-bits arithmetic, the algorithm is also efficient. Each polygon pixel is updated only once. Only two fixed-point adds and one shift operation is necessary for each pixel.

The polygon drawing algorithm:

Start in topmost vertex and go round polygon.
For each polygon edge \((x_0, y_0)\) to \((x_1, y_1)\)
do
  if upwards-going
    swap endpoints
  Let \(\Delta x = (x_1 - x_0) / (y_1 - y_0)\)
  For each scanline \(y\) (from \(y_0\) to \(y_1\)) do:
    \(x = \text{floor}(x_0 + \Delta x \times (y - y_0) + 0.5)\)
  if downwards-going
    store \(x\) in \(\text{scan}\_\text{table}[y]\)
  else
    draw scan from \(x\) to \(\text{scan}\_\text{table}[y] - 1\) (inclusive).

4.5 Touching Objects

Though the algorithms are stable, touching objects can cause trouble because of round-off errors. If we, for instance, take the intersection of two cubes which stand on top of each other, this can result in output of "dangling"
polygon segments in the touching surface, unless the vertices of the touching faces are identical.

To avoid this problem, the geometry can be given small offset values, which should be larger than the round-off error in the z-buffer calculations.
5 ADAPTABILITY

5.1 An Improved Set of RasterOps

By making some slight modifications to the set of raster operations, the scratch buffer may be eliminated. We would have to replace the simple scan conversion, compare, and copy mask operation with a more complex \texttt{interpolate\_polygon\_and\_compare\_with\_z-buffer}-operation. In the Sun-3 implementation, this would mean that the working set of the program could be reduced from approximately 4 Mb to 3 Mb.

5.2 Z-Buffer Resolution Problems.

The ICAN Raster Workstation works with 12 bit-planes, allowing 11 bits for the depth resolution of the image. This is an absolute minimum which may be sufficient for single objects or simple scenes. For images with a higher depth resolution, say several complicated objects with space in-between, 16 - 32 bits would be recommended for the Z-buffer.

However, using hither and yon clipping on the scene prior to the z-buffer algorithm may improve the depth resolution somewhat [ZACHRISEN89c].

One of the most noticeable defects is where adjacent faces join on silhouette edges\textsuperscript{†}. Here, the probability that the edge of the back-facing polygon will show is 50\%. However, imposing a constraint on surface normal directions makes it possible to remove all back-facing polygons prior to sending them to the z-buffer algorithm. Another cure for this is to do the z-coordinate sampling offset 0.5 pixels from the edge [ZACHRISEN89a]. This can easily be implemented in the polygon scan algorithm shown in Ch. 4.4.

5.3 A Scan Line Algorithm

The CSG algorithm can clearly be used efficiently in a scan line z-buffer

\textsuperscript{†} Because the front-facing polygon and the back-facing polygon share the same edge, z-coordinates are identical, and improved depth resolution will not cure this problem.
setting as well, requiring one Z-buffer, one CSG-buffer and two mask bits per scan line cell.

This approach has been followed in a diploma project under the author's guidance [MELEN87]. The implementation of the CSG-buffer algorithm was carried out on a network of Transputers, with parallel evaluation of the SOP's products. The implementation was coded in OCCAM under TDS. With four parallel T414 processors for product evaluation, the system gave reasonable (near interactive) response to transformations of objects in the tree; a cylindrical hole can be "drilled" into a solid block. This clearly points towards applications in real-time simulation of machining operations.

In a subsequent student project [FÁBERG88], the system was given a more "user-friendly" and "open ended" interface, with menu control and provisions for importing data structures from other systems.

5.4 RasterOp Software Architecture

Though in our case the RasterOp instruction set is the interface between a general purpose workstation and a microprogrammed raster sub-system, the same interface may be utilized between general graphics algorithms running on a main application processor (whatever that may be) and a specialized graphics subsystem realized with its own graphical processor. The latter would be based on a subroutine implementation of the RasterOps on perhaps just another Motorola 680xx processor, or on a general processor with a comprehensive instruction set for frame buffer operations like the Texas graphics processor [SHORT86], or on a programmable DSP (Digital Signal Processor).

5.5 A RasterOp Server

The flexibility provided by this instruction set makes it a good basis for a rendering processor, whether this processor is hardwired, microprogrammed, or programmed. The latter option is used in modern network-distributed graphic systems, where different client programs send graphic commands to a workstation server for display. An example of this is the X Window System [SCHIEFLER86]. In X the server manages shared resources, like the screen area, and directs output from each client program to
the allocated windows. The concepts and functionality in X is defined by the X protocol [SCHEIFLER88].

X provides so-called Pixmaps, which are off-screen raster buffers, but which have the same graphic functionality as screen windows. Compared with the system described in this report, X provides polygon drawing, tiling and raster operations with 16 different logical combinations. Line drawing with interpolated colour indices is not available, neither is arithmetic comparisons between images. However, X provides a standard way of making additions to the protocol and the server, and it would be a manageable job to upgrade the functionality of the X server to work as a full-blown renderer for hidden-surface graphics and CSG.

There also exists a proposed extension to X for handling three-dimensional graphics, called PEX or PHIGS-on-X [PEX88a, PEX88b]. This combines the functionality of the PHIGS+ graphic standard proposal with the functionality of a windowing system. However, as opposed to the renderer-server sketched above, the PEX server is a closed rendering environment providing hidden-surface display of a fixed number of primitives and a fixed (but large) menu of lighting and shading techniques.
6 THE RENDERING ENVIRONMENT

The z-buffer algorithms are part of a larger system for rendering called GAS (GAS stands for Graphics for All Seasons) [ZACHRISEN89a]. GAS contains a whole menu of algorithms for rendering. All algorithms use a common data structure for representing the various object types in the scene.

The GAS development is also an exercise in object-oriented programming of large graphics systems. In excess of 70 classes of objects have been defined in a class hierarchy. For programming, a class pre-processor has been used. The output of this is standard "C"-programs. The system has been running under UNIX on a VAX and on a 68000-based workstation.
7 SOME PERFORMANCE MEASUREMENTS

7.1 Measurements on the ICAN Raster Workstation

To give the reader some indication of the performance of the system, a few timing measurements are given here. No exhaustive analysis of the bottlenecks in the rendering system has been carried out. The system as it stands is limited by its abilities to handle transformations with floating point coordinates. This is because no floating point processor is present, and floating point arithmetic is handled by an emulation package.

Table 7.1.
Comparable timing data for MOVIE-BYU and GAS on an ICAN Raster Workstation (Motorola 68010 w/o FPP). The scene consists of a robot consisting of 139 polygons rendered on a 512 x 400 screen area.

<table>
<thead>
<tr>
<th></th>
<th>MOVIE-BYU</th>
<th>GAS</th>
<th>GAS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Z-Buffer</td>
<td></td>
<td>BSP</td>
</tr>
<tr>
<td>1:</td>
<td>Initialize (file input etc.)</td>
<td>48</td>
<td>16</td>
</tr>
<tr>
<td>2:</td>
<td>Per frame</td>
<td>76</td>
<td>1.71</td>
</tr>
</tbody>
</table>

The measurements show that on the ICAN Raster Workstation, both the GAS z-buffer RasterOp-based algorithm, and the GAS rendering algorithm based on BSP (Binary Separating Planes), are significantly faster than the scan line algorithm used in MOVIE-BYU†. The higher setup time for the BSP algorithm compared with the z-buffer algorithm is due to the building of the BSP tree. More discussion on the different types of algorithms are found in volume 1[ZACHRISEN89]. The significant difference between Movie-BYU and the other algorithm is probably because Movie-BYU uses a lot of floating point arithmetic which is slow on this workstation configuration.

7.2 Measurements on the Sun-3/470

Due to a disk breakdown on the ICAN Raster workstation, the system was reimplemented on a Sun-3 workstation. The Sun-3 does not have any sort of raster processor. All raster operations are thus implemented in software,

† A description of the rendering system MOVIE-BYU is found in volume 1.
working on memory raster matrices. The evaluated implementation used 512x512 matrices of 32 bits numbers. This provides sufficient depth resolution for elaborate constructions. The final image is transferred from the Draw-buffer to the Sun's CG4 frame buffer by means of the Moviebox device interface and dithering software.

Because of the large buffers involved, four buffers of 1 Mb each, the system caused major swapping operations when run on a regular Sun-3/50 with 4 Mb. Because of this, the testing and timing were instead carried out on a Sun-3/470 with 16 Mb. The tested workstation is equipped with Motorola 68030 processor and Motorola 68882 FPP. If it had been crucial to run the system on a 4 Mb workstation, the raster buffers could have been limited to 16 bit numbers rather than 32 bit numbers. This would still provide better depth resolution than the ICAN Raster implementation.

The timings below are the CPU time measured by the "CLOCK(3C)" system call [SUN OS].

7.2.1. Z-Buffer Algorithm

Table. 7.2.
Comparable timing data for Moviebox and GAS Z-Buffer algorithm on a Sun-3/470 with 68882 FPP. Note that Moviebox timings include image output to a run-length-encoded file, while GAS timings only include image generation in the Draw-Buffer.

<table>
<thead>
<tr>
<th></th>
<th>#polygons</th>
<th>depth</th>
<th>Moviebox</th>
<th>GAS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>complexity</td>
<td>Sun-3/470</td>
<td>Sun-3/470</td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>Robot</td>
<td>62</td>
<td>0.58</td>
<td></td>
</tr>
<tr>
<td>2a:</td>
<td>Boxes 4x4x14</td>
<td>5376</td>
<td>0.0912</td>
<td>1.31</td>
</tr>
<tr>
<td>2b:</td>
<td>(zoom 2)</td>
<td>5376</td>
<td>0.366</td>
<td>1.66</td>
</tr>
<tr>
<td>2c:</td>
<td>(zoom 3.4)</td>
<td>5376</td>
<td>1.0147</td>
<td>2.17</td>
</tr>
<tr>
<td>3a:</td>
<td>Camshaft (w/o. culi)</td>
<td>1806</td>
<td>0.2546</td>
<td>2.27</td>
</tr>
<tr>
<td>3b:</td>
<td>(zoomed 3)</td>
<td>1806</td>
<td>1.3098</td>
<td>8.61</td>
</tr>
</tbody>
</table>

The scenes are:

1. The first scene consists of the same robot as was rendered on the ICAN Raster Workstation in Ch. 7. 1.
2a-b. This scene consists of 4x4x14 hexahedrons. This is a scene with a large degree of overlapping polygons. It has been measured at different zooming factors.

3a-b. This scene consists of a camshaft. The scene has been used for measurements on Moviebox on other workstations. It has been measured in two different zooming factors.

When the image output time included in the Moviebox measurements is subtracted, the measurements indicate that the RasterOp-based z-buffer algorithm is approximately twice as slow as the more dedicated z-buffer algorithm in Moviebox. This overhead in the RasterOp-based algorithm is obvious as the raster operations work on axis-aligned rectangular extents of the polygons.

7.2.2. Shadow Casting Algorithm

Table 7.3.
Timing data for the shadow casting algorithm on a Sun-3/470. Images were generated in 512 x 512 resolution using flat (cosine) shading. Time includes the image generation in the Draw-Buffer, but excludes the dithering and copying of the draw-buffer into the CG4 frame buffer.

<table>
<thead>
<tr>
<th></th>
<th>polys</th>
<th>polypix</th>
<th>#weighted</th>
<th>RasterOp</th>
<th>GAS sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a:</td>
<td>House w/o shadow</td>
<td>9</td>
<td>88k</td>
<td>2102k</td>
<td>1.08</td>
</tr>
<tr>
<td>1b:</td>
<td>House with shadow</td>
<td>73</td>
<td>796k</td>
<td>22343k</td>
<td>14.16</td>
</tr>
</tbody>
</table>

The test scene consists of a house that is shown on colour plate 3a. The house was rendered with a normal z-buffer algorithm and with the shadow casting algorithm.

The computed time for the shadow casting algorithm is approximately ten times that of the simple z-buffer algorithm. The cost of shadow casting can be described as follows. Each polygon in the scene will generate 6 new polygons which bounds the shadow volume. Each bounding polygon will be somewhat larger than the polygons of the original scene. Moreover, the scene
is rendered twice; once with only ambient lighting, and once with normal lighting, but with shadowed areas inhibited.

### 7.2.3. CSG Algorithm

Table 7.4.
Timing data for the CSG algorithm on a Sun-3/470. Images were generated in 512 x 512 resolution using flat (cosine) shading. Time includes the image generation in the Draw-Buffer, but excludes the dithering and copying of the draw-buffer into the CG4 frame buffer.

<table>
<thead>
<tr>
<th></th>
<th>#poly</th>
<th>#polypix</th>
<th>#weighted</th>
<th>RasterOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a:</td>
<td>Cube subtraction</td>
<td>23</td>
<td>237k</td>
<td>2366k</td>
</tr>
<tr>
<td>1b:</td>
<td>Cube subtraction (x 1.5)</td>
<td>23</td>
<td>537k</td>
<td>4266k</td>
</tr>
<tr>
<td>2:</td>
<td>Bracket</td>
<td>186</td>
<td>1322k</td>
<td>6600k</td>
</tr>
<tr>
<td>3a:</td>
<td>Unijoint</td>
<td>1377</td>
<td>2110k</td>
<td>17256k</td>
</tr>
<tr>
<td>3b:</td>
<td>Unijoint w. cutaway</td>
<td>1948</td>
<td>4641k</td>
<td>25521k</td>
</tr>
</tbody>
</table>

The examples are:

1a: A smaller cube subtracted from a larger cube (base of the bracket). This is shown on colour plate 4a and 4b.

1b: The same object scaled by a factor of 1.5.

2: The bracket discussed in Ch 4.1. This is shown on colour plate 5.

3a: A universal joint for the driving axle of a car. This is a rather complex part. It is shown on colour plate 6a.

3b: The same universal joint with a cutaway to show the innards. This is shown on colour plate 6b.

The data listed in the tables are the number of polygons output, the total area of the polygons output, a weighted number of the raster operations involved, and the rendering time in seconds.
The weighted RasterOp count weights each raster operation by its relative complexity (= number of memory accesses involved). Thus a rectangle fill has weight 1, while a copy operation has weight 2, and a compare operation has weight 3.

The measurements indicate that the raster operations dominate the polygon scan conversion for the scenes above. The execution time seems to grow proportionally with the weighted raster operation area. On the Sun-3/470, the execution rate seems to be $= 1\,000\,000$ raster operations per second.

The bracket on example 2 above, was also tested in the scan line implementation of the algorithm. The measured generation times were 13.1 seconds on a single T414 Transputer and 3.6 seconds on a configuration involving five T414 Transputers [MELEN88].
8 RELATED WORK

Rossignac and Requicha [ROSSIGNAC86] from the University of Rochester (a well-known contributor to the development of solid modelling) report on an algorithm they describe as the first z-buffer algorithm for display of Constructive Solid Geometry. Their approach is different from the one described here. They send each face of each primitive of each solid to a z-buffer renderer. For each point which is visible according to the z-buffer test, they do a classification using the CSG-tree structure and update the z-buffer and intensity buffer if the point is on the actual surface of the solid. For speed-up they compute so-called I-zones of each primitive. The potentially visible part of the surface of a primitive is the intersection of the surface with its I-zone. The performance of their algorithm is compared with that of a ray casting algorithm, and is reported to be in the range 1 to 25 times that of the ray casting algorithm. Compared with ray casting, the z-buffer algorithm excels when rendering solids with complex surfaces which project to small screen areas. No absolute speed measurements are reported.

While most of the research community has been busy making nice images with ray-tracing algorithms and simple pixel-addressable frame-buffers, some work has been put into the construction of more intelligent frame buffers. The most well-known of these efforts are the Pixel-planes [FUCHS81] and Pixel-powers [FUCHS85] systems developed by Fuchs et al. at the University of North-Carolina. Pixel-planes is constructed with very deep pixels (32 or 72 bits) giving space to the depth coordinate and the colour value and two mask bits. Algorithms for z-buffer hidden-surface removal and CSG-rendering on their system have been developed [FUCHS85, JANSEN86].

In a recent paper, Fournier and Fussell [FOURNIER88] give a more formal treatment of operations on intelligent frame buffers in the context of a pixel automaton. They categorize different intelligent frame buffers according to their storage capability and their operations and show that the visible-surface problem can be solved with a "FB_{11p}", a frame buffer with one register, one flag and the capability to compute predicates (compare values), and that a two-register, one-flag frame buffer with more general operators, a "FB_{21f}", is needed to compute convex intersection of half spaces.
There have also been some recent papers on the formal treatment of raster operations; Willis and Watters propose a general three-input RasterOp for colour displays [WILLIS88], and Fiume presents a formal treatment of bitmap operations and rasterization in the context of monochrome displays [FIUME87].
9 CONCLUSIONS

Z-buffer algorithms are powerful tools for obtaining near-real-time display of shaded images on engineering workstations.

The extended RasterOp instruction set (combined with appropriate scan conversion primitives), provides a very flexible basis for various picture generation algorithms. The price for this flexibility is a slightly reduced speed compared with a specialized microprogrammed z-buffer algorithm with uniform colour polygon output.

Using RasterOps as the basis for rendering algorithms also provides us with window manager compatibility. Moreover, avoiding the use of a dedicated z-buffer means that other tasks running on the engineering workstation may benefit from added memory resources whenever the rendering process is not running.

By employing the appropriate techniques, good quality displays can be obtained even with very limited depth and colour resolution on the displays.

Though in our case the RasterOp interface is implemented as microprogrammed instructions for a bit-slice processor, the algorithms and software architecture may be relevant also for distributed graphic environments.
40 Volume 2 - Rendering on a RasterOp Processor
APPENDIX A. COLOUR PLATES

The colour illustrations in this volume are all produced as photographs taken from the face of the Hitachi monitor of a Sun-4/110 workstation with CG4 frame buffer. The frame buffer is limited to 256 simultaneous colours. For simplicity, 3x8 bits RGB colours were dithered to 8 x 8 x 4 levels of red, green and blue, respectively before output. All single images are rendered as 512x512 pixels, as this is the default size of the GAS buffers.

The photographs were captured on 100 ASA colour print film, using a single-lens reflex camera with 50 mm lens, 1/2 second, f 8. In the final printed report, photos have been copied on a Canon Colour Copier.

Plate 1.
Transparency is simulated by dot pattern in these images from the human body. The upper, left image show an oblique skin, while the skin transparency factor is varied to 0.8, 0.5, and 0.2 on the other images. The model has been reconstructed from tomograph images. (Courtesy CART project, O. H. Kristensen.)

The following colour plates illustrate the workings of the shadow casting algorithm.

Plate 2a.
To start with, the scene is drawn taking into account the ambient light only. The depth coordinates are stored in the Z-buffer. This plate shows a shaded image of the contents of the Z-buffer after the initial pass of the shadow casting algorithm. Higher z-values are shown with brighter colour.

Plate 2b.
The shadow volume from the triangle is shown in the Z-buffer overlay plane. The overlay plane is shown as a red, semi-transparent overlay.

Plate 2c.
The comparison of the Z-buffer with the shadow volume, causes the projection of the triangle on the surface to be found to be inside the shadow volume. The projected triangle is
disabled for further output by setting "highvalues". "Highvalues" are shown with bright green colour.

Plate 2d.
The scene is output taking into account ambient lighting and the light from the point source. Because the projected triangle has been disabled in the Z-buffer, this is not output and will have the colour resulting from only the ambient illumination.

Plate 2e.
By repeating the shadow volume process, the algorithm is able to show the illumination effect of multiple light sources.

Plate 3a.
A simple model of a house with shadow casting effects added.

Plate 3b.
The same house with the light coming from a different direction.

The following colour plates illustrate the workings of the CSG rendering algorithm. The demonstration object is simply the subtraction of a small cube from a larger cube.

Plate 4a.
In the first part of the CSG rendering process, the small cube is subtracted from the surface of the large cube. The top, left image shows the CSG-buffer after the depth coordinates of the visible part of the large cube have been interpolated. The small cube's depth coordinates are compared, and a mask is formed for the part of the surface that is inside the small cube. This is shown as red overlay in lower, left image. This mask is removed from the CSG-buffer (upper, right image). The remaining z-coordinates are copied into the Z-buffer, and the non-masked surface is drawn in the Draw-buffer (lower, right image).
Plate 4b.
These images show the rest of the process; the "inner" surface of the smaller cube is output and intersected with the large cube. The result is added to the Z-buffer and the Draw-buffer. The resulting Draw-buffer is shown in the lower, right image.

Plate 5.
This is a "bracket" constructed from two concentric cylinders and the two cubes from the example above.

Plate 6a.
This is a more complex part; a model of a universal joint formed by several intersections and subtractions.

Plate 6b.
By subtracting a cube from the universal joint, a cutaway view can be obtained. The "solidness" of the model is clearly illustrated.
APPENDIX B. REFERENCES


ICAN81 "ICAN Raster Workstation," Product sheet, ICAN a/s, Moloveien 1, N-3191 Horten.


PHONG75  Phong, B. T., "Illumination for Computer Generated Images," Communications of ACM 18, June 1975, pp. 311-317


Computer Graphics Rendering Techniques with an Emphasis on Performance Issues

Volume 3

Fast Algorithms for Polygon Clipping and Boxing
ABSTRACT OF VOLUME 3

A system for fast clipping of polygons has been designed. The system is well suited for complex environments. It has been designed based on an analysis of the clipping requirements for a "general purpose" rendering system.

The clipping algorithm is based on the same principle as the traditional Sutherland-Hodgman algorithm, passing the polygon through a pipeline of clipping stages. However, this algorithm makes use of external boxing information to reduce the necessary clipping to a minimum. Normally, polygons which intersect the window boundaries will be clipped just against one single boundary.

As the extent testing part of the clipper will be the critical component when traversing huge scenes, a new, highly efficient extent testing algorithm has been designed.

The clipping algorithm has a structure similar to a "push-down automaton", and is therefore referred to as the "Stacking Clipper". The algorithm is available for traditional 2-D clipping as well as 3-D clipping in homogeneous coordinates. Moreover, the algorithm can easily be adapted to general half-space clipping.

Tests show that the two-dimensional version of the Stacking Clipper has a speed comparable to the Liang-Barsky algorithm when extent testing is not used. When extent testing is used, this algorithm is 30% to 50% faster than the Liang-Barsky algorithm. The three-dimensional (homogeneous coordinate) version of the Stacking Clipper together with the new extent testing algorithm shows a speed improvement of approximately 50% compared with an implementation of the Sutherland-Hodgman polygon clipper.

This volume also discusses clipping for modelling purposes, and how clipping regions expressed as general Boolean expressions of halfspaces can be implemented as networks of clipping processors.
PREFACE TO VOLUME 3

The following paper is based on an implementation carried out in spring 1987, with subsequent improvements 1988/89.

The external funding for this project has come mainly from NTNF ("Parallell grafisk presentasjonsenhet").

The clipping algorithm is part of the Moviebox rendering system.

The paper was written using Microsoft Word (version 3), with most figures in-line in the text after being prepared with MacDraw (general drawings), or Excel (curves and tables). Some colour plates are referred to in the text. These have been collected in Appendix A.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT OF VOLUME 3</td>
<td>I</td>
</tr>
<tr>
<td>PREFACE TO VOLUME 3</td>
<td>II</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>III</td>
</tr>
<tr>
<td>1. INTRODUCTION TO CLIPPING</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Different Clipper Contexts</td>
<td>1</td>
</tr>
<tr>
<td>1.2. Line vs. Polygon vs. Volume Clipping</td>
<td>3</td>
</tr>
<tr>
<td>1.3. Existing Algorithms</td>
<td>4</td>
</tr>
<tr>
<td>1.3.1. The Sutherland-Cohen Line Clipping Algorithm</td>
<td>5</td>
</tr>
<tr>
<td>1.3.2. The Sutherland-Hodgman &quot;Reentrant Polygon Clipper&quot;</td>
<td>6</td>
</tr>
<tr>
<td>1.3.3. The Liang-Barsky Algorithm for Polygon Clipping</td>
<td>8</td>
</tr>
<tr>
<td>1.3.4. Other Clipping Algorithms</td>
<td>8</td>
</tr>
<tr>
<td>1.3.4.1 Liang-Barsky Line Clipping Algorithm</td>
<td>8</td>
</tr>
<tr>
<td>1.3.4.2 The Cyrus-Beck Line Clipping Algorithm</td>
<td>9</td>
</tr>
<tr>
<td>1.3.4.3 Weiler-Atherton Polygon Intersector</td>
<td>9</td>
</tr>
<tr>
<td>1.3.4.4 Burkert and Noll Algorithm</td>
<td>10</td>
</tr>
<tr>
<td>1.3.4.5 Kilgour's Boundary Chain Algorithm</td>
<td>10</td>
</tr>
<tr>
<td>1.3.5. Some Critical Remarks on the Algorithms</td>
<td>10</td>
</tr>
<tr>
<td>2. DESIGN</td>
<td>13</td>
</tr>
<tr>
<td>2.1. The Scene Geometry - Preconditions</td>
<td>13</td>
</tr>
<tr>
<td>2.2. Overall Design Decisions</td>
<td>14</td>
</tr>
<tr>
<td>2.3. Mathematics for Clipping</td>
<td>17</td>
</tr>
<tr>
<td>2.3.1. Two-Dimensional Halfspace Clipping</td>
<td>17</td>
</tr>
<tr>
<td>2.3.2. Three-Dimensional Halfspace Clipping</td>
<td>20</td>
</tr>
<tr>
<td>2.3.3. Clipping in Homogeneous Space</td>
<td>21</td>
</tr>
<tr>
<td>2.4. Extent Testing Algorithms</td>
<td>26</td>
</tr>
<tr>
<td>2.4.1. Where Does the Extent Come From?</td>
<td>26</td>
</tr>
<tr>
<td>2.4.2. Two- and Three-Dimensional Extent Testing Sequence</td>
<td>27</td>
</tr>
<tr>
<td>2.4.3. &quot;Brute Force&quot; Boxing</td>
<td>29</td>
</tr>
<tr>
<td>2.4.4. Michener's Algorithm</td>
<td>29</td>
</tr>
<tr>
<td>2.4.5. A Fast Boxing Algorithm</td>
<td>30</td>
</tr>
<tr>
<td>2.5. A Fast Clipping Algorithm</td>
<td>32</td>
</tr>
<tr>
<td>2.5.1. Finding the Output Vertices</td>
<td>33</td>
</tr>
<tr>
<td>2.5.2. Algorithm Pseudocode</td>
<td>34</td>
</tr>
<tr>
<td>2.5.3. Algorithm Walkthrough</td>
<td>37</td>
</tr>
<tr>
<td>2.5.4. Implementation Details</td>
<td>38</td>
</tr>
</tbody>
</table>
3. BENCHMARKS AND COMPARISONS ........................................ 40
   3.1. Two-Dimensional Benchmarks .................................... 41
         3.1.1. A Large, Concave Polygon ............................... 41
         3.1.2. Boxing Tests ........................................... 45
         3.1.3. Optimal Box Size ....................................... 49
   3.2. Three-Dimensional Benchmarks .................................. 51
   3.3. Generated Code Size ............................................ 56
4. ADAPTABILITY ............................................................. 58
   4.1. Clipping of Rational Parametric Curves ......................... 58
   4.2. General Halfspace Clipping .................................... 61
   4.3. Viewing Frustum Clipping in Object Space ..................... 62
   4.4. Modelling Clip .................................................. 63
   4.5. Building Divider Networks ...................................... 66
   4.6. Clipper Network From Boolean Expression ..................... 68
   4.7. Clipper Networks From Concave Polygons ....................... 72
5. CONCLUSIONS ............................................................ 73

APPENDIX A. COLOUR PLATES .............................................. A1
APPENDIX B. A FAST BOXING ALGORITHM ................................ B1
APPENDIX C. STACKING CLIPPER ........................................ C1
APPENDIX D. LIANG-BARSky ALGORITHM ................................ D1
APPENDIX E. SUTHERLAND-HODGMAN ALGORITHM ......................... E1
APPENDIX F. REFERENCES .................................................. F1
1. INTRODUCTION TO CLIPPING

One of the best principles of computational efficiency is to make use of the properties of the actual data to save time; that is, handle the general case, while providing fast code for the often occurring special cases. For instance, in rendering systems, polygons of general shape should be supported, while in most cases the bulk data will consist of convex polygons with three or four edges. In the case of clipping, it is important to realize that very few polygons will actually be intersected by the window boundaries: most will be well outside the window or fully inside the window.

1.1. Different Clipper Contexts

While a polygon clipping algorithm is basically used to find out which part of an input polygon is inside a clipping area or volume, the actual shapes of both the polygon and the clipping region will vary greatly depending on the environment of the clipper. Clipping algorithms can be used at several places in the rendering pipeline:

1. The traditional "Two-dimensional window clipper" clips polygons in two-dimensional object space to a rectangular window in \( \mathbb{R}^2 \) before mapping to viewport and device coordinates. This is the type of clipping algorithm traditionally found in graphic packages like GKS [ENDERLE84].

2. The "Manhattan Universe clipper" clips graphics primitives (lines / text / polygons) against a list of rectangles which corresponds to the output window with the areas of overlapping, higher priority windows subtracted. In the case of contemporary windowing systems like the X Window System [SCHIEFLER86] or the Macintosh Graphics [APPLE85], the clipping is carried out in \( \mathbb{I}^2 \).

3. The "Three-dimensional viewing volume clipper" clips polygons in three-dimensional object space before mapping to the viewport and device coordinates. A close relative of this is
2 Volume 3 - Fast Algorithms for Polygon Clipping and Boxing

the clipper that accepts polygons in four-dimensional homogeneous coordinate space and clips these to the viewing pyramid before projecting to three-dimensional space and mapping into viewport and device coordinates.

4. Modelling systems often require to make cut-away views of objects. This clipping region can be a simple halfspace or a general volume limited by higher order surfaces. In the PHIGS+ proposal [VERBECK87], the user can specify clipping on arbitrary halfspaces.

Many hidden surface algorithms use clipping algorithms in their effort to determine visibility in the output picture. This is particularly the case with the class of hidden surface algorithms operating in object space;

5. The "Cookie-cutter clipper" was designed by Weiler and Atherton [WEILER77] to subtract the area of one polygon from another polygon. This is used for computing the area to output when a polygon is obscured by another polygon which is nearer to the eye.

6. A "Screen subarea clipper" is used in the Warnock hidden surface algorithm [NEWMAN79]. When an area of the screen contains too many polygons, the area is subdivided into four adjacent subareas. The polygons in the area are clipped along the dividing lines and sorted into the subareas. This recursive screen area subdivision process can go on right down to pixel level.

7. To avoid aliasing effects, Catmull uses a special "Pixel clipper" in his hidden surface algorithm [CATMULL78]. This is used for pixels where several polygons contribute to the visual result. Catmull clips all candidate polygons to the pixel boundaries before computing the actual visual area.
1.2. Line vs. Polygon vs. Volume Clipping

A line clipping algorithm will input a set of line segments and output a new set of line segments where each line segment has been clipped against the clipping area boundaries.

A polygon clipping algorithm will clip each edge against the clipping area boundaries but will make sure that the output edges form a closed area, introducing new vertices into the output when needed.

Before the first "real" polygon clipping algorithms were designed, polygons were normally processed by line clippers and the output line segment were interconnected in a more or less "ad-hoc" manner. Special cases like when the polygon surrounds a corner in the clipping window, are not trivial to handle this way.

Fig. 1.1
A "non-trivial" polygon clipping situation occurs when the polygon surrounds window corners such that "turning points" must be inserted in the output polygon.

Today we see "solid modelling workstations" where solid models are represented with a set of boundary polygons. Clipping these against the view pyramid gives the desired result for the left, right, bottom and top boundaries. However, when intersecting the scene with "hither" clipping planes, the polygon representation will be revealed. The front polygons will be clipped
away, exposing the "shell" representation of the solid volume. This is illustrated on colour plate 1. (The "yon" clipping will be carried out on the back of the scene, away from the observer and will not create problems.) The boundary representation will similarly be exposed when freely chosen halfspace clipping is in use.

To prevent these "holes" from appearing, we will have to introduce new polygons to keep the polygon surface closed. This is a similar problem to the one with polygon closing described above, only in a higher dimension. An ad-hoc solution is to collect all edges introduced by the "hither" clipping boundary, creating new polygons with a properly sorted set of edges. This is a simple task for convex solids as is demonstrated in the GAS system [ZACHRISEN89b]. General, concave solids need more consideration.

When the clipping object is a solid represented by its boundary (a set of polygons), and the clipping volume is represented by its boundary planes, the output of the clipping algorithm should be the set of bounding polygons for the intersected volume. It can be shown that the boundary of the intersected volume equals the union of the boundary of the solid (clipped to the clipping boundaries), and the boundary of the clipping volume (clipped to the solid boundaries), and the coincident boundaries of the solid and the clipping volume that bound the same regions.

A general treatment of Boolean operations on n-dimensional objects, represented by its (n-1)-dimensional boundaries, can be found in Putnam and Subrahmanyam's paper [PUTNAM86]. The problem of Boolean operations on three-dimensional solids represented by a set of bounding polygons, is also discussed in [ZACHRISEN89b].

1.3. Existing Algorithms

A variety of clipping algorithms have been designed over the years.

There are three algorithms that will primarily be described here. These were selected because they are in general use, and because they have had a major impact on the described algorithm and are used for comparative performance tests.
Some other algorithms that have relevance to fast window clipping are shortly described afterwards.

1.3.1. The Sutherland-Cohen Line Clipping Algorithm.

The line clipping algorithm invented by Ivan Sutherland and Dan Cohen [NEWMAN79] clips lines against a two-dimensional rectangular window or a three-dimensional viewing frustum.

It was designed with two goals in mind:
- Reject off-screen lines as fast as possible.
- Find efficiently the clipped endpoints.

In the two-dimensional version, the algorithm divides the space occupied by the unclipped picture into nine regions, extending the boundaries of the window towards infinity. A four-bit code is assigned to each region. Each bit tells whether a point is outside (bit set) or inside (bit unset) a certain boundary.

Fig. 1.2.
Region classification codes for points in Sutherland-Cohen line clipper.
In pseudo-code, the algorithm can be expressed as follows:

```
1 Compute region code of each line endpoint;
2 While (at least one endpoint code is nonzero) {
  3     if (same bit is set in codes of both endpoints)
  4         terminate; /* Line fully outside */
  5         clip line against boundary designated by non-zero bit;
  6         replace line endpoint with intersection point;
  7         recompute region code of new endpoint;
}
8 output line;
```

All lines, whether fully outside or fully inside the window will need an average of 1.5 + 1.5 comparisons for computing the region code of each endpoint (step 1) (see Ch. 2.4.2). On-screen lines will be categorized by the first region code check (step 2) and can be output. Trivially rejectable lines, that is lines with both endpoints outside the same boundary, will be rejected in step 3. Other lines with a more complex relation to the window boundaries will require actual computation of intersection points and new region codes before being rejected or output in clipped form.

1.3.2. The Sutherland-Hodgman "Reentrant Polygon Clipper".

The Reentrant Polygon Clipper was designed as a simple, yet powerful solution to the problem of clipping polygons to convex regions.

The basic principle of the algorithm is that clipping is divided into multiple stages. Each stage clips the input polygon to a single extended boundary line, and outputs a closed polygon in a format identical to the input format. The clipping stage removes vertices outside the boundary, and introduces new vertices at boundary crossings. By letting each stage clip against just a single boundary, the problem of finding a closed polygon to output becomes simple, even when the polygon surrounds window corners.
By chaining four stages, clipping against a rectangular two-dimensional window can be done.

By chaining six stages, a six-plane truncated pyramid can be used as the clipping volume.

Moreover, the algorithm can easily be set up to handle clipping against any convex region in two- or three-dimensional space.

The name "Reentrant Polygon Clipper" is because the algorithm can be readily implemented as a recursive algorithm, with intermediate storage of two vertices (initial point and previous point for the polygon). Most implementations of the algorithm, however, use an implementation with a different subroutine for each stage.

![Diagram](image)

Fig. 1.3.

Polygon clipped by different stages in a two-dimensional clipping pipeline. Notice the degenerated polygon resulting from the bottom-stage clip (arrow).
It should be noticed that if concave polygons are input to the "Reentrant Polygon Clipper", degenerate polygons may be output (see Fig. 1.3).

1.3.3. The Liang-Barsky Algorithm for Polygon Clipping.

In contrast to the Sutherland-Hodgman algorithm which logically handles the relation between the polygon and a single window boundary at a time, the Liang-Barsky algorithm handles the relation between all clipping boundaries and a single polygon edge at a time [LIANG83].

The algorithm uses a parametric representation for the line;

\[
\begin{align*}
  x &= x_i + \Delta x_i \cdot t, \text{ where } \Delta x_i = (x_{i+1} - x_i) \\
  y &= y_i + \Delta y_i \cdot t, \text{ where } \Delta y_i = (y_{i+1} - y_i)
\end{align*}
\]

\((x_i, y_i)\) and \((x_{i+1}, y_{i+1})\) are the endpoints of edge number 'i'. The parameter 't' will be in the range \(0 \leq t \leq 1\) on the actual edge. The clipping situation is categorized by the values of the parameter at the intersection of the extended line \((-\infty \leq t \leq +\infty)\) with the extended window boundary lines. Six different cases need to be handled.

When concave polygons are input, the algorithm may output degenerate polygons.

An implementation of the algorithm in "C" is found in Appendix D.

1.3.4. Other Clipping Algorithms.

1.3.4.1 Liang-Barsky Line Clipping Algorithm.

The Liang-Barsky line clipping algorithm [LIANG84] uses a parametric representation for the line to be clipped. The intersection parameters at the intersection points are computed (see Ch. 2.3.1). For each boundary, as long as an edge is not completely outside, the intersection parameters are recomputed. When all parameters for the intersections have been computed the actual clipped endpoints are computed using the minimized/maximized parameters (see Cyrus-Beck algorithm below).
1.3.4.2 The Cyrus-Beck Line Clipping Algorithm

The Cyrus-Beck line clipping algorithm [ROGERS85a] clips a line against a convex clipping region. It is particularly effective when clipping a line against a large set of bounding planes - for instance when intersecting a line with a cylindrical surface approximated by planes.

The Cyrus-Beck algorithm represents a line segment parametrically. The parameter value for an intersection with a plane is computed in the same way as described in Ch. 2.3.1.

Because a line can intersect a convex region in at most two points, finding the actual intersection points is limited to finding the minimum of the maximum parameter value and the maximum of the minimum parameter values (a classical linear programming problem).

The algorithm is useful in both two- and three-dimensions.

[ROGERS85b] describes how it can be used for polygon clipping.

1.3.4.3 Weiler-Atherton Polygon Intersector

The context of the Weiler-Atherton polygon clipping algorithm [WEILER77] is a hidden surface algorithm working in object space. The algorithm computes the area of a polygon which is not covered by polygons of higher priority.

As a scene is traversed front-to-rear, a polygon is processed as follows:

1) It is clipped against the region occupied by previously rendered polygons,

2) The non-obscured part is rendered,

3) The non-obscured part is added to the clipping region.

The Weiler-Atherton algorithm is able to handle clipping of a concave subject polygon against a concave clipping region.
The principle of the algorithm is to find all intersections of the subject polygon and the clip polygon and create a vertex list sorted clockwise for both polygons with the intersections in place. The resulting "inside" polygon can be found by starting at the subject polygon list at an "entering" intersection and then alternating between the two lists for each intersection until the starting point is revisited. An "entering" intersection is where an edge of the subject polygon enters into the clipping region. Starting in the clip polygon list at a "leaving" intersection and alternating between the two lists at intersection, will give the "outside" polygon parts. As multiple polygon parts can result from the concave region clipping, the traversal must be repeated until all vertices in the two lists have been visited.

1.3.4.4 Burkert and Noll Algorithm

Burkert and Noll [BURKERT88] describe how the Liang-Barsky polygon clipper can be extended for clipping concave polygons against three-dimensional clipping windows. Their algorithm avoids output of degenerate edges. They comment that their algorithm can be generalized to clipping in homogeneous coordinates, but that the number of arithmetic operations increases considerably.

1.3.4.5 Kilgour's Boundary Chain Algorithm

Kilgour's Boundary Chain algorithm [KILGOUR87] is a polygon clipping algorithm which is able to produce separate output sheets on output instead of degenerate polygons as with the Sutherland-Hodgman algorithm. It uses the Sutherland-Cohen line clipper for computing the edge intersections. The vertices are collected into lists with the intersections inserted and traversed as with the Weiler-Atherton algorithm. Kilgour computes for each pair of intersections a "wrap number" which can be used to categorize the inside and outside parts according to different definitions for what is inside a polygon (mainly of interest when self-intersecting polygons or polygons with holes are defined).

1.3.5. Some Critical Remarks on the Algorithms.

As formulated in [LIANG83], the Liang-Barsky algorithm is limited to rectangular clipping windows.
Due to its computations on parameter values, rather than actual coordinate values at the intermediately computed intersection points, the Liang-Barsky algorithm will handle edges that intersect multiple boundaries efficiently. Moreover, the clipping algorithm can be implemented in a single procedure with a main loop over all polygon vertices, and simple if-then-else-constructs branching on the different cases of window intersection.

The Liang-Barsky polygon clipper bears a reputation of being the most efficient polygon clipper around. This is probably due to some concluding remarks in [LIANG83] that for an "arbitrary" example input polygon, the algorithm "required only half the execution time than that of the traditional Sutherland-Hodgman reentrant polygon clipping algorithm".

The "arbitrary" polygon used for timing comparison (see Fig. 3.1) is interesting in being a good "burn-in" test for the algorithm. It has many edges, generating "turning" points at several corners, and results in a degenerate polygon. However, as is evident from some timing experiments (see Ch. 3.1), the concave polygon seems to be very much in favour of the Liang-Barsky algorithm. Moreover, if there is such a thing as one "typical" polygon for clipping algorithm benchmarking, I find it unlikely that it will be similar to this "arbitrary" polygon.

The Sutherland-Hodgman polygon clipper's major advantage is its simplicity and flexibility. It is effective when used with only a single clipping boundary, but when many vertices are piped through several clipping stages, efficiency suffers.

The Weiler-Atherton algorithm is very flexible allowing concave subject polygons and concave regions, but efficiency suffers when simpler cases are to be handled. While the context is three-dimensional hidden surface removal, the algorithm basically does area calculations in $\mathbb{R}^2$.

The Cyrus-Beck algorithm is efficient for line clipping with convex clipping regions, but the polygon clipping extension seems to spoil this.

The Burkert-Noll algorithm probably possesses the same properties as the Liang-Barsky algorithm. The negative characteristics of this algorithm when
used with homogeneous clipping regions is a definite disadvantage. Burkert and Noll do not give any clues as to the performance of their algorithm, nor is any implementation or coding described.
2. DESIGN

2.1. The Scene Geometry - Preconditions

When designing a fast general purpose clipper for a general purpose graphics package, we have to concentrate on the most commonly occurring clipping situations. In most interesting cases we can state the following preconditions:

P1. There are many polygons inside the window.

P2. Average polygon size is much smaller than window size.

For many applications, the window/clipping is used for navigation through a huge environment. This is typically the case with CGI-systems (Computer Generated Imagery) for visual vehicle simulators and applications such as land survey or mapping. For these applications we have:

P3a. The number of polygons outside the window is much larger than the number of polygons inside the window.

Alternatively, when viewing three-dimensional designed objects, we keep most of the object on screen, but occasionally some parts of the object may go outside the boundary. For this, we at least can say:

P3b. The number of polygons fully inside the window is much larger than the number of polygons intersecting the boundary.

For three-dimensional imagery, when tiled polygons are used to approximate surface shapes, we know:
P4. The vast majority of polygons will be convex, and have only three or four edges.

If P2 holds, we can deduce:

P5. The probability that a polygon intersects a window boundary is small \((p << 1.0)\), and the probability that a polygon intersects two or more window boundaries is very small \((p^2 << 1.0)\).

For most man-created scenes there will be some structure on the polygons in the scene.

P6. Polygons will tend to be geometrically clustered into some kind of objects, surface patches of higher order etc.

### 2.2. Overall Design Decisions

Of the described algorithms, only the original line clipper of Sutherland and Cohen explicitly states that it was designed to handle polygons fully inside or fully outside the clipping area most efficiently, which seems to be according to our preconditions.

If the benchmarking of Liang and Barsky is to be taken seriously, one should believe that their polygon clipper was designed to handle large, complicated polygons efficiently.

In this clipper design we will concentrate on scenes which have the properties described in the last section. The design must therefore handle efficiently (in priority order):

1. Objects or polygon clusters completely outside the window.
2. Objects or polygon clusters completely inside the window.
3. Objects or polygon clusters intersecting a single boundary.
Objects or polygon clusters intersecting several boundaries.

The clipping will be handled by two mechanisms: a boxing algorithm for objects and a polygon clipper for polygons.

---

Fig. 2.1

Geometry processing stages consisting of fast extent testing algorithm, viewing transformation and Stacking Clipper. Matrix C contains clipping boundaries. Matrix T contains transformation. Concatenated matrix T C contains clipping boundaries transformed to object space (see Ch. 2.4).
The boxing algorithm will test the extent of the object against the window boundaries and decide how to handle the polygons inside the object. The extent of the object is some simple geometric volume which is guaranteed to contain all elements (polygons) in the object. Here we have chosen to use a box with axes aligned with the object coordinate system. The window or clipping region will be described by a set of two- or three-dimensional halfspaces.

The extent test decides on the further treatment of the polygons in the object:

i) If the box is completely outside the window, all contained polygons are discarded.

ii) If the box is completely inside the window, all contained polygons are drawn without any clipping.

iii) If the box intersects one or more boundaries, a bitmask is generated indicating the intersections. All contained polygons are passed to the clipper along with the bitmask.

The clipping algorithm will take a polygon and perform clipping against only the boundaries indicated in the bitmask. Most frequently only a single boundary will be used for polygon clipping. Though the boxing algorithm has indicated intersection with window boundaries, few of the contained polygons will actually have to be clipped against the indicated boundaries.

The one-boundary-plane-at-a-time philosophy of Sutherland-Hodgman will be adopted. We can thus model our clipping algorithm by a pipeline of clipping stages, each implementing one single clipping boundary. Clipping a polygon to a single boundary is a rather uncomplicated task, and this task can simply be extended to cutting an input polygon into two output polygons along the boundary. The clipper with this added (inverted) output can be named a divider.
Fig. 2.2
A clipping stage shown as a "box" with polygon input channel, "inside" polygon output channel, and optional "outside" polygon output channel.

We can thus model each clipping stage as a "box" with one polygon input channel and one "inside boundary" polygon output channel and an optional "outside boundary" polygon output channel. For clipping to convex regions only the "inside" channels are used. More discussion on building divider networks is found in Ch. 4.

2.3. Mathematics for Clipping

2.3.1. Two-Dimensional Halfspace Clipping

A two-dimensional halfspace consists of the area of the plane on one side of a line in the plane. Using the implicit equation of the line:

\[ ax + by + c = 0, \]

a point \((x_1, y_1)\) in the plane will have a distance to the line

\[ d = \frac{a * x_1 + b * y_1 + c}{\sqrt{a^2 + b^2}} \]
When the equation coefficients are normalized, that is, $a^2 + b^2 = 1$, then the distance equation is simplified to:

$$ d = a \times x_1 + b \times y_1 + c $$

For clipping, we need not use the normalized boundary equation coefficients, as we are only making use of the sign of the distance measure and the ratios of distances from two points to the boundary. However, we adopt the convention that points inside the halfspace have a positive distance measure to the line.

We will use a column vector to represent the boundary line by its coefficients:

$$ \chi = [a, b, c]^T. $$

By representing a point $(x_1, y_1)$ by its homogeneous row vector:

$$ \rho = [x_1, y_1, 1] $$

the distance measure is simply the inner (dot) product:

$$ \lambda = \rho \cdot \chi $$

and a point $\rho$ is inside the boundary $\chi$ when $\rho \cdot \chi > 0$. 

Fig. 2.3
Finding the intersection point of an edge and a boundary.

The intersection point $\rho$ of an edge with endpoints $\rho_1$ and $\rho_2$ and a boundary $\chi$, can be computed by:

$$\rho = \rho_1 + \alpha \times (\rho_2 - \rho_1),$$

where:

$$\alpha = \frac{d_1}{d_1 + d_2}$$

$$d_1 = \rho_1 \cdot \chi$$

$$d_2 = \rho_2 \cdot \chi$$

Note: Because we use the ratio of the distances ($\alpha$) rather than the absolute distances, the boundary coefficients in $\chi$ need not be normalized.

We also let the predicate $\text{in}(\text{point}, \text{halfspace})$ denote whether a point is inside the halfspace. The normal rectangular clipping region can thus be described by the Boolean expression:

$$\text{in}(P, \alpha) \text{ and } \text{in}(P, \beta) \text{ and } \text{in}(P, \chi) \text{ and } \text{in}(P, \delta)$$
where:

\[ \alpha = [1, 0, -w_{x1}]^T \]
\[ \beta = [-1, 0, w_{x2}]^T \]
\[ \gamma = [0, 1, -w_{y1}]^T \]
\[ \delta = [0, -1, w_{y2}]^T \]

and \( w_{x1}, w_{x2}, w_{y1}, w_{y2} \) represent the low and high x and low and high y of the window boundary. A clipper implementation consists of the four clipping stages pipelined on the "inside" channel.

### 2.3.2. Three-Dimensional Halfspace Clipping

In three-dimensional space, a plane divides the space into two halfspaces.

If we have a boundary plane represented by the implicit equation

\[ ax + by + cz + d = 0, \]

we can use a column vector to represent the plane by its coefficients:

\[ \chi = [a, b, c, d]^T \]

By representing a point \((x_1, y_1, z_1)\) by its homogeneous row vector:

\[ \rho = [x_1, y_1, z_1, 1], \]

the distance measure is the inner product:

\[ \lambda = \rho \cdot \chi \]

and a point \( \rho \) is inside the halfspace bounded by the plane \( \chi \) when

\[ \rho \cdot \chi > 0. \]
2.3.3. Clipping in Homogeneous Space

Three-dimensional clipping is closely related to the viewing transformations. As made specific in the Core system specifications [GSPC79, FOLEY84, or ZACHRISEN89a], the viewing transformation is composed of several transformation matrices which map the general viewing frustrum into a canonical view volume which is a cut-off unit pyramid with tip (Centre Of Projection) at the origin and its axis along the negative z-axis†.

For perspective projections, the viewing transformation can be cleverly implemented as a 4x4 transformation matrix followed by division on the homogeneous coordinate.

Fig. 2.4

a) The canonical view volume for perspective viewing and
b) the view volume after being mapped by the perspective transformation.
The canonical view volume is shown on Fig. 2.4a. The perspective transformation using the matrix:

\[
\mathbf{M} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & \frac{1}{1-a} & -1 \\
0 & 0 & \frac{a}{1-a} & 0
\end{bmatrix}
\]

maps the canonical viewing volume into a parallelepiped. The homogeneous division accomplishes the perspective (z-coordinate) foreshortening, and preserves relative depth and straight lines and planes. The latter is central when preparing data for a hidden surface algorithm, as in the GAS system.

Fig. 2.4a shows the view frustrum in the "normalized viewing reference coordinate system" (or "normalized eye coordinate system"), where the slopes of the side-planes of the viewing pyramid have been scaled to unity. Fig. 2.4b shows the viewing frustrum after being mapped into the projection coordinate system. This maps the viewing (truncated) pyramid into a box with \(-1 \leq x \leq 1, -1 \leq y \leq 1, -1 \leq z \leq 0\). The centre of projection (or eye position) is transformed into a point at infinity (with homogeneous representation \([0, 0, 1, 0]\)). The "far" clipping plane is mapped into the plane \(z = -1\), and the near clipping plane is mapped to the plane \(z = 0\).

There are three commonly used methods for three-dimensional clipping:

a) We can clip to the simple parallelepiped after perspective transformation. This simplifies the clipping process. However, at this point, the rear viewing pyramid has collapsed with the front viewing pyramid into the clipping region. This is caused by the perspective division, and makes this approach unacceptable for our use.

b) We can clip to the canonical clipping volume, prior to the viewing transformation. The clipping planes are:

\(-x - z = 0, x - z = 0, -y - z = 0, y - z = 0, -z = a, z = 1,\)

where \(a\) is the hither clipping distance (see Fig. 2.4a). This works well, and avoids unwanted graphic elements from the "rear" viewing pyramid. However, it disallows any use of the
homogeneous coordinate for special modelling or vanishing point transformations. Moreover, the perspective transformation must be carried out after clipping.

c) The third alternative is to perform three-dimensional clipping after the perspective transformation, but before the division part of it. Thus we do the clipping in homogeneous coordinates. This may seem somewhat more complicated, but has the desired effect of avoiding the "rear" pyramid, and allowing the use of the homogeneous coordinate during modelling. For an effective implementation, it allows the perspective transformation matrix to be concatenated with the viewing and modelling transformation matrices.

Clipping in homogeneous coordinates is a natural choice for our design, and will be the topic of the rest of this chapter. Some special problems with this approach will also be discussed in Ch. 5.

As a homogeneous coordinate \((xw, yw, zw, w)\), for any value of \(w\) represents the three-dimensional point \((x, y, z)\), the clipping region

\[-1 \leq x \leq 1, \ -1 \leq y \leq 1 \text{ and } -1 \leq z \leq 0\]

on Fig. 2.4b would be represented by a volume limited by the six hyperplanes in homogeneous space:

\[
\begin{align*}
x + w &= 0, \\
-x + w &= 0, \\
y + w &= 0, \\
-y + w &= 0, \\
z + w &= 0, \\
z &= 0.
\end{align*}
\]

As a line bounds a halfspace in \(E^2\), a plane bounds a halfspace in \(E^3\), and a hyperplane (3-flat) bounds a halfspace in \(E^4\) [FRALEIGH87].

Disregarding the X-coordinate, the bounding planes in homogeneous space are shown on Fig. 2.5a and Fig. 2.5b.
Fig. 2.5

a) The "top" and "bottom" boundary planes and
b) the "hither" and "yon" boundary planes used for clipping in homogeneous space
(projected onto yzw-space).

The left, right, bottom and top clipping planes restrict graphics from the "rear" viewing volume from appearing. The hither and yon clipping restrict the range of the depth coordinates output.

The clipping region can be described by the Boolean expression:

\[
\text{in}(P, A) \text{ and in}(P, B) \text{ and in}(P, C) \text{ and}
\text{in}(P, D) \text{ and in}(P, E) \text{ and in}(P, F),
\]

where the plane coefficients are (for planes in homogeneous coordinates, five coefficients are used):

\[
\begin{align*}
A &= [1, 0, 0, 1, 0]^T \\
B &= [-1, 0, 0, 1, 0]^T \\
C &= [0, 1, 0, 1, 0]^T \\
D &= [0, -1, 0, 1, 0]^T \\
E &= [0, 0, 1, 1, 0]^T \\
F &= [0, 0, -1, 0, 0]^T
\end{align*}
\]

(left),
(right),
(bottom),
(top),
(far)
(near),
It can be noted that for our purpose - projection onto three-space - all hyperplanes in homogeneous space will contain the origin. That is, the fifth coefficient will be \( e = 0 \), and we will for convenience leave it out, and represent a halfspace in homogeneous space with a four-element vector.

The distance from a point \((x_1, y_1, z_1, w_1)\) in homogeneous space to the boundary of a halfspace

\[
a \ast x + b \ast y + c \ast z + d \ast w + e = 0,
\]
equals:

\[
\lambda = a \ast x_1 + b \ast y_1 + c \ast z_1 + d \ast w_1
\]

when \( e = 0 \) and the coefficients are normalized.

If we use a column vector to represent the homogeneous halfspace bounding plane by its (non-trivial) coefficients:

\[
\chi = [a, b, c, d]^T.
\]

and represent a point \((x_1, y_1, z_1, w_1)\) by a row vector:

\[
\rho = [x_1, y_1, z_1, w_1],
\]

the distance measure once again becomes the four-element inner product:

\[
\lambda = \rho \cdot \chi
\]

and a point \( \rho \) is inside the halfspace bounded by the plane \( \chi \) when

\[
\rho \cdot \chi > 0.
\]

Using this distance measure, intersection computations can be done as in two-dimensional space, using non-normalized halfspace coefficients if more convenient.
2.4. Extent Testing Algorithms

Extent testing is used to prune the scene of invisible parts without going into the geometrical detail of the part. Extent testing implies overhead on the rendering of the scene and is only useful when a reasonable part of the scene can be removed from rendering or clipping.

Though extents are here only considered in the context of making clipping more efficient, extents are of vital importance to rendering techniques such as ray tracing, and also very useful for interactively "picking" objects on the screen.

Extents have a simple geometric shape, keyed to their use. Thus, for extent testing in ray tracing, spheres are commonly used. For our use, we choose an upright extent box. When an upright box is used for extent testing, the extent testing process is commonly referred to as boxing.

For interactive viewing of objects, the objects will have to be transformed by a modelling transformation or a viewing transformation or both. The viewing transformation is very likely to change between frames (if we "walk around" the object). The modelling transformation may change between frames (if we simulate the movement of parts). For most problem areas, however, the geometry inside an object will be unchanged over frames.

When object geometry is unchanged over frames, we will define the upright extent box in the object coordinate system. The extent box will have to be transformed by the modelling transformation and the viewing transformation prior to testing against the clipping halfspaces.

2.4.1. Where Does the Extent Come From?

Sometimes, it is not obvious how the scene is partitioned into disjoint extent boxes. The extent may be defined manually, implied by the overall structure of the scene or heuristically, based on some optimal partitioning criterion.
For instance, in our harbour data test example, extents (or objects) were defined manually during digitization of the landscape, with approximately 32 points per object (see colour plates 2a-c). In our stylized test where the scene consists of nicely arranged boxes, extents are set up to contain a specified number of polygons.

Inside a general purpose graphics package like GPGS-F or PHIGS, one may generate a hierarchy of extents, directly corresponding to the structured display file (nested picture segments or objects).

In an environment where no structure clues are implicit, heuristic algorithms can be used to build the extent structure. Goldsmith and Salmon describe an algorithm for building an extent hierarchy for ray tracing [GOLDSMITH87]. For ray tracing, trees with small branching ratios are found to be optimal.

In general, we must weigh the cost of constructing the extent structure with the actual computation gains during rendering. With a dynamically changing scene, this can be more difficult to judge.

### 2.4.2. Two- and Three-Dimensional Extent Testing Sequence

Using our design policy of Ch. 2.2, we will optimize the rejection of boxes.

For instance, for an untransformed two-dimensional extent, the testing sequence (1):

\[
\text{if} \ (x_{\text{min}} > \text{wind}_{\text{right}} \ || \ x_{\text{max}} < \text{wind}_{\text{left}} \ || \ y_{\text{min}} > \text{wind}_{\text{top}} \ || \ y_{\text{max}} < \text{wind}_{\text{bottom}}) \\
\text{reject;}
\]

is more efficient than the testing sequence† (2):

\[
\text{if} \ (x_{\text{min}} > \text{wind}_{\text{right}} \ || \ y_{\text{min}} > \text{wind}_{\text{top}} \ || \ x_{\text{max}} < \text{wind}_{\text{left}} \ || \ y_{\text{max}} < \text{wind}_{\text{bottom}}) \\
\text{reject;}
\]

† Assuming the || operator does not evaluate its right operand when the left operand is TRUE, as in the "C" programming language.
When only a small part of the scene is inside the window, the probability that each test can reject the object is:

\[
P(x_{\text{min}} > \text{wind}_{\text{right}}) = 0.5 - \delta \\
P(x_{\text{min}} < \text{wind}_{\text{right}} \land x_{\text{max}} < \text{wind}_{\text{left}}) = 0.5 - \delta \\
P(x_{\text{min}} < \text{wind}_{\text{right}} \land y_{\text{min}} > \text{wind}_{\text{top}}) = (0.5 + \delta) \times (0.5 - \delta) \\
P(x_{\text{min}} < \text{wind}_{\text{right}} \land x_{\text{max}} > \text{wind}_{\text{left}}) = 2\delta \\
P(x_{\text{min}} < \text{wind}_{\text{right}} \land x_{\text{max}} > \text{wind}_{\text{left}} \land y_{\text{min}} > \text{wind}_{\text{top}}) = 2\delta \times (0.5 - \delta)
\]

where \( \delta \) is small\(^{\dagger \dagger} \). With the sequence (1), the expected no. of boundary tests to perform is:

\[
E_1 = 1 \times (0.5 - \delta) + \\
2 \times (0.5 - \delta) + \\
3 \times (2\delta) \times (0.5 - \delta) + \\
4 \times (2\delta) \times (0.5 + \delta)
\]

\[E_1 = 1.5\]

while for (2):

\[
E_2 = 1 \times (0.5 - \delta) + \\
2 \times (0.5 + \delta) \times (0.5 - \delta) + \\
3 \times (0.5 - \delta) \times (0.5 + \delta) + \\
4 \times (2\delta) \times (0.5 + \delta)
\]

\[E_2 = 1.75\]

We see that the first testing sequence performs slightly better than the last one and that we need an average of 1.5 boundary tests for discarding boxes when the actual scene is much smaller than the total environment.

In the following we will concentrate on three-dimensional boxing.
2.4.3. "Brute Force" Boxing

Carried out the straightforward way, each of the eight corner points of a three-dimensional extent box will have to be transformed by the concatenated model-view transformation. Then each point will have to be tested against each clipping boundary. This would amount to 28 floating point operations for transforming the point, and then 1 floating point operation for testing against each boundary. With the normal 6 boundaries, this would sum up to 

\[(28 + 6) \times 8 = 272\]

floating point operations for the eight corners of the extent.

If the transformation is carried out more cleverly, transforming relative positions of the extent corners rather than the absolute positions, the number of floating point operations could be reduced to 71 for the transformation, which would give a total of 

\[71 + 6 \times 8 = 119\]

floating point operations.

2.4.4. Michener's Algorithm

Michener has devised an algorithm to transform an upright extent to a new upright extent [FOLEY84]. This algorithm requires only 24 additions and 18 multiplications for the transformation with a 3x3 matrix, using a method similar in spirit to the simplex method of linear programming. The algorithm can not be used with perspective projections. For scenes where parallel projection is sufficient, the total work involved for doing boxing tests would be 48 floating point operations for a 4x3 transformation matrix multiplication and 6 floating point operations for comparing the extent with the clipping boundaries, a total of 54 floating point operations. (Because the extent is transformed to a new upright extent, clipping tests are greatly simplified.)
2.4.5. A Fast Boxing Algorithm

The computations for the six three-dimensional clipping boundaries can be expressed as the product of three matrices:

\[ \mathbf{D} = \mathbf{P} \mathbf{T} \mathbf{C} \]

\[ \mathbf{P} = \begin{bmatrix} x_{\text{min}} & y_{\text{min}} & z_{\text{min}} & 1 \\ x_{\text{max}} & y_{\text{min}} & z_{\text{min}} & 1 \\ x_{\text{min}} & y_{\text{max}} & z_{\text{min}} & 1 \\ x_{\text{max}} & y_{\text{max}} & z_{\text{min}} & 1 \\ x_{\text{min}} & y_{\text{min}} & z_{\text{max}} & 1 \\ x_{\text{max}} & y_{\text{min}} & z_{\text{max}} & 1 \\ x_{\text{min}} & y_{\text{max}} & z_{\text{max}} & 1 \\ x_{\text{max}} & y_{\text{max}} & z_{\text{max}} & 1 \end{bmatrix} \]

\[ \mathbf{T} = \begin{bmatrix} t_{11} & t_{12} & t_{13} & t_{14} \\ t_{21} & t_{22} & t_{23} & t_{24} \\ t_{31} & t_{32} & t_{33} & t_{34} \\ t_{41} & t_{42} & t_{43} & t_{44} \end{bmatrix} \]

\[ \mathbf{C} = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} \]

The rows of the \( \mathbf{P}_{[8 \times 4]} \) matrix are the corner points of the extent box in homogeneous coordinates. The matrix \( \mathbf{T}_{[4 \times 4]} \) is the concatenated model/view transformation. The columns of the matrix \( \mathbf{C}_{[4 \times 6]} \) are the six boundary planes. The resulting matrix \( \mathbf{D}_{[8 \times 6]} \) contains, in its six columns, the distance measure from each corner point to a clipping boundary (left, right, bottom, top, yon, hither, respectively). We choose the boundary plane coefficients such that a positive distance \( d_{i,j} \) means that point no. \( i \) is inside the clipping window boundary. Thus, if each element \( d_{1,j} .. d_{8,j} \) in a column of \( \mathbf{D} \) has the same sign, the whole extent with all contained polygons does not intersect the boundary number \( j \).
By concatenating $T$ and $C$ and making use of relative positions of corner points of the extent, we can greatly simplify boxing computations.

$$
T \ C = \begin{bmatrix}
    t_{11} + t_{14} & -t_{11} + t_{14} & t_{12} + t_{14} & -t_{12} + t_{14} & t_{13} + t_{14} & -t_{13} \\
    t_{21} + t_{24} & -t_{21} + t_{24} & t_{22} + t_{24} & -t_{22} + t_{24} & t_{23} + t_{24} & -t_{23} \\
    t_{31} + t_{34} & -t_{31} + t_{34} & t_{32} + t_{34} & -t_{32} + t_{34} & t_{33} + t_{34} & -t_{33} \\
    t_{41} + t_{44} & -t_{41} + t_{44} & t_{42} + t_{44} & -t_{42} + t_{44} & t_{43} + t_{44} & -t_{43}
\end{bmatrix}
$$

By naming each row vector of $D \ d_i$, we can set up the following recursive formulas for $d_i$:

$$
\begin{align*}
    d_1 &= \begin{bmatrix} x_{\text{min}} & y_{\text{min}} & z_{\text{min}} & 1 \end{bmatrix} \ T \ C \\
    d_2 &= d_1 + \begin{bmatrix} x_{\text{max}} - x_{\text{min}} & 0 & 0 & 0 \end{bmatrix} \ T \ C \\
    d_i &= d_{i-2} + \begin{bmatrix} 0 & y_{\text{max}} - y_{\text{min}} & 0 & 0 \end{bmatrix} \ T \ C, \text{ for } i = 3, 4 \\
    d_j &= d_{j-4} + \begin{bmatrix} 0 & 0 & z_{\text{max}} - z_{\text{min}} & 0 \end{bmatrix} \ T \ C, \text{ for } j = 5, 6, 7, 8
\end{align*}
$$

For each boundary (each column in the distance matrix $D$), there are only 6 multiplications and 20 additions that must be carried out for the extent test. When box sizes are small compared to the window, and the window size is small compared to the whole scene, the average number of floating point operations to reject a box will be only $1.5 \times (20 + 6) = 39$ floating point operations!

Conceptually, this way of doing the boxing computations is equivalent to transforming the clipping boundaries into object space. The concatenated matrix $T \ C$ contains the transformed clipping boundaries.

---

† The factor 1.5 is explained in Ch. 2.4.2.
2.5. A Fast Clipping Algorithm

The basic design criteria for the algorithm is to do clipping boundary-by-boundary (The Sutherland-Hodgman way) rather than edge-by-edge (The Liang-Barsky way). This makes it possible more easily to make use of the information from the extent test and activate just the necessary clipping stages. A second advantage of this is that the algorithm may easily be expandable to clip against other shapes than a rectangular window.

Moreover, in each clipping stage the handling of edges fully inside or fully outside the clipping boundary should be as fast as possible.

The algorithm should not pose any restrictions on the size of the polygon, that is, the algorithm should not need an internal buffer with a limiting size. From experience, polygons will always exceed the expected maximum number of vertices. A typical example is the polygon shape resulting from the area under the curve in "area charts" (see Fig. 2.6).

Fig. 2.6
Complex, concave polygon which may cause problems in clipping algorithms where the maximum number of vertices is restricted to some compiled-in constant.
A more implementation related issue is that floating point calculations must be minimized. This has to do with the fact that the algorithm is intended to be used on cheap PC/workstations or raster system firmware. In these hardware environments, integer operations are often 1-2 orders of magnitude faster than floating point operations.

The basic structure of the algorithm is similar to a "push-down automaton" as used in grammar parsing [HOPCROFT79]. Points are input on top of the stack and will simply "float through" all clipping stages as long as the edge from the previous point does not cross the boundary managed by the stage. When a stage has processed an edge, and the edge is not clipped, the endpoint is forwarded to the next stage inside a record together with a bitmask indicating which remaining stages the point should pass through.

Due to this special structure, the algorithm is referred to as the "Stacking Clipper".

2.5.1. Finding the Output Vertices

Adopting the Sutherland-Hodgman principle of conceptually doing the clipping against one single boundary at a time simplifies the core of the algorithm. Each new vertex of the polygon is categorized as inside or outside the clipping boundary; this can be done by computing a three-element dot-product between the homogeneous representation of the vertex and the line or plane vector. A positive dot-product indicates that the point is inside, a negative dot-product indicates that the point is outside.
Fig. 2.7.
Polygon with edges demonstrating the four cases to be handled for clipping - AB: both inside, BC: going out, CD: both outside, DA: going in.

After both endpoints of an edge have been categorized, four cases must be handled (referring to the edges on Fig. 2.7):

AB: Both are inside - the new point (B) is passed on.

BC: The new point is outside, while the last point was inside - the intersection is passed on.

CD: Both points are outside - nothing should be passed on.

DA: The old point was outside, while the new point is inside - the intersection and the new point (A) are passed on.

When going in or out, the coordinates of the intersection are computed, based on the dot-products used for edge endpoint classification (see Ch. 2.3.1).

2.5.2. Algorithm Pseudocode

The pseudocode shown here is patterned after "C" language constructs and is probably slightly more readable than the implemented code in the Appendix.

Note that, by default, each stage prepares the stack record for the next stage. Thus no special stack handling is needed for unclipped edges.
The data structure consists of the stack of points, and records keeping the status information for each clipping stage. The point stack is the core of the "push-down automaton" style algorithm. Each stack entry consists of one (pointer to a) point and a bitmask determining the stage to process the point.

The most central local variables are the stack pointer (sp), and a pointer which is set to the current stage (stage).

```c
enum pstat {NONE, IN, OUT};  /* Point classify (initial: NONE) */
struct cstack {
    struct vertex *point;    /* Actual point (with attributes) */
    Bits clipmask;           /* Bit set means clipping needed
                              one bit for each stage,
                              lsb = next stage */
} Clipstack[NSTACK],
struct cstack *sp;          /* Clipping stack */
/* Clipping stack pointer */

struct cstage {
    enum pstat status;      /* Previous status */
    float prevdist;         /* Previous distance to boundary */
    struct vertex firstp;   /* First point - for closing */
    struct vertex prevp;    /* Previous point - for computing intersections */
    struct vertex isectp;   /* Storage for comp. intersection */
    double (*clipproc)();   /* Handler for each clipping boundary. Usage: distance = (*clipproc)(pt) */
} Clipstage[NSTAGES];
```

The algorithm loops until there are no more input points and the stack is empty.
repeat FOREVER {
    if (STACK_EMPTY()) {
        if (MORE_INPUT())
            /* Enter input polygon point onto stack */
            PUSH(boundarybits, INPUT_POINT());
        else
            PUSH(boundarybits, CLOSE);
    }

    /* Examine stack record */
    if (sp->clipmask != 0) { /* Handle by next active stage */
        struct cstage *stage = FIND_NEXT_STAGE(sp->clipmask);
        enum pstat startstatus = stage->status;
        enum pstat endstatus; /* Status of line endpoint */
        float dist; /* Distance endpoint to boundary */

        /* Update topofstack record (for next stage) */
        CLEAR_THIS_STAGE'S_BIT(sp->clipmask);

        if (sp->point == CLOSE) {
            /* Pass CLOSE on if output generated. */
            if (startstatus == NONE)
                break;
            PUSH(sp->clipmask, stage->firstp);
            stage->status = NONE;

            endp = sp->point;

            /* Compute distance for point to boundary */
            dist = (*stage->clipproc)(endp);
            endstatus = (dist < 0.0) ? OUT : IN;

            if (startstatus == NONE) { /* First point - store */
                stage->firstp = endp;
                stage->status = endstatus;
                POP();
            }
            else if (endstatus != startstatus) { /* Intersecting */
                stage->isectp = INTERSECT(startp, endp,
                stage->prevdist / (stage->prevdist - dist));

                if (endstatus == IN) /* Transmit intersection */
                    PUSH(sp->bitmask, isectp);
            }
            else if (endstatus == OUT) /* Both out - no output */
                POP();

            if (stage->status != NONE) { /* Update stage cont. */
                stage->prevp = endp;
                stage->prevdist = dist;
                stage->status = endstatus;
            }
        }
        else if (sp->point != CLOSE) { /* Reached collector */
            OUTPUT(sp->point);
            POP();
        }
        else /* Closing point passed thru all stages */
            break;
    } /* End of while not stack empty */
2.5.3. Algorithm Walkthrough

The functionality of the algorithm can best be illustrated by showing snapshots of how a simple polygon is processed by the algorithm. We choose a triangle which has a "turning point" outside the lower, left corner of a two-dimensional window. The triangle is processed by the "left" and "bottom" stages of the algorithm.

Fig. 2.8

Triangle ABC is clipped to AKL by stage 1 and then to AKMN by stage 2.

<table>
<thead>
<tr>
<th>Stage 1:</th>
<th>Stage 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack:</td>
<td>Input</td>
</tr>
<tr>
<td>-</td>
<td>A</td>
</tr>
<tr>
<td>-</td>
<td>B A A</td>
</tr>
<tr>
<td>S2:K</td>
<td>A B</td>
</tr>
<tr>
<td>-</td>
<td>C A B K K</td>
</tr>
<tr>
<td>-</td>
<td>Close A C K K</td>
</tr>
<tr>
<td>S2:Close</td>
<td>S2:A</td>
</tr>
<tr>
<td>S2:Close</td>
<td>S2:A S3:M</td>
</tr>
<tr>
<td>S2:Close</td>
<td>S2:A</td>
</tr>
<tr>
<td>S2:Close</td>
<td>S3:A S3:N</td>
</tr>
<tr>
<td>S2:Close</td>
<td>S3:A</td>
</tr>
<tr>
<td>S2:Close</td>
<td></td>
</tr>
<tr>
<td>S3:Close</td>
<td>S3:K</td>
</tr>
<tr>
<td>S3:Close</td>
<td></td>
</tr>
</tbody>
</table>

The table above shows the stack (three columns growing from left to right) and the state of each stage at the start of the loop in the algorithm. When the stack is empty, new points are input. This is indicated in the fourth column. S1 denotes the first (left) clipping stage, S2 denotes the second (bottom)
clipping stage, and S3 denotes the vertex collector. Thirteen iterations of the loop are necessary before the polygon MNAK is output as the result of the algorithm.

It can be noted that because each stage delays outputting the first point till the polygon is closed, the polygon vertices are rotated by one vertex for each active clipping stage.

2.5.4. Implementation Details.

The actual "C" code is shown in Appendix C. In the actual implementation the following design considerations were made in order to improve the usefulness of the clipping module.

The algorithm could well be invoked by pushing all points of the input polygon onto the stack. Though being an elegant solution, in the general case this would require a huge stack, or would limit the polygon size to something reasonable (100 vertices?). It was instead chosen to let the algorithm fetch a new vertex from an outside data structure whenever the stack becomes empty.

When an edge has been clipped in a stage, the output vertices will directly be processed by subsequent stages and thus vertices will tend to move rather directly from the input array to the output array rather than being stored in a deep stack. Each clipping stage can have at most two points active on the stack at any time (worst case: when an edge enters the visible region). Thus, for a six-stage three-dimensional clipper a stack with 12 records suffices.

The actual implementation does not move vertex records around: vertices are stored in the input array, the output array or intermediately as one of the three points stored at each stage. Each stage stores the first vertex entered to the stage (for closing), the previous vertex entered to the stage (for the edge being clipped), and has an additional store for one intersection point. When records are built on the stack, they contain a pointer to the appropriate vertex rather than the vertex itself.

The algorithm, as presented above, has input, output, distance computation and intersection computation located at a single place. These operations are likely candidates for being implemented as in-line macros rather than external
functions. In the "C" code implementation shown in the Appendix, input and output are handled by in-line macros, while intersection computation is handled by a macro only in the two-dimensional version.
3. BENCHMARKS AND COMPARISONS

Many papers will claim great speed advantages for their algorithms. The trouble is, the authors have never actually compared their algorithms with anything else, or only compared them in a trivial way. Someone will lovingly craft an implementation of an algorithm over a period of months or years. Then the would-be author feels it must be compared with something, so a weekend is spent tossing together a crude implementation of some older algorithm. Lo and behold, the proposed algorithm is faster!

James F. Blinn

Three different test suites have been run to evaluate the Stacking Clipper algorithm. The first two run in two-dimensional space, clipping a complex polygon and using a stylized database consisting of rectangles, respectively. The two-dimensional algorithm has its performance compared with an implementation of the two-dimensional Liang-Barsky polygon clipper. The third test runs in three-dimensional space using "real" data from a model of Singapore harbour, comparing performance with an implementation of the Sutherland-Hodgman Reentrant Polygon Clipper.

The Stacking Clipper has been implemented in "C" with reasonable regard to optimization using pointer arithmetic. The "generic" implementation of the "clipping engine" of the algorithm is shown in Appendix C. It is used for both the two-dimensional clipper and the three-dimensional clipper. The definitions for the two-dimensional and three-dimensional version of the clipper are shown in the prelude.

The tests were carried out on a Sun-3/50 with Motorola 68881 floating point processor (some of the tests also on a Sun-3 with floating point arithmetic in software). The timing was done using the Sun-3 operating system interface function "CLOCK(3C)" [SUN OS]. "Clock()" works on a discrete timing event, with a time step of 16.67 milliseconds. The CPU time (in Unix terms the sum of the user and system times of the calling process) is measured. All tests were run repeatedly, such that the actual intervals timed were in the 1-20 seconds range.
3.1. Two-Dimensional Benchmarks

Two sets of tests were run in two-dimensional space.

The two-dimensional Liang-Barsky polygon clipper was coded in the "C" language. The implementation was based on their Pascal program in [LIANG83]. The "C" implementation was given a fair optimization using register variables and pointer arithmetic. The actual program is listed in Appendix D.

3.1.1. A Large, Concave Polygon.

The first set of tests uses a scene consisting of a single concave polygon with nine vertices - nearly identical to the one used by Liang and Barsky for benchmarking their algorithm with Sutherland-Hodgman's [LIANG83]. The polygon is translated and scaled with respect to the window. However, the polygon size is in the same range as the window dimensions. As stated earlier, this is hardly a "typical" scene for a general purpose clipper, or at least, the Stacking Clipper was designed with quite another environment in mind.

The polygon was scaled and translated, producing six different test cases shown as a) to f) in the figure below.
Fig. 3.1
Six different tests for a two-dimensional clipper.

The tests are as follows:

a) Polygon is larger than window, outputting a degenerate polygon. The test is the same as used in [LIANG83]. This is a good test to see if the insertion of "turning" vertices works.

b) Polygon is smaller than window. This test shows the actual overhead in the clipper for "passing on" vertices.

c) A small part of the polygon (three vertices) goes outside the right window boundary.

d) Most of the polygon is outside the window with just two vertices entering past the right boundary.

e) The whole polygon is outside the window. This will effectively test the overhead for rejecting the polygon.

f) The window is completely covered by the polygon.
To verify that the output from the Liang-Barsky algorithm and from the Stacking Clipper are correct (and identical), the resulting output is shown in Fig. 3.2. This is taken as a screen dump of a Sun-3 running the X Window System (that's why there are jaggies). The dump shows the output from the tests a), c), and d). Due to the left-handed coordinate system of X, the polygon is vertically mirrored compared with Fig. 3.1.

Fig. 3.2
Screen dumps of the X window system onto laser printer showing tests a), c), and d). The centre window on each figure shows the input polygon unclipped. The top window on each figure is the output from Liang-Barsky algorithm. The lower window is the output from the Stacking Clipper.
The timing results are shown below.

Fig. 3.3
Times in seconds for doing tests a) to f) 1000 times on a Sun-3/50 workstation with floating point unit Motorola 68881. Note that curve 2) coincides with curve 3) for tests a) and f).

It can be seen that the only cases when the Liang and Barsky algorithm is at advantage, are the tests where the tested polygon is larger than the window: a) and f). In these cases, the Stacking Clipper will gradually reduce the nine-vertex polygon towards the window square, but will have to cope with a complex shape through most clipping stages.

When the polygon is fully inside the window, the algorithms perform equally well: b).

When the Stacking Clipper knows from the boxing test that only a single window boundary is intersected by the polygon, or when the polygon is fully outside the window, the Stacking Clipper outperforms the Liang and Barsky algorithm: c), d), and e).
The Stacking Clipper without any boxing information performs about as well as the Liang and Barsky algorithm for all but the worst cases: a) and f).

Fig. 3.4
Times in seconds for doing tests a) to f) 1000 times on a Sun-3/50 workstation without floating point unit. Note that curve 2) coincides with curve 3) for tests a) and f).

The tests were also run on a Sun-3/50 workstation which did not have the floating point unit. In the tests, the Stacking Clipper is even more at advantage (see Fig. 3.4).

3.1.2. Boxing Tests

Another test was set up to test the effect of boxing. The test consisted of constructing a scene of 64x64 rectangles, which were rotated by an angle $\phi$ to avoid alignment with the window boundaries. The rectangles were divided into boxes. Three different cases are shown in the chart below:

- 64x64 means one single box containing all rectangles (4096).
- 32x32 means four boxes containing 1024 polygons each.
- 4x4 means 256 boxes containing 16 polygons each.

The initial scene was fully inside the window (zoom = 1.0). Then the scene was zoomed by 1.11, forcing a few of the rectangles and the outmost boxes out of the window. The window was then zoomed with factors of 1.67, 5
and 20 of the original size, reducing the number of polygons on screen to only a few. The actual scenes with the two highest magnification factors are shown in Fig.3.5. The more complicated scenes are too complex to render on a low-resolution raster screen (becomes almost solid black).

Fig. 3.5
Test images originally with 64x64 rotated rectangles zoomed with a factor of 5 and 20, respectively. Top window in each figure is generated by Liang-Barsky algorithm. The lower and middle windows show the output of the Stacking Clipper, with and without boxing, respectively. (Screen dump from X Window System.)
Fig. 3.6
Times in seconds for clipping scene with 4096 rotated rectangles on a Sun-3/50 workstation with floating point unit Motorola 68881.

Fig. 3.6 shows the result of running this test. The following comments are related to each curve:

1) Liang-Barzky algorithm with no boxing. The curve falls gradually as zoom is increased from 1.0 to 20.0 and polygons change from being fully inside the window to being fully outside. The algorithm seems to be about 50% faster on rejectable polygons, than on clipped polygons or polygons fully inside screen.

2) Stacking Clipper with single box containing 64x64 rectangles. In all but the zoom = 1 situation, the single box goes outside the window on all four sides. Thus the Stacking Clipper is working as if no boxing were present. The resulting curve shows a performance very similar to the Liang-Barsky, with the latter slightly ahead when many polygons are clipped, while the Stacking Clipper is at a slight advantage when polygons can be rejected.
3) Stacking Clipper with four boxes containing 32x32 rectangles each. When zoomed, the boxes will go outside on two sides each. This leaves the Stacking Clipper with just two clipping boundaries instead of four. The resulting curve shows (as expected) a speed-up of approximately 50% compared with 1).

4) Liang-Barsky algorithm with 256 boxes containing 4x4 boxes each. Boxing is performed to avoid spending time clipping when boxes are fully on or fully off screen. Compared with the non-boxing Liang-Barsky (1) this greatly improves performance.

5) Stacking Clipper with 256 boxes containing 4x4 rectangles each. Using the boundary bits from the boxing test, this effectively reduces the clipping time by nearly 90% compared with the non-boxing Liang-Barsky algorithm, and by approximately 40% compared with the Liang-Barsky algorithm with boxing.

Fig. 3.7
Times in seconds for clipping scene with 4096 rotated rectangles on a Sun-3/50 workstation without floating point unit.
The tests on a workstation with slower floating point operations, make the Stacking Clipper really shine compared to the Liang-Barsky algorithm. While the execution time for the Stacking Clipper is approximately doubled, when going to the less powerful machine, the execution time of the Liang-Barsky algorithm is increased by a factor of four.

### 3.1.3. Optimal Box Size

The box size will have to be chosen to minimize the total cost of the boxing test and clipping for the actual scene.

With large boxes, off-screen clusters of polygons can effectively be rejected, while many polygons are likely to be categorized as intersecting the window boundary (even though many of these polygons may actually be completely inside the window or completely outside the window). With smaller boxes, a better classification of each polygon is possible, reducing the cost of the clipping itself, but at the cost of performing more boxing tests.

The best solution, when scenes are complicated, is to have extent trees with a reasonable branching ratio. The criteria for introducing a new level in an extent tree is that the cost of the added extent test must be smaller than the anticipated reduction in extent tests on the level below.
Fig. 3.8
Times in seconds for clipping scene as a function of box size with the Stacking Clipper.

The figure above shows how execution time depends on box size. With this constructed environment, the optimal box size was around 4x4 for all tested zoom factors. From the curves, there seems to be a slight tendency in favour of using larger boxes with larger zoom factors. Nested boxes were not tested.

Fig. 3.9
Result of boxing test with different zoom factors when box contains 4x4 polygons.
The figure above shows the result of the extent tests with different zoom factors and boxes containing 4x4 polygons each. More than 75% of the boxes were categorized as either fully inside or outside. Note that only a few boxes intersected 1 or 2 boundaries (Clip 1 and Clip 2, respectively), while none of the boxes intersected more than two boundaries.

3.2. Three-Dimensional Benchmarks

For the three-dimensional benchmarks, an implementation of the Sutherland-Hodgman algorithm was chosen for comparison. The "C"-coded algorithm was taken from a public domain pre-release of PEX, produced by S. Thomas et al. at the University of Michigan. The code was in the "contributed" section of the X Window System Distribution (X.V11R3), autumn 1988 [SCHIEILER86]. It is a straightforward implementation using separately coded subroutines for each stage. The actual code is shown in Appendix E.

All tests were carried out on a Sun-3/50 with a Motorola 68881 floating point processor, running as single user. Timing tests were performed as described in Ch. 3.1. The time measured is the time for traversing the scene data structure, clipping and transforming points to screen space. All points in the scene are transformed, regardless of the result of the boxing test.

The data base was part of a model of Singapore harbour, manually digitized and with manually defined extents. The model was aimed at a ship navigation simulator and consists of a simplified landscape and some stylized buildings at the quays.

The database contains 1108 triangles, divided into 53 clusters or extents. This gives an average of 21 polygons per box.

In the test, the model was viewed from an elevation of 20 degrees, the clipping was performed with and without depth clipping (hither and yon). The scene was zoomed in from a first view which contained the whole scene. The zooming was performed with a magnification of $\sqrt{2}$ between each measurement, up to a zoom factor of 256.
With the initial zoom factor, the scene contains a large number of very small polygons. The scene is shown with different zoom factors on colour plates 2a, 2b, and 2c.

The actual distribution of polygon sizes is shown below (after projection to screen space of 512x512 pixels).

![Polygon size (area) distribution with zoom factor 1.0.](image)

**Fig. 3.10.**

Polygon size (area) distribution with zoom factor 1.0.

Some characteristic parameters for the scene (unit is pixels):

Minimum polygon size: $2.3 \times 10^{-4}$

Maximum polygon size: 818

Mean polygon size: 57

Median polygon size: 3.7

The tests were run with both algorithms, with and without boxing and with the improved three-dimensional extent testing algorithm described in Ch. 2.2.3.
Fig. 3.11.
Extent test results for Singapore harbour scene. Numbers shown on abscissa axis are squared zoom factors.

The histogram in the above figure shows the results of the boxing tests. There is a gradual change from polygons all-inside to polygons all-outside, with the most dramatic change around zoom = 6. Clipping increases to a small peak in this area and then decreases. For all zoom factors tested, more than 70% of the boxes are either fully outside or inside the clipping volume and less than 15% of the boxes need to be clipped at more than one boundary.

The timing results when depth clipping is not used are shown below.
Fig. 3.12.
Timing results for Singapore harbour scene with Sutherland-Hodgman and Stacking Clipper algorithm, when depth clipping is disabled. Numbers are relative to the scene processing time with all clipping disabled.

The tests are discussed one by one:

1) All polygons were passed through the Sutherland-Hodgman algorithm without any prior boxing tests. The curve shows a slight decline with a stronger decline in the area around zoom = 6. When clipping is turned on, the scene processing time more than doubles (scan conversion is excluded).

2) The boxing test was included, and used to discard the polygons for boxes fully outside, or just by-pass the clipper for boxes fully inside. For partly clipped boxes, the resulting polygons were passed through the Sutherland-Hodgman clipping algorithm. The boxing test reduces the clipping overhead from ≈ 100% to 15%. For all zoom factors, the boxing test reduces the processing time by 50% for objects mostly on screen and by 30% for objects mostly off screen.
3) The Stacking Clipper without any boxing information is about 25% faster than the Sutherland-Hodgman algorithm for objects mostly on screen, and in the same range for zoom factors \( \geq 8 \).

4) When boxing test with boundary intersection bits is used, the Stacking Clipper is about 50% faster than the Sutherland-Hodgman algorithm applied without prior boxing, and up to 20% faster than the Sutherland-Hodgman algorithm with boxing test. The improvement is largest when most polygon-boundary intersections occur, that is in the zoom = 6 area.

5) The improved boxing test used with the Stacking Clipper improves the processing time by approximately 10% compared with the results in 4).

![Graph showing improvement in %](image.png)

**Fig. 3.13.**

Improvement in % when using the efficient boxing test along with the Stacking Clipper, compared to using the Sutherland-Hodgman algorithm.

All in all, when using the efficient boxing test along with the Stacking Clipper, we get a significant performance improvement over the non-boxing
Sutherland-Hodgman algorithm. The improvement is in the order of 50%, and seems to be present for all zoom factors.

![Graph](image)

Fig. 3.14.
Timing results for Singapore harbour scene with Sutherland-Hodgman and Stacking Clipper algorithm, depth clipping is enabled. Numbers are relative to the scene processing time with all clipping disabled.

When hither and yon clipping is enabled, the Stacking Clipper without prior boxing is slightly slowed down (curve 3). The improvement from the Sutherland-Hodgman algorithm to the Stacking Clipper with boxing is still in the 50% range.

### 3.3. Generated Code Size

The implementation resulted in amazingly compact code. The generated code from the three-dimensional Stacking Clipper was only 20% of the PEX Sutherland-Hodgman code, and the generated code for the two-dimensional
Stacking Clipper was slightly smaller than the code for the Liang-Barsky algorithm†.

† On a Sun-3/50 with optimization flag on, and with code generation for 68881 floating point processor enabled.
4. ADAPTABILITY


Normally, geometry input to the viewing transformation will lie in the \( w = 1 \) 3-flat\(^\dagger\) in homogeneous coordinates. Scaling, transformations and rotations used for modelling will not take the geometry out of the \( w = 1 \) 3-flat. Only a perspective transformation transforms objects out of this 3-flat.

However, this is not the case when so-called rational parametric curves are used in the modelling process. NURBS (Non-uniform Rational B-Splines), for instance, are popular when modelling free-form surfaces. They are one of the available graphic output primitives in the PEX/PHIGS+ proposal.

With rational parametric curves, the homogeneous coordinate is a function of the parameter in the same way as the \( x, y, \) and \( z \) coordinates.

\[
	xw = P(t) \\
yw = Q(t) \\
w = R(t).
\]

When \( P(t), Q(t) \) and \( R(t) \) are quadratic polynomials, any conic section can be represented [BLINN78].

For instance, a circle segment in the first quadrant can be defined by:

\[
	xw = 1 - t^2 \\
yw = 2 * t \\
w = 1 + t^2
\]

when \( 0 \leq t \leq 1 \) [FAUX79].

\(^\dagger\) In linear algebra a 3-flat is a three-dimensional subspace offset from the origin [FRALEIGH87].
The $w$ coordinate can also become negative. When the $w$ coordinate varies from a positive value to a negative value along the curve, the curve passes through infinity. The two segments of a hyperbola can be traced out by:

$$xw = 1 + t^2$$
$$yw = \sqrt{2} \cdot t$$
$$w = 1 - t^2$$

when $-\infty \leq t \leq +\infty$ [BLINN78]. The transition from one segment to another happens for $t = \pm 1$, when the hyperbola passes through infinity ($w = 0$).

In a similar - but simpler - way, the two parts of an "extern" line segment between $(1/2, 0)$ and $(-1/2, 0)$ can be represented by:

$$xw = 1$$
$$yw = 0$$
$$w = 2 - 4 \cdot t$$

for $0 \leq t \leq 1$ (see the figure below).
Fig. 4.1

a) A rational line \((xw = 1, yw = 0, w = 2 - 4 * t)\) shown in the XW plane, representing an "extern" line with two visible segments when projected onto the w=1 plane.
b) The XW clipping region.
c) The line after clipping with its projected line segment.
d) The mirrored line after clipping with its projected line segment.

The clipping algorithm as defined in Ch. 2 cannot handle these special cases properly. This is obvious when considering the desired output from the "external" line segment AB in Fig. 4.1.a. The line segment is clipped on the \(-x + w = 0\) boundary, outputting only one of the external lines (between \(t=0\) and \(t=0.25\)) as seen in Fig. 4.1.c. This does not come as a surprise, since the clipper is not designed to output more than a single line segment.

A method to handle rational curves properly, is proposed by [SUTHERLAND74]. When an object has homogeneous coordinates on both sides of the \(w = 0\) plane, the object is passed through the clipping algorithm.
twice: once as defined and once with all coordinates multiplied by -1 (see A"B" on Fig. 4.1.d). This will give the desired result (the line segment between $t = 0.75$ and $t = 1$ on Fig. 4.1.d).

However, there still exists the possibility of singularities when an object passes through the $w = 0$ plane. Herman and Reviczky suggest clipping to subspaces $w \leq -\varepsilon$ and $\varepsilon \leq w$ [HERMAN88].

In our case, a good solution will be to use a combination of these suggestions:

1. Clip all geometry on the $w \geq \varepsilon$ halfspace.

2. The "inside" output of this is passed through the normal clipping pipeline.

3. The "outside" output is multiplied by -1 and again clipped on the $w \geq \varepsilon$ halfspace before being passed through the clipping pipeline. (If the clipper has no "outside" output, any object intersecting the $w - \varepsilon = 0$ plane should be inverted and passed through the clipper again.)

4.2. General Halfspace Clipping

By performing the classification (intersection) comparisons on general line or plane equations, the clipping engine can be set up for clipping inside general convex polygons/polyhedra instead of just rectangles/pyramids. Moreover, as described earlier, the "outside" part of the polygon can easily be taken care of and passed through other clipping stages. Any clipping region described by a Boolean expression of halfspaces can thus be mapped to a network of clipping stages.
4.3. Viewing Frustum Clipping in Object Space

a) Clipping after transformation

![Diagram of clipping after transformation]

b) Clipping before transformation

![Diagram of clipping before transformation]

Fig 4.2
With the Stacking Clipper, clipping can be performed after transformation a) or before transformation b).

As clipping may well be performed on general planes, in the Stacking Clipper, clipping can be done prior to transformation rather than after. That is, we can transform the clipping planes into object space and perform the clipping there. The advantage of this is that points that are rejected during clipping need not be transformed into screen space.
The transformed clipping planes for the normalized viewing frustrum are those computed for the fast boxing algorithm in Ch. 2.X.

However, the work involved in computing the distances to general planes (columns in the concatenated matrix $\mathbf{T C}$) is larger than that for computing the distances to the planes in the normalized viewing frustrum. These distance computations are a major part of a clipping algorithm.

When a boxing preprocessor gets rid of off-screen geometry prior to any further processing of it, the advantage of doing clipping in object space is reduced.

As a conclusion we can say that the Stacking Clipper can be used for clipping in object space, but that with the integrated boxing/clipping algorithm this does not increase the speed of the presentation.

### 4.4. Modelling Clip

So far, we have only considered clipping to the (convex) viewing volume. Modelling clip may be regarded as a generalization of this problem. While in view volume clipping, graphics are clipped to a simple convex body, modelling clip allows the clipping volume to be specified as combination of halfspaces. In PHIGS (and PEX), the clipping volume is defined as an intersection of a list of halfspaces. More generally, an arbitrary Boolean expression of halfspaces could be defined.

First, we will comment on PHIGS's modelling clip. Then general clipping regions will be considered.

The PHIGS user specifies a modelling clip halfspace with a point $\mathbf{P}$ and a normal vector $\mathbf{n}$ in the Modelling Coordinate System (MC). The halfspace should be transformed into the World Coordinate Space (WC) using the current modelling transformation $\mathbf{M}_p$. Logically, modelling clipping will be carried out in WC. Objects to be clipped will thus be transformed by their own modelling transformation $\mathbf{M}_o$ prior to clipping. It should be noted that the object's modelling transformation is not necessarily the same as the clipping halfspace's modelling transformation. Modelling transformations are specified as a general $4 \times 4$ matrix and will not necessarily be affine.
transformations. (Affine transformations are those that can be expressed as a 3x3 matrix and a translation vector.)

Herman and Reviczky [HERMAN88] suggest carrying out this clipping in homogeneous coordinates (prior to division by the homogeneous coordinate) to avoid singularity problems. The clipping halfspace is then four-dimensional and the bounding plane contains the origin.

In four dimensional space, the boundary of the halfspace is a subspace P3. Herman suggests a method for finding the equation of the subspace P3 as follows:

1. Determine the three-dimensional plane S in MC from n and P.
2. Select three non co-linear points S_i lying in S and a point E lying in the halfspace.
3. Transform S_i and E by the linear transformation M_p. Denote the transformed points S'_i and E'. As a result of this transformation, the points will be in a three-dimensional subspace of the four-dimensional space. S', and E' will be used to define the corresponding halfspace.
4. The points S'_i plus the point O (origin) define the subspace P3.
5. The equation of P3 can be derived by determining a normal vector to P3 n4 = (n1, n2, n3, n4) and an arbitrary point X = (x1, x2, x3, x4). The normal vector n4 is computed using a special four-dimensional cross-product [HERMAN87] of the three four-dimensional vectors OS'_i.

It seems, however, that the coefficients of the four-dimensional clipping halfspace can be found without explicitly selecting points in the plane, and without doing four-dimensional cross-products:

1. Let the four-dimensional clipping halfspace bounded by P3 be defined by (this is the equation of a hyperplane or a 3-flat in E^4):

   \[ a_p x + b_p y + c_p z + d_p w + e_p = 0 \]
We will try to find the coefficients for the hyperplane.

2. Let \( \rho \) be a three-dimensional plane defined by a normal vector \( \mathbf{n} \) \((n_x, n_y, n_z)\) and a point \( P_0(x_0, y_0, z_0) \) in the plane.

\[
\rho = [n_x, n_y, n_z, d]^T, \text{ where } \\
d = -(n_x \times x_0 + n_y \times y_0 + n_z \times z_0)
\]

3. Let \( S_i \) be a freely chosen point in the plane \( \rho \) represented by a homogeneous coordinate row vector \((x_i, y_i, z_i, 1)\). Then we must have:

\[
S_i \cdot \rho = 0
\]

and assuming \( M_p \) is non-singular:

\[
S_i \cdot (M_p \cdot M_p^{-1}) \cdot \rho = 0, \text{ or }
\]

\[
S_i' \cdot (M_p^{-1} \cdot \rho) = 0
\]

4. The hyperplane \( P3 \) can be defined by four points in \( E^4 \), or alternatively, by a 2-flat and an additional point not in the 2-flat. Since \( O(0, 0, 0, 0) \) is on the subspace \( P3 \) and not in \( S' \) we must have:

\[
e_p = 0
\]

and \( P3 \) can be described by a four-element coefficient vector:

\[
\gamma = [a_p, b_p, c_p, d_p]^T
\]

5. The arbitrarily chosen transformed point \( S_i' = S_i \cdot M_p \), represented by a row vector \((x_i', y_i', z_i', w_i')\) must be on the hyperplane \( P3 \), and we must therefore have:

\[
a_p x_i' + b_p y_i' + c_p z_i' + d_p w_i' = 0, \text{ or more compactly: }
\]

\[
S_i' \cdot \gamma = 0
\]

6. Since the coefficient vector of \( \rho \) transformed by \( M_p \) satisfies the latter for all \( S_i' \), the coefficient vector of the clipping hyperplane can be chosen as the four-element coefficient vector:
\[ \gamma = (M_p^{-1} \cdot \rho) \]

but with a fifth element \( e_p = 0 \) added.

A point \( E \) inside the halfspace would satisfy:

\[ E \cdot (M_p \cdot M_p^{-1}) \cdot \rho > 0 \]

\[ E' \cdot \gamma > 0 \]

The distance measure to use to determine if a point \( P \), transformed by its modelling transformation \( M_0 \) is inside or outside the four-dimensional clipping halfspace is:

\[ 1 = P \cdot M_0 \cdot M_p^{-1} \cdot \rho \]

Normally, it will be worthwhile to concatenate \( (M_0 \cdot M_p^{-1} \cdot \rho) \) into a four-element vector prior to extent testing or clipping the polygons inside the object. This is equivalent to transforming the clipping plane from \( MC(1) \), into \( WC \), and back into \( MC(2) \) for doing the clipping there.

Boxing against the modelling clip region can be done efficiently using the technique discussed in Ch. 2.4.

### 4.5. Building Divider Networks

A concave clipping region as shown in the figure below can be represented as:

\[ \text{in}(P, \alpha) \text{ and } \text{in}(P, \beta) \text{ and } (\text{in}(P, \chi) \text{ or } \text{in}(P, \delta)) \text{ and } \text{in}(P, \varepsilon) \]

where the homogeneous boundary vectors could be:

\[ \alpha = [0, 1, -y_1]^T, \]
\[ \beta = [1, 0, -x_1]^T, \]
\[ \chi = [-1, -1, c_1]^T, \]
\[ \delta = [1, -1, c_2]^T, \]
\[ \varepsilon = [-1, 0, x_2]^T, \]
x1, x2, y1, c1, c2 are constants, their magnitude being the distance from (0, 0) to the boundary line.

This could be realized as a "network" of clipping stages as follows:

Fig. 4.3
Clipper network b) for implementing the concave clipping region in a). Each stage has its "inside" pipe going downwards and its optional "outside" pipe going to the right.

This would of course require one separate "collector" stage for each parallel branch of clippers. Moreover, unless handled by a separate "fusion"-stage, polygons may be split into smaller sub-polygons by the or-stages of the network.

However, this may well be considered as a "feature" rather than a bug, as it guarantees that if a convex polygon is input to the clipper, convex polygons (one or more) will be output from the clipper. If a hidden surface algorithm is the recipient of the clipped polygons, convex polygons are mandatory or at least desirable for shading interpolation.
Implementing an optional "outside" polygon output channel in the Stacking Clipper, and making it possible to handle general clipping networks, is straightforward. Instead of just sending output to stage "n+1", stage no. "n" would have to represent the next two output stages explicitly, addressing stacked output to the "inside" or "outside" stage, respectively.

4.6. Clipper Network From Boolean Expression

In PEX, the modelling clip can be specified as lists of halfspaces combined with some operator. The semantics of this operator is, however, not described in the current PEX documents [PEX88a], [PEX88b].

With general Boolean expressions of halfspaces, one can in principle define any clipping region.

Automatic generation of clipper networks from general Boolean expressions is not trivial, if something like an "optimal" network structure is desired.

A simple solution, however, is to transform the general Boolean expression into canonical sum-of-products form\(^\dagger\). (A sum is here formed by the or operator, and a product by the and operator.) The sum-of-products form can be mapped directly into parallel pipelines of clipping stages, each implementing one product (a convex region). An input polygon would have to be input to all clipping stages. Boxing, as previously described, could be performed for each clipping pipeline to reduce the necessary work.

A sum-of-products form of the concave polygon above would be:

\[
\text{(in}(P, \alpha) \text{ and in}(P, \beta) \text{ and in}(P, \chi) \text{ and in}(P, \varepsilon)) \text{ or (in}(P, \alpha) \text{ and in}(P, \beta) \text{ and in}(P, \delta) \text{ and in}(P, \varepsilon))
\]

\(^\dagger\) Algorithms for transforming general Boolean expressions into sum-of-products canonical form, can be found for instance in the GAS system [ZACHRISEN89b], where it is used in the context of solid modelling.
Fig. 4.4

b) Clipper network for implementing the sum-of-products form of the concave clipping region a).

This is mapped into the network shown in the figure above. It should be obvious that the hand-coded network from Fig. 4.3 will perform better.

The biggest shortcoming of this is that several of the pipelines will probably output the same polygons. In the concave polygon example, both pipelines will produce output for the clipping region which is the intersection of the two convex regions i.e. the region described by:

\[
\text{in}(P, \alpha) \text{ and } \text{in}(P, \beta) \text{ and } \text{in}(P, \varepsilon) \text{ and } \\
\text{in}(P, \delta) \text{ and } \text{in}(P, \chi)
\]

Introducing a special buffering and sequencing component makes possible a more direct Boolean expression to network transformation. This component receives output vertices from multiple stages and stores them in a buffer.
When all predecessors have been closed, the sequencer sorts the received vertices and outputs vertices one polygon at a time.

Using the buffer/sequencer and divider stages, we can construct "multipliers" and "adders" with one or two outputs†.

![Diagram](image)

**Fig. 4.5**

General a) "multiplier", b) "adder" and c) "dividing multiplier" networks built with dividers.

The figure above shows general adder and multiplier networks.

† Note: The arithmetic terms sum, product, adder, multiplier and the arithmetic operators + and * or logic operators and and or are used instead of the more correct terms intersection and union and the operators ∩ and ∪.
Below is shown a grammar which makes it possible to build a divider network as the Boolean expression is parsed. To improve the constructed network, the grammar handles single output components (clippers) separately from two-output components (dividers). In particular, a single output multiplier is much simpler than a multiplier with output and inverted output. The "semantic rules" in the grammar refer to Fig. 4.7. The grammar can easily be modified to generate multi-operand (rather than just two-operand) adders and multipliers. If the Boolean expression contains negations, these can be moved down to the halfspace level using De Morgan’s theorem (see [ZACHRISEN89b]).

\[
\begin{align*}
<\text{clippingregion}> & ::= <\text{expression1}> \\
& | \\
<\text{expression1}> & ::= <\text{expression2}> \ '+' <\text{term1}> \quad \{ \text{ADDER(a, b)} \} \\
& | <\text{term1}> \\
<\text{term1}> & ::= <\text{term1}> '***' <\text{primary1}> \quad \{ \text{MULTIPLIER(a, b)} \} \\
& | <\text{primary1}> \\
<\text{primary1}> & ::= '(' <\text{expression1}> ')') \\
& | \quad \text{HALFSPACE} \quad \{ \text{CLIPPER(a)} \} \\
<\text{expression2}> & ::= <\text{expression2}> \ '+' <\text{term2}> \quad \{ \text{DIV ADDER(a, b)} \} \\
& | <\text{term2}> \\
<\text{term2}> & ::= <\text{term2}> '***' <\text{primary2}> \quad \{ \text{DIV MULTIPLIER(a, b)} \} \\
& | <\text{primary2}> \\
<\text{primary2}> & ::= '(' <\text{expression2}> ')') \\
& | \quad \text{HALFSPACE} \quad \{ \text{DIVIDER(a)} \} \\
\end{align*}
\]

Fig. 4.6
Grammar for building clipper networks from general Boolean expressions. Semantic rules are shown in curly brackets and refer to basic network operations on the figure below.
Fig 4.7
Basic construction blocks used by the network-building parser's semantic rules; a) CLIPPER, b) DIVIDER, c) MULTIPLIER, d) DIVIDING MULTIPLIER, e) ADDER, f) DIVIDING ADDER.

4.7. Clipper Networks from Concave Polygons

If specified as a general polygon (or polyhedron), the clipping region can be mapped into a network of clipping stages. The simplest solution would be to divide concave clipping regions into disjoint convex regions. These can readily be implemented by parallel pipelines of clipping stages, each implementing a convex region.

For transforming concave (or degenerate) polygons into disjoint convex polygons, the clipping algorithm can provide help. Basically, one can - when identifying a concave angle between edges, or edges crossing each other (degenerate polygons) - clip the polygon along one of the offending edges into two disjoint regions. Some additions to the clipping algorithm are necessary to put together convex polygons on output rather than new degenerate polygons. A treatment of this subject is given in [SUTHERLAND74].
5. CONCLUSIONS

The Stacking Clipper algorithm is not well suited to tasks where the clipped polygon is large compared to the window, for instance, pixel clipping. But, the Stacking Clipper performs as well or better than other known clipping algorithms for "typical" scenes where the window size is big compared to the displayed polygons. Together with the new extent testing algorithm, it provides a very powerful solution for traversing large geometric environments. The flexibility of the Stacking Clipper algorithm makes it suitable for three-dimensional view pyramid clipping as well as for clipping against more complex Boolean combinations of halfspaces. It performs best when some structural information in the rendered scene can be utilized.

The homogeneous coordinate Stacking Clipper and the new extent testing algorithm have proven to be powerful and robust parts of the Moviebox rendering system [ZACHRISEN89a, ZACHRISEN89f].
APPENDIX A. COLOUR PLATES

The colour illustrations in this volume are all produced as photographs taken from the face of the Hitachi monitor of a Sun-4/110 workstation with CG4 frame buffer. The frame buffer is limited to 256 simultaneous colours. Moviebox divides the colour table into slices for each material to be rendered in order to provide smooth shading. All images have been rendered by Moviebox in 1150x900 pixels resolution.

The photographs were taken using 100 ASA colour print film, using a single-lens reflex camera with 50 mm lens, 1/2 second, f 8. In the final printed report, photos have been copied on a Canon Colour Copier.

Plate 1.
The boundary representation of a solid is revealed when "hither" clipping is on (left image). (Model courtesy of APS project, SINTEF.)

The following colour plates have been generated from a Singapore harbour data model which is courtesy of Norcontrol Simulations.

Plate 2a.
Test image of Singapore Harbour with zoom factor = 1.0.

Plate 2b.
Test image of Singapore Harbour with zoom factor = 2.5.

Plate 2c.
Test image of Singapore Harbour with zoom factor = 8.0.
APPENDIX B. A FAST BOXING ALGORITHM

/*
* boxing.c
* This file contains an extent testing algorithm, working in
* homogeneous coordinate space.
* It is particularly suited for the Stacking Clipper, but may be
* used with other clipping algorithms as well.
*
* Author: Morten Zachrisen
*        ELAB - RUNIT
*        7034 Trondheim-NTH
* Date: 1989
* Copyright (c) 1989 Morten Zachrisen
*/

typedef int Bits;            /* For bits (16 or more) */

struct _mtransf {
    float tr_mat[4][4];
};

/* cpm_trbox:
* Reverse transform clipping limits by specified matrix.
* Test against extent.
* Return clipping status bits if box intersects,
* 0 if inside viewing pyramid,
* -1 if all outside.
*/
Bits cpm_trbox(vtran, extent, clipstat)
struct _mtransf *vtran;        /* Transform to be applied */
float *extent;                  /* Untransf. extent: xmin, xmax,.. */
Bits clipstat;                  /* Bits designate limits to test */
{
    register struct _mtransf *t = vtran;
    register Bits clipbits; /* Accum. intersecting boundaries */
    register Bits stat;
    float pl[4];            /* Plane to check against */

    /* No clipping - status OK */
    if (clipstat == 0)
        return 0;

    clipbits = 0;

    /* Inversely transform clipping limits:
    * left:  [1 0 0 1]
    * right: [-1 0 0 1]
    * bottom: [0 1 0 1]
    * top:   [0 -1 0 1]
    * far:   [0 0 1 1]
    * near:  [0 0 -1 0]
    */
if (clipstat & 1) {
    /* left */
    pl[0] = t->tr_mat[0][0] + t->tr_mat[0][3];
    pl[1] = t->tr_mat[1][0] + t->tr_mat[1][3];
    pl[2] = t->tr_mat[2][0] + t->tr_mat[2][3];
    pl[3] = t->tr_mat[3][0] + t->tr_mat[3][3];

    if ((stat = clipExtent(extent, pl)) < 0)
        return -1;
    else if (stat > 0)
        clipbits |= 1;
}

if (clipstat & 2) {
    /* right */
    pl[0] = -t->tr_mat[0][0] + t->tr_mat[0][3];
    pl[1] = -t->tr_mat[1][0] + t->tr_mat[1][3];
    pl[2] = -t->tr_mat[2][0] + t->tr_mat[2][3];
    pl[3] = -t->tr_mat[3][0] + t->tr_mat[3][3];

    if ((stat = clipExtent(extent, pl)) < 0)
        return -1;
    else if (stat > 0)
        clipbits |= 2;
}

if (clipstat & 4) {
    /* bottom */
    pl[0] = t->tr_mat[0][1] + t->tr_mat[0][3];
    pl[1] = t->tr_mat[1][1] + t->tr_mat[1][3];
    pl[2] = t->tr_mat[2][1] + t->tr_mat[2][3];
    pl[3] = t->tr_mat[3][1] + t->tr_mat[3][3];

    if ((stat = clipExtent(extent, pl)) < 0)
        return -1;
    else if (stat > 0)
        clipbits |= 4;
}

if (clipstat & 8) {
    /* top */
    pl[0] = -t->tr_mat[0][1] + t->tr_mat[0][3];
    pl[1] = -t->tr_mat[1][1] + t->tr_mat[1][3];
    pl[2] = -t->tr_mat[2][1] + t->tr_mat[2][3];
    pl[3] = -t->tr_mat[3][1] + t->tr_mat[3][3];

    if ((stat = clipExtent(extent, pl)) < 0)
        return -1;
    else if (stat > 0)
        clipbits |= 8;
}

if (clipstat & 16) {
    /* far */
    pl[0] = t->tr_mat[0][2] + t->tr_mat[0][3];
    pl[1] = t->tr_mat[1][2] + t->tr_mat[1][3];
    pl[2] = t->tr_mat[2][2] + t->tr_mat[2][3];
    pl[3] = t->tr_mat[3][2] + t->tr_mat[3][3];

    if ((stat = clipExtent(extent, pl)) < 0)
        return -1;
    else if (stat > 0)
        clipbits |= 16;
if (clipstat & 32) {
    /* near */
    pl[0] = t->tr_mat[0][2];
    pl[1] = t->tr_mat[1][2];
    pl[2] = t->tr_mat[2][2];
    pl[3] = t->tr_mat[3][2];

    if ((stat = clipExtent(extent, pl)) < 0)
        return -1;
    else if (stat > 0)
        clipbits |= 32;
}

return clipbits;

int
clipExtent(extent, plane)

float *extent;  /* Extent of object (lox, hix, loy... */
float *plane;   /* Transformed plane [4] */

{  
#define C_INSIDE 1
#define C_OUTSIDE 2

register Bits stat = 0;  /* Accumulating clipping codes */
float dist[4];            /* Temporary storing distances */
float d;
float dsize;
register float *rdistpt;
register int i;

/* Left, lower, rear */

if (d > 0.0)
    stat |= C_INSIDE;
else if (d < 0.0)
    stat |= C_OUTSIDE;
if (stat == (C_INSIDE|C_OUTSIDE))
    return 1;
dist[0] = d;

/* Right, lower, rear */
d = dist[0] + (extent[1] - extent[0]) * plane[0];

if (d > 0.0)
    stat |= C_INSIDE;
else if (d < 0.0)
    stat |= C_OUTSIDE;
if (stat == (C_INSIDE|C_OUTSIDE))
    return 1;
dist[1] = d;

/* Upper, rear points */
dsize = (extent[3] - extent[2]) * plane[1];
for (rdistpt = dist, i = 0; i < 2; i++) {
    d = *rdistpt++ + dsize;
if (d > 0.0)
    stat |= C_INSIDE;
else if (d < 0.0)
    stat |= C_OUTSIDE;
if (stat == (C_INSIDE|C_OUTSIDE))
    return 1;
rdistpt[1] = d;
}

/*@ Front points */

for (rdistpt = dist, i = 0; i < 4; i++) {
    d = *rdistpt++ + dsize;
    if (d > 0.0)
        stat |= C_INSIDE;
    else if (d < 0.0)
        stat |= C_OUTSIDE;
    if (stat == (C_INSIDE|C_OUTSIDE))
        return 1;
}
if (stat == C_OUTSIDE)
    return -1;
else
    return 0;
APPENDIX C. STACKING CLIPPER

This Appendix contains the complete code of the Stacking Clipper. The code consists of a generic "clip_engine" along with definitions for either a two-dimensional or a three-dimensional clipper.

The three-dimensional code variant is generated after a
#define threed
, otherwise a two-dimensional version is generated.

The two-dimensional intersection computation is handled by a macro, while the three-dimensional intersection computation is handled by a function.

In the MOVIEBOX renderer, the three-dimensional version of this code is used, but with colour interpolation added in intersection computation function (data type "Color").

/*@
 * clipper.c - a special, highly efficient
 * stack machine polygon clipper - a transformation
 * of the Sutherland-Hodgman algorithm.
 * It can be generated in versions for 2-D or 3-D clipping.
 *
 * Author: Morten Zachrisen
 * RUNIT
 * 7034 Trondheim-NTH
 * Date: May 21 1987
 * Copyright (c) 1987 Morten Zachrisen
 *
/*@

/* Some "syntactic sugar": */
#define FOREVER for (; ;)
#define NIL 0L /* Pointer NIL */
#define OBJ_COPY(a, b) ((b) = (a))
typedef int Bits; /* For bits only (16 or more) */

#ifdef threed
#define N_CSTAGES 6 /* No of clipping stages: 3D = 6 */
struct vertex {
  float x, y, z, h;
  Color pt_col;
};
#else /* not threed */
#define N_CSTAGES 4 /* No of clipping stages: 2D = 4 */
struct vertex {
  float x, y;
};
#endif /* not threed */
C2 Volume 3 - Fast Algorithms for Polygon Clipping and Boxing

/* Clipping status values: */
#define ST_CLOSED 0 /* No clipping status computed */
#define ST_IN 1 /* Inside boundary */
#define ST_OUT -1 /* Outside boundary */

static struct cstage { /* ST_CLOSED, ST_IN or ST_OUT */
    int status; /* Previous distance to boundary */
    struct vertex firstp; /* First point in stage - for closing */
    struct vertex prevp; /* Previous point in stage - for computing intersections */
    struct vertex isectp; /* Storage for computed intersection point */
    double (*clipproc)(); /* Handler for each clipping boundary. Usage: distance = (*clipproc)(pt) */
}
Clipstage[N_CSTAGES];

#ifdef thread
/* All clipping stages compute the distance from a point to the boundary. */
static double
clip_left(pt) register struct vertex *pt; /* Point to test against boundary */
{
    return (pt->h + pt->x);
}

static double
clip_right(pt) register struct vertex *pt; /* Point to test against boundary */
{
    return (pt->h - pt->x);
}

static double
clip_bott(pt) register struct vertex *pt; /* Point to test against boundary */
{
    return (pt->h + pt->y);
}

static double
clip_top(pt) register struct vertex *pt; /* Point to test against boundary */
{
    return (pt->h - pt->y);
}

static double
clip_far(pt) register struct vertex *pt; /* Point to test against boundary */
{
    return (pt->h + pt->z);
}

static double
clip_near(pt) register struct vertex *pt; /* Point to test against boundary */
{
    return (-pt->z);
}
/*
 * clip_init:
 * Initialize clipping engine prior to first input polygon.
 * clip_engine resets itself between polys.
 */
clip_init()
{
    Clipstage[0].clipproc = clip_left;
    Clipstage[1].clipproc = clip_right;
    Clipstage[2].clipproc = clip_bott;
    Clipstage[3].clipproc = clip_top;
    Clipstage[4].clipproc = clip_far;
    Clipstage[5].clipproc = clip_near;
    Clipstage[0].status = ST_CLOSED;
    Clipstage[1].status = ST_CLOSED;
    Clipstage[2].status = ST_CLOSED;
    Clipstage[3].status = ST_CLOSED;
    Clipstage[4].status = ST_CLOSED;
    Clipstage[5].status = ST_CLOSED;
}

/*
 * Compute the intersection point given start and endpoint and
 * intersection point parameter (0.0 - 1.0).
 */
static void cs_isect(startpt, endpt, alpha, isectpt)
    register struct vertex *startpt, *endpt; /* Start and end vertices */
    float alpha; /* Fraction of intersection */
    register struct vertex *isectpt; /* Intersection point (output) */
{
    isectpt->x = startpt->x + (endpt->x - startpt->x) * alpha;
    isectpt->y = startpt->y + (endpt->y - startpt->y) * alpha;
    isectpt->z = startpt->z + (endpt->z - startpt->z) * alpha;
    isectpt->h = startpt->h + (endpt->h - startpt->h) * alpha;
    /* Do colour interpolation if necessary... */
}
#else /* not threed */

/* Clipping boundaries */
static float clip_xleft = 0.0;
static float clip_xright = 0.0;
static float clip_ybottom = 0.0;
static float clip_ytop = 0.0;

/*
 * Sets window for two-dimensional clipping.
 */
set_window(xmin, xmax, ymin, ymax)
    float xmin, xmax, ymin, ymax;
{
    clip_xleft = xmin;
    clip_xright = xmax;
    clip_ybottom = ymin;
    clip_ytop = ymax;
    clip_init();
}

/* Boxing function returns -1 outside, 0 inside,
 * or boundary bits: 0x1 - 0xf indicating boundary intersections.
int
test_box(xmin, xmax, ymin, ymax)
float xmin, xmax, ymin, ymax;
{
    register int stat;
    if (xmax <= clip_xleft || xmin >= clip_xright ||
        ymax <= clip_ybottom || ymin >= clip_ytop)
        stat = -1;
    else {
        stat = 0;
        if (xmin < clip_xleft)
            stat |= 1;
        if (xmax > clip_xright)
            stat |= 2;
        if (ymin < clip_ybottom)
            stat |= 4;
        if (ymax > clip_ytop)
            stat |= 8;
    }
    return stat;
}

/*   All clipping stages compute the distance from a point to the
   boundary.                      */
static double
clip_left(pt)
register struct vertex *pt;  /* Point to test against boundary */
{
    return (pt->x - clip_xleft);
}

static double
clip_right(pt)
register struct vertex *pt;  /* Point to test against boundary */
{
    return (clip_xright - pt->x);
}

static double
clip_bott(pt)
register struct vertex *pt;  /* Point to test against boundary */
{
    return (pt->y - clip_ybottom);
}

static double
clip_top(pt)
register struct vertex *pt;  /* Point to test against boundary */
{
    return (clip_ytop - pt->y);
}

/*   clip_init:
   *   Initialize clipping engine prior to first input polygon.
   *     clip_engine resets itself between polys.     */
clip_init()
{
    Clipstage[0].clipproc = clip_left;
}
Clipstage[1].clipproc = clip_right;
Clipstage[2].clipproc = clip_bott;
Clipstage[3].clipproc = clip_top;
Clipstage[0].status = ST_CLOSED;
Clipstage[1].status = ST_CLOSED;
Clipstage[2].status = ST_CLOSED;
Clipstage[3].status = ST_CLOSED;
}
#endif /* not thread */

/* Macros for handling interface to polygon/vertex types etc. */
define CS_INPUT() ( (-n >= 0) ? inpoly++ : NIL ) /* Input next point */
define CS_OUTPUT(px) ( no++, OBJ_COPY(*(px), *outpoly++))

#define threed
#define CS_ISECT(startpt, endpt, alpha, isectpt) \ cs_isect(startpt, endpt, alpha, isectpt)
    /* Compute intersection */
#else /* not thread */
define CS_ISECT(startpt, endpt, alpha, isectpt) \ ((isectpt)->x = \ (startpt)->x + ((endpt)->x - (startpt)->x) * (alpha),\ (isectpt)->y = \ (startpt)->y + ((endpt)->y - (startpt)->y) * (alpha))
#endif /* not thread */

/* Macros for handling different stack etc. */
define CS_INIT(sp) ( (sp) = &Clipstack[N_CSTACK]) /* Init stack */
define CS_EMPTY(sp) ( (sp) >= &Clipstack[N_CSTACK]) /* Stack empty? */
define CS_PUSH(sp) ( ---(sp) ) /* Move stack pointer */
define CS_POP(sp) ( (sp)++) /* Pop stack */
define CS_DUP(sp) ( ---(sp), OBJ_COPY(sp[1], sp[0])) /* Duplicate top item */

/*
 * clip_engine:
 * Pops points from stack until stack empty, handling
 * active clipping boundaries one by one.
 * NOTE: Works on pointers to vertex record, thus all vertic
 * must be stored somewhere while on stack.
 * Input and output, stack operations (and interpolation) a:
 * taken care of by MACROS (CS_xxx).
 * Outputs no. of output vertices from pipeline.
 */
int clip_engine(clipmask, inpoly, n, outpoly)
    Bits clipmask; /* Polygon boundary bits
    (see above) */
    struct vertex *inpoly; /* Input polygon */
    int n; /* No. of input vertices */
    struct vertex *outpoly; /* Output polygon */
{
    define N_CSTACK (N_CSTAGES << 1) /* Maximum size */
    struct cstack [ /* Actual point (with attrib. s) */
    struct vertex *point;
    Bits clipmask; /* Bit set means clipping needed, 
    one bit for each stage,
    lab = next stage */
    struct cstage *stage; /* Stage pointer - points to sta:
    status struct */
Clipstack[N_CSTACK]; /* Clipping stack */

register struct vertex *endp; /* Current point to process */
register struct cstack *sp; /* Stack pointer */
register struct cstage *stagept; /* Stage record pointer */
register int endstatus; /* Status for current point */
register int startstatus; /* Status for prev. point */
float distance; /* Distance to boundary */
int no; /* No. of output vertices */

no = 0;
CS_INIT(sp);
FOREVER {
  if (CS_EMPTY(sp)) {
    /* Enter next polygon point onto stack */
    CS_PUSH(sp);
    sp->clipmask = clipmask;
    sp->stage = &Clipstage[0];
    sp->point = CS_INPUT(); /* Gets NIL at end */
  }

  /* Treat point on top of stack */
  if (sp->clipmask != 0) {
    register Bits thisclipmask;

    /* Skip all unnecessary stages */
    for (stagept = sp->stage,
        thisclipmask = sp->clipmask;
        (thisclipmask & 1) == 0; thisclipmask >>= 1)
      stagept++;

    startstatus = stagept->status;

    /* Update topofstack record (for next stage) */
    sp->stage = stagept + 1;
    sp->clipmask = thisclipmask >> 1;

    if (sp->point == NIL) {
      /* NIL - point is CLOSE-command. Pass CLOSE-command on to next stage after closing polygon in this stage if output has been generated */
      if (startstatus == ST_CLOSED)
        break;
      CS_DUP(sp);
      sp->point = &(stagept->firstp);
      stagept->status = ST_CLOSED;
    }

    endp = sp->point;

    /* Compute distance for point to boundary */
    distance = (*stagept->clipproc)(endp);
    endstatus = (distance < 0.0) ? ST_OUT : ST_IN;

    if (startstatus == ST_CLOSED) {
      /* First point to reach this far - store */
      OBJ_COPY(*endp, stagept->firstp);
      CS.Pop(sp);
      stagept->status = endstatus;
    } else if (endstatus != startstatus) {

register struct vertex *isectp = &stagept->isectp;
register struct vertex *startp = &stagept->prevp;
float alpha; /* Clipping fraction */
alpha = stagept->prevdistance /
(stagept->prevdistance - distance);

/* Compute intersection coordinates at alpha */
CS_ISECT(startp, endp, alpha, isectp);

if (endstatus == ST_IN)
    /* Going in - transmit intersection */
    /* and this endpoint to next stage */
    /* - push on stack in reverse order */
    CS_DUP(sp);

    /* Going in or out - pass on intersection */
    sp->point = isectp;
} else if (endstatus == ST_OUT)
    /* Both points outside - no output */
    CS_POP(sp);
else /* Both points inside - just pass */
    /* this endpoint (topofstack) on to next stage */

if (stagept->status != ST_CLOSED) {
    /* Update last point in stage */
    OBJ_COPY(*endp, stagept->prevp);
    stagept->prevdistance = distance;
    stagept->status = endstatus;
}

else if (sp->point != NIL) {
    /* No more clipping stages - point ready to output */
    CS_OUTPUT(sp->point);
    CS_POP(sp);
}
else /* Closing point passed thru all stages */
    break;

} /* End of while not stack empty */

return no;
APPENDIX D. LIANG-BARSKY ALGORITHM

/*
 * barsky.c - Two-dimensional line clipping algorithm after Liang and
 * Barssy.
 * 
 * Translated from Pascal to C by Morten Zachrisen, RUNIT-F
 * 
 * Date: 1989 02 21 Copyright (c) 1985 Morten Zachrisen
 */

#define INFINITY 1.0e+10

/*@ Clipping status */
#define CLIP_OUTSIDE -1
#define CLIP_NOSTAT 0
#define CLIP_INSIDE 1

struct vertex {
    float x, y;
};

/*@ Window boundaries */
static float clip_xleft;
static float clip_xright;
static float clip_ybottom;
static float clip_ytop;

#define appendvertex(xx, yy) (outp[no].x = (xx), \ 
    outp[no].y = (yy), no++)

/*@ Clip input polygon and returns no. of output vertices */
int b_polyClip(inp, outp, n)
    struct vertex *inp; /* Input polygon */
    struct vertex *outp; /* Output polygon */
    int n; /* No of input vertices */
{
    float deltax, deltay; /* Line segment components */
    float xin, yin; /* Enter window coordinate */
    float xout, yout; /* Leave window coordinate */
    float tinx, tincy; /* Parameter at enter window */
    float toutx, touty; /* Parameter at leave window */
    float tin1, tin2; /* Parameter at first and second entry */
    float tout1, tout2; /* Parameter at first and second exit */

    register struct vertex *lastpt; /* Startpoint */
    register struct vertex *nextpt; /* Endpoint */
    register int i; /* Input vertex counter */
    register int no; /* Output vertex counter */

    no = 0;

    for (lastpt = inp, nextpt = lastpt + 1, i = 0; i < n; i++, lastpt = nextpt, nextpt++) {
        if (i == n - 1)
            nextpt = inp; /* Reuse first point */

        /* Edge v[i] to v[i+1] */
        deltax = nextpt->x - lastpt->x;

        

if (deltax > 0.0) {
    /* Edge points to left */
    xin = clip_xleft;
    xout = clip_xright;
} else {
    /* Edge points to right */
    xin = clip_xright;
    xout = clip_xleft;
}

deltay = nextpt->y - lastpt->y;
if (deltay > 0.0) {
    /* Edge points up */
    yin = clip_ybottom;
    yout = clip_ytop;
} else {
    /* Edge points down */
    yin = clip_ytop;
    yout = clip_ybottom;
}

if (deltax != 0.0)
    tinx = (xin - lastpt->x) / deltax;
else /* Vertical */
    tinx = -INFINITY;

if (deltay != 0.0)
    tiny = (yin - lastpt->y) / deltax;
else /* Horizontal */
    tiny = -INFINITY;

if (tinx < tiny) {
    /* First entry at x then y */
    tin1 = tinx;
    tin2 = tiny;
} else {
    /* First entry at y then x */
    tin1 = tiny;
    tin2 = tinx;
}

if (tin1 <= 1.0) {
    /* Case 2 or 3 or 4 or 6 */
    if (0.0 < tin1) {
        /* Case 5 - turning vertex */
        appendvertex(xin, yin);
    }
}

if (tin2 <= 1.0) {
    /* Case 3 or 4 or 6 */
    if (deltax != 0.0)
        toutx = (xout - lastpt->x) / deltax;
    else /* Vertical */
        if (clip_xleft <= lastpt->x &&
            lastpt->x <= clip_xright)
            toutx = INFINITY; /* Inside */
        else
            toutx = -INFINITY;
}

if (deltay != 0.0)
    touty = (yout - lastpt->y) / deltax;
else /* Horizontal */
    if (clip_ybottom <= lastpt->y &&
        lastpt->y <= clip_ytop)
/* Inside */
touty = INFINITY;
else    /* Outside */
touty = -INFINITY;
}

if (toutx < touty)    /* First exit at x */
tout1 = toutx;
else    /* First exit at y */
tout1 = touty;

if (0.0 < tin2 || 0.0 < tout1) {
    /* Case 4 or 6 */
    if (tin2 <= tout1) {
        /* Case 4 - visible segment */
        if (0.0 < tin2) {
            /* v[i] outside window */
            if (tiny < tinx)    /* Vertical boundary */
                appendvertex(xin, lastpt->y + tinx * deltay);
            else    /* Horizontal boundary */
                appendvertex(lastpt->x + tiny * delta yin);
        }
        if (tout1 < 1.0) {
            /* v[i+1] outside window */
            if (toutx < touty)
                appendvertex(xout, lastpt->y + toutx * deltay);
            /* Vertical boundary */
            else
                appendvertex(lastpt->x + touty * deltay, yout);
            /* Horizontal boundary */
        } else /* v[i+1] inside window */
            appendvertex(nextpt->x, nextpt->y);
    }    /* of Case 4 */
else {
    /* Case 6 - turning vertex */
    if (tiny < tinx)    /* Second entry at x */
        appendvertex(xin, yout);
    else    /* Second entry at y */
        appendvertex(xout, yin);
}

}    /* Case 4 or 6 */
}    /* Case 3 or 4 or 6 */
}    /* Case 2 or 3 or 4 or 6 */
}    /* Edge processing */
return no;    /* No. of output vertices */
APPENDIX E. SUTHERLAND-HODGMAN ALGORITHM

NOTE: In order to reduce this Appendix from 18 to 10 pages, the code for clipping and closing the x-right, y-bottom, and y-top stages, has been left out. This is completely symmetrical to the clipping and closing of the x-left stage.

The code has been slightly changed to use a right-handed coordinate system instead of a left-handed (#ifdef lefthanded).

(Due to the wide lines, the algorithm unfortunately has to be printed in 9 point Courier.)

/*
 * mipolyclip.c - Code to do reentrant polyline clipping
 *
 * Copyright 1988
 * Center for Information Technology Integration (CITI)
 * Information Technology Division
 * University of Michigan
 * Ann Arbor, Michigan
 *
 * All Rights Reserved
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee is hereby
 * granted, provided that the above copyright notice appear in all
 * copies and that both that copyright notice and this permission
 * notice appear in supporting documentation, and that the names of
 * CITI or THE UNIVERSITY OF MICHIGAN not be used in advertising or
 * publicity pertaining to distribution of the software without
 * specific, written prior permission.
 *
 * THE SOFTWARE IS PROVIDED "AS IS." CITI AND THE UNIVERSITY OF
 * MICHIGAN DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
 * NO EVENT SHALL CITI OR THE UNIVERSITY OF MICHIGAN BE LIABLE FOR ANY
 * SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
 * AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING
 * OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
 * SOFTWARE.
 */
#include <stdio.h>
define FLOAT float
define B32

typedef struct { 
  FLOAT x B32;
  FLOAT y B32;
  FLOAT z B32;
  FLOAT w B32;
  long color; /* To make compatible with moviebox */
} pexCoord4D;

/*********************************************************************************/
typedef struct Polyclip_Cb_Type
{
pexCoord4D first;        /* Save for first vertex processed. */
int first_vertex;        /* Indicates whether the vertex being */
                        /* processed is first. */
int output_occurred;     /* Indicator that a vertex was outputted. */
float plane;             /* Value for plane being clipped to. */
pexCoord4D previous;     /* Save for previous vertex processed. */
} pexPolyclip_Cb;

#define CARD32 int

/* Global storage associated with polygon clipping routines. */
static int out_count;    /* Storage for number of vertices clipped. */
static pexCoord4D *out_ptr; /* Next clipped vertex to store. */
static int vertices_left; /* Storage for vertices left in out_vertices. */

/*
planes0-5 = six values for clipping planes (xl, xr, yb, yt, zh, zy)
plane_cbs = global storage for planes to clip to and shit.
*/
static pexPolyclip_Cb plane_cbs[6] =
{
    {0.0,0.0,0.0},1.0,-1.0,{0.0,0.0,0.0},       /* x left (min) */
    {0.0,0.0,0.0},1.0,1.0,{0.0,0.0,0.0},        /* x right (max) */
    {0.0,0.0,0.0},1.0,-1.0,{0.0,0.0,0.0},       /* y bottom (min) */
    {0.0,0.0,0.0},1.0,1.0,{0.0,0.0,0.0},        /* y top (max) */
    {0.0,0.0,0.0},1.0,0.0,{0.0,0.0,0.0},        /* z hither (min) */
    #ifdef lefthanded
    {0.0,0.0,0.0},1.0,0.0,{0.0,0.0,0.0},        /* z yon (max) */
    #else
    {0.0,0.0,0.0},1.0,-1.0,{0.0,0.0,0.0},       /* z yon (min in rhc) */
    #endif
};

/****************************************************************************

| "pexPolyclip_out"
| Purpose: Store away a clipped vertex.
| Parameters:
| cb -> clipping plane information
| current -> polygon vertex to be stored
| Globals:
| out_count = count of vertices placed in out_vertices
| out_ptr = cursor to run across out_vertices when storing
| vertices
| vertices_left = global storage for down counter for max_out_count

****************************************************************************/

pexPolyclip_out(cb, current)
pexPolyclip_Cb *cb;
pexCoord4D *current;
{
extern int vertices_left; /* Storage for vertices left in out_vertices. */

if (--vertices_left >= 0) {
    out_ptr->x = current->x;
    out_ptr->y = current->y;
    out_ptr->z = current->z;
    out_ptr->w = current->w;
    out_ptr++;
    out_count++;
}

/******************************************************************************
 * | "pexpolyclip_xl"
 * | Purpose:
 * |   Clip a polygon vertex to the left x clipping plane.
 * | Parameters:
 * |   cb  -> clipping plane information for left % clipping plane
 * |   current  -> polygon vertex to be processed
 /******************************************************************************/

pexpolyclip_xl(cb, current)
    pexPolyclip_Cb *cb;
    pexCoord4D *current;
{
    FLOAT alpha;        /* Ratio of line on each */
    /* side of clipping plane. */
    FLOAT delta;        /* Distance in % direction between */
    /* current point and clip plane. */
    pexCoord4D intersection; /* Storage for point which is */
    /* intersection of edge and plane. */

    /* Processing first point. Save its value in previous and first storage */
    /* locations of cb. Clear first point flag. */
    if (cb->first_vertex)
    {
        cb->first.x = current->x;
        cb->previous.x = current->x;
        cb->first.y = current->y;
        cb->previous.y = current->y;
        cb->first.z = current->z;
        cb->previous.z = current->z;
        cb->first.w = current->w;
        cb->previous.w = current->w;
        cb->first_vertex = 0;
    }

    /* This is not the first point of the polygon, and the edge from */
    /* previous to current does not cross the clipping boundary. So just */
    /* save the current point as the \ previous point. */
    else if ( (current->x - (current->w * cb->plane)) *
       (cb->previous.x - (cb->previous.w * cb->plane)) )
E4 Volume 3 - Fast Algorithms for Polygon Clipping and Boxing

```c

>= 0.0 )
{
    cb->previous.x = current->x;
    cb->previous.y = current->y;
    cb->previous.z = current->z;
    cb->previous.w = current->w;
}

/ *--------------------------------------------------------*/
| This is not the first point, and the edge from current to previous |
| crosses the clipping plane. So calculate the intersection point |
| between the edge and the plane and output it to the next clipping |
| stage. Then set the previous vertex to the values of the current |
| vertex. |
|---------------------------------------------------------------------*/
else
{
    delta = current->x - (current->w * cb->plane);
    alpha = delta / ( delta - (cb->previous.w - cb->plane));
    intersection.x = current->x + alpha*(cb->previous.x - current->x);
    intersection.y = current->y + alpha*(cb->previous.y - current->y);
    intersection.z = current->z + alpha*(cb->previous.z - current->z);
    intersection.w = current->w + alpha*(cb->previous.w - current->w);

cb->output_occurred = 1;
    pexpolyclip_xr(cb+1, &intersection);

cb->previous.x = current->x;
    cb->previous.y = current->y;
    cb->previous.z = current->z;
    cb->previous.w = current->w;
}

/ *--------------------------------------------------------*/
| Test to see if we should output the vertex now stored in previous. If |
| it is visible we should send it to the next clipping stage. |
|---------------------------------------------------------------------*/

if ( cb->previous.x >= (cb->previous.w * cb->plane) )
{
    cb->output_occurred = 1;
    pexpolyclip_xr(cb+1, &cb->previous);
}

return 0;
}

*******************************************************************************
| * "pexpolyclip_xr" |
| "pexpolyclip_yb" |
| "pexpolyclip_zc" |
| * These have been omitted from this listing (similar to pexpolyclip_xl) |
| * | |
*******************************************************************************

*******************************************************************************
| * "pexpolyclip_zh" |
| * | |
| Purpose: |
| Clip a polygon vertex to the z hither clipping plane. |
| * | |
| Parameters: |
| * cb -> clipping plane information for z hither clipping plane |
| * current -> polygon vertex to be processed |
| * |
*******************************************************************************
```
pexpolyclip_zh(cb, current)
pexpolyclip_CB *cb;
pexCoord4D *current;
{
    FLOAT alpha; /* Ratio of line on each */
    /* side of clipping plane. */
    FLOAT delta; /* Distance in y direction between */
    /* current point and clip plane. */
    pexCoord4D intersection; /* Storage for point which is */
    /* intersection of edge and plane. */

    /* Processing first point. Save its value in previous and first storage */
    /* locations of cb. Clear first point flag. */
    
    if (cb->first_vertex)
    {
        cb->first.x = current->x;
        cb->first.x = current->x;
        cb->first.y = current->y;
        cb->previous.x = current->y;
        cb->first.y = current->y;
        cb->previous.x = current->z;
        cb->previous.z = current->z;
        cb->first.w = current->w;
        cb->previous.w = current->w;
        cb->first_vertex = 0;
    }
    
    /* This is not the first point of the polygon, and the edge from */
    /* previous to current does not cross the clipping boundary. So just */
    /* save the current point as the \ previous point. */
    
    #ifdef left-handed
    else if (( (current->z - (current->w * cb->plane)) *
                (cb->previous.z - (cb->previous.w * cb->plane)))
               >= 0.0 )
    #else
    else if (( (current->z - (current->w * cb->plane)) *
                (cb->previous.z - (cb->previous.w * cb->plane)))
               <= 0.0 )
    #endif
    {
        cb->previous.x = current->x;
        cb->previous.y = current->y;
        cb->previous.z = current->z;
        cb->previous.w = current->w;
    }
    
    /* This is not the first point, and the edge from current to previous */
    /* crosses the clipping plane. So calculate the intersection point */
    /* between the edge and the plane and output it to the next clipping */
    /* stage. Then set the previous vertex to the values of the current */
    /* vertex. */
    
    else
    {
        delta = current->z - (current->w * cb->plane);
        alpha = delta / ( delta - (cb->previous.z -
                             (cb->previous.w * cb->plane)));
    }
intersection.x = current->x + alpha*(cb->previous.x - current->x);
intersection.y = current->y + alpha*(cb->previous.y - current->y);
intersection.z = current->z + alpha*(cb->previous.z - current->z);
intersection.w = current->w + alpha*(cb->previous.w - current->w);

cb->output_occurred = 1;
pexpolyclip_zy(cb+1, &intersection);

cb->previous.x = current->x;
cb->previous.y = current->y;
cb->previous.z = current->z;
cb->previous.w = current->w;

} /* Test to see if we should output the vertex now stored in previous. If it is visible we should send it to the next clipping stage. */

 ifdef lefthanded
 if ( cb->previous.z >= (cb->previous.w * cb->plane) )
 else
 if ( cb->previous.z <= (cb->previous.w * cb->plane) )
 endif
 {
 cb->output_occurred = 1;
pexpolyclip_zy(cb+1, &cb->previous);
}

 return 0;

} /**************************************************************************/

/**************************************************************************/
/* pexpolyclip_zy */
/* Purpose:
 * Clip a polygon vertex to the yon z clipping plane.
 * Parameters:
 * cb -> clipping plane information for left % clipping plane
 * current -> polygon vertex to be processed
 */
******************************************************************************/

pexpolyclip_zy(cb, current)
pexpolyclip_Cb *cb;
pexCoord4D *current;
{
 FLOAT alpha; /* Ratio of line on each */
 /* side of clipping plane. */
 FLOAT delta; /* Distance in y direction between */
 /* current point and clip plane. */
 pexCoord4D intersection; /* Storage for point which is */
 /* intersection of edge and plane. */

/**************************************************************************/
/* Processing first point. Save its value in previous and first storage */
/* locations of cb. Clear first point flag. */
******************************************************************************/

if (cb->first_vertex)
{
 cb->first.x = current->x;
 cb->previous.x = current->x;
 cb->first.y = current->y;
 cb->previous.y = current->y;
 cb->first.z = current->z;

cb->previous.z = current->z;
cb->first.w = current->w;
cb->previous.w = current->w;

cb->first_vertex = 0;
}

/*---------------------------------------------------------------------------------------------*/
/* This is not the first point of the polygon, and the edge from */
/* previous to current does not cross the clipping boundary. So just */
/* save the current point as the "previous point. */
/*---------------------------------------------------------------------------------------------*/

else if (( (current->z - (current->w * cb->plane)) *  
            (cb->previous.z - (cb->previous.w * cb->plane)) )  
            >= 0.0 )
{
    cb->previous.x = current->x;
    cb->previous.y = current->y;
    cb->previous.z = current->z;
    cb->previous.w = current->w;
}

/*---------------------------------------------------------------------------------------------*/
/* This is not the first point, and the edge from current to previous */
/* crosses the clipping plane. So calculate the intersection point */
/* between the edge and the plane and output it to the next clipping */
/* stage. Then set the previous vertex to the values of the current */
/* vertex. */
/*---------------------------------------------------------------------------------------------*/

else
{
    delta = current->z - (current->w * cb->plane);
    alpha = delta / ( delta - (cb->previous.z -  
                        (cb->previous.w * cb->plane)) ) ;
    intersection.x = current->x + alpha*(cb->previous.x - current->x);
    intersection.y = current->y + alpha*(cb->previous.y - current->y);
    intersection.z = current->z + alpha*(cb->previous.z - current->z);
    intersection.w = current->w + alpha*(cb->previous.w - current->w);

    cb->output_occurred = 1;
    pexpolyclip_out(cb+1, &intersection);

    cb->previous.x = current->x;
    cb->previous.y = current->y;
    cb->previous.z = current->z;
    cb->previous.w = current->w;
}

/*---------------------------------------------------------------------------------------------*/
/* Test to see if we should output the vertex now stored in previous. If */
/* it is visible we should send it to the next clipping stage. */
/*---------------------------------------------------------------------------------------------*/

#elifdef lefthanded
    if ( cb->previous.z <= (cb->previous.w * cb->plane) )
#else
    if ( cb->previous.z >= (cb->previous.w * cb->plane) )
#endif
{
    cb->output_occurred = 1;
    pexpolyclip_out(cb+1, &cb->previous);
}

return 0;
}
pexpolyclose_xl(cb)
pexpolyclip_Cb
{
    FLOAT alpha; /* Ratio of line on each side of */
    /* clipping plane. */
    FLOAT delta; /* Distance in % direction between */
    /* current point and clip plane. */
    pexCoord4D intersection; /* Storage for point which is */
    /* intersection of edge and plane. */

    /***********************************************************
    | Output occurred. So close the polygon back to its first vertex. If |
    | the edge from the last vertex to the first vertex crosses the |
    | clipping plane, generate the intersection point and output it. |
    ********************************************************/

    if (cb->output_occurred)
    {
        if ( ( (cb->previous.x - (cb->previous.w * cb->plane)) *
            (cb->first.x - (cb->first.w * cb->plane)) )
            < 0.0)
        {
            delta = cb->first.x - (cb->first.w * cb->plane);
            alpha = delta / ( delta - (cb->previous.x -
                (cb->previous.w * cb->plane)) );
            intersection.x = cb->first.x + alpha*(cb->previous.x -
                cb->first.x);
            intersection.y = cb->first.y + alpha*(cb->previous.y -
                cb->first.y);
            intersection.z = cb->first.z + alpha*(cb->previous.z -
                cb->first.z);
            intersection.w = cb->first.w + alpha*(cb->previous.w -
                cb->first.w);

            pexpolyclip_xr(cb+1, &intersection);
        }
    }

    /***********************************************************
    | Reset flags and close off next stage. |
    ********************************************************/

    cb->output_occurred = 0;
    cb->first_vertex = 1;
    pexpolyclose_xr(cb+1);

    return 0;
}

******************************************************************************

| "pexpolyclose_xr"
| "pexpolyclose_yb"
| "pexpolyclose_yt"
| These have been omitted from this listing (similar to pexpolyclose_xl)
******************************************************************************

******************************************************************************
pexpolyclose_zh(cb)
pexPolyclip_Cb      *cb;
{
  FLOAT       alpha;   /* Ratio of line on each */
               /* side of clipping plane. */
  FLOAT       delta;   /* Distance in % direction between */
               /* current point and clip plane. */
  pexCoord4D   intersection;  /* Storage for point which */
               /* is intersection of edge and plane. */

  /* Output occurred. So close the polygon back to its first vertex. If */
  /* the edge from the last vertex to the first vertex crosses the */
  /* clipping plane, generate the intersection point and output it. */

  if (cb->output_occurred)
  {
    if ( ((cb->previous.z - (cb->previous.w * cb->plane)) *
          (cb->first.z - (cb->first.w * cb->plane))) < 0.0 )
    {
      delta = cb->first.z - (cb->first.w * cb->plane);
      alpha = delta / (delta - (cb->previous.z -
                         (cb->previous.w * cb->plane)));
      intersection.x = cb->first.x + alpha*(cb->previous.x - cb->first.x);
      intersection.y = cb->first.y + alpha*(cb->previous.y - cb->first.y);
      intersection.z = cb->first.z + alpha*(cb->previous.z - cb->first.z);
      intersection.w = cb->first.w + alpha*(cb->previous.w - cb->first.w);
      pexpolyclip_zy(cb+1, &intersection);
    }
  }

  /* Reset flags and close off next stage. */
  cb->output_occurred = 0;
  cb->first_vertex = 1;
  pexpolyclose_zh(cb+1);
  return 0;
}

pexpolyclose_zy(cb)
E10 Volume 3 - Fast Algorithms for Polygon Clipping and Boxing

```c
pexPolyclip_Cb *cb;
{
  FLOAT alpha; /* Ratio of line on each side */
  /* of clipping plane. */
  FLOAT delta; /* Distance in % direction between */
  /* current point and clip plane. */
  pexCoord4D intersection; /* Storage for point which is */
  /* intersection of edge and plane. */

  /************************************************************************/**/*.|
  | Output occurred. So close the polygon back to its first vertex. If |
  | the edge from the last vertex to the first vertex crosses the    |
  | clipping plane, generate the intersection point and output it.  |
  +----------------------------------------------------------------------*/

  if (cb->output_occurred)
  {
    if (((cb->previous.z - (cb->previous.w * cb->plane)) *
        (cb->first.z - (cb->first.w * cb->plane))) < 0)
    {
      delta = cb->first.z - (cb->first.w * cb->plane);
      alpha = delta / ((delta - (cb->previous.z -
        (cb->previous.w * cb->plane))));
      intersection.x = cb->first.x + alpha*(cb->previous.x - cb->first.x);
      intersection.y = cb->first.y + alpha*(cb->previous.y - cb->first.y);
      intersection.z = cb->first.z + alpha*(cb->previous.z - cb->first.z);
      intersection.w = cb->first.w + alpha*(cb->previous.w - cb->first.w);

      pexpolyclip_out(cb+1, intersection);
    }
  }

  /************************************************************************/**/*.|  |
  | Reset flags.                                                        |
  +----------------------------------------------------------------------*/

  cb->output_occurred = 0;
  cb->first_vertex = 1;

  return 0;
}
```

//***************************
// "PexClipPolygon"
// Purpose:
// Clip a polygon against a viewport and return resultant polygon
// Parameters:
// in_count - number of vertices in polygon to be clipped
// in_vertices -> array of vertices for polygon to be clipped
// max_out_count = maximum number of vertices in out_vertices
// out_vertices -> array to store clipped vertices in
// Globals:
// out_count = count of vertices placed in out_vertices
// out_ptr = runs across out_vertices when storing vertices
// plane_cbs = for planes to clip to(by pexpolyclip_init call)
// vertices_left = for down counter for max_out_count
// Function value:
// if > 0 it indicates number of vertices in output polygon
// if = 0 polygon was clipped
// if < 0 then clipped polygon has more than max_out_count vertices
// */
CARD32
PexClipPolygon(in_count, in_vertices, max_out_count, out_vertices)
    int in_count;
    pexCoord4D *in_vertices;
    int max_out_count;
    pexCoord4D *out_vertices;
{
    pexCoord4D *current;  /* Cursor for processing in_vertices. */
    extern int out_count;  /* Storage for number of vrts clipped. */
    extern pexCoord4D *out_ptr;  /* Next clipped vertex to store. */
    extern pexPolyclip_Cb plane_cbs[6];  /* Storage for planes to clip to. */
    extern int vertices_left;  /* Storage for vertices left out_vert. */
    static int initialized = 0;

    /*---------------------------------------------------------------*/
    /* Initialize global storage locations associated with our parameters. */
    /* Then set indices for processing in_vertices. */
    /*---------------------------------------------------------------*/

    vertices_left = max_out_count;
    out_count = 0;
    out_ptr = out_vertices;
    current = in_vertices;

    /*---------------------------------------------------------------*/
    /* Loop through input array clipping the vertices based on the clipping */
    /* planes in "planes". */
    /*---------------------------------------------------------------*/

    while (-=-in_count >= 0)
    {
        pexPolyclip xl(plane_cbs, current);
        current++;
    }

    /*---------------------------------------------------------------*/
    /* If we exceeded the size of out_vertices, send back a -1 indication. */
    /* Otherwise return the number of vertices processed. */
    /*---------------------------------------------------------------*/

    if (vertices_left < 0)
    {
        fprintf(stderr,
                "Exceeded out_vertices size in the Pex polygon clipping code.\n");
        return -out_count;
    }

    pexPolyclip xl(plane_cbs);
    return out_count;
}
E12 Volume 3 - Fast Algorithms for Polygon Clipping and Boxing
APPENDIX F. REFERENCES


SUTHERLAND74  Sutherland, I. E., G.W. Hodgman "Reentrant Polygon Clipping", Communications of ACM, January 1974.  


Computer Graphics Rendering Techniques
with an Emphasis on Performance Issues

Volume 4

Adding Structure to Bit-map Displays
ABSTRACT OF VOLUME 4

In colour raster graphics programs, overlays and special look-up tables are very useful for implementing interactive techniques. However, without a structured approach to the use of the look-up table, the problem can easily get out of hand. A simple, recursive algorithm is described which avoids difficulties by dividing the frame buffer into logical layers.
PREFACE TO VOLUME 4

The following is a re-writing of a paper that appeared in Computer Graphics and Applications, July 1984. The text and programs have been slightly modified, and an afterword has been appended. Some of the figures have been directly copied from the paper.
TABLE OF CONTENTS

ABSTRACT OF VOLUME 4 ..............................................I
PREFACE TO VOLUME 4 .............................................II
TABLE OF CONTENTS ..................................................III
1. Introduction ..........................................................1
2. Hardware Interface ...............................................1
3. A Workstation with Structured Raster Display ..............2
4. A Recursive Algorithm for Layering the Video Look-Up Table .4
5. Layers, Display Managers and GKS ..........................7
6. Conclusions .........................................................8
7. Afterwords, 1989 ....................................................8
APPENDIX A. REFERENCES ...........................................A1
1. Introduction

Designers of interactive programs for a colour raster display frequently find it is convenient to divide the graphic image into "layers", each of which presents an independent set of data. The layers are often spatially correlated, and are displayed together. This separation and restructuring process can be viewed as a computer graphics analogy to the display of multiple transparencies with an overhead projector.

This layered structure is typically used in VLSI layout programs [TANIMOTO80], [HEGEBÖ83]; it is also found in systems that integrate computer graphics with video images. Further, for most types of interactive CAD programs, geographic information systems, and image analysis programs, designers would also like to put entities like grids, cursors, and pop-up menus in overlay planes.

![Diagram of Bit-Plane Selection Register and Video Look-Up Table]

Fig. 1.
A bit-plane selection register and a video look-up table set up to divide a three-plane frame buffer into two logical layers. One layer (plane 0) is overlaid on other layers (planes 1 and 2).

2. Hardware Interface

Available frame buffer bit-planes are typically allocated to the desired layers. On graphic output, a bit-plane selection register selects the appropriate layer.
By means of "clever" video look-up table programming, layers can be visualized and given front-to-back priorities and transparency attributes [FOLEY82]. In our experience, however, this causes the implementors of application programs a lot of trouble. Utilities for managing layers should therefore be part of the basic graphics software.

Figure 1 shows a diagram of the central components necessary to divide a frame buffer into two layers. The indicated setting of the video look-up table will overlay one layer (red colour) on a two-bit (four gray levels) image.

3. **A Workstation with Structured Raster Display**

The ICAN Kernel Raster Workstation [ICAN83] is equipped with a frame buffer with 16 planes of 1024 x 1024 pixels, a special display file, and an optional transformation unit. It handles retained, transformable graphics (polygons and lines) together with unretained graphics (images, raster-op graphics).

During the development of basic graphics software for the ICAN Kernel Raster Workstation family, designers decided to include the layer concept [ZACHRISEN84b, SOLEM84]. Layers conveniently separate continuously refreshed, dynamic drawings from static information. Moreover, introduction of a one-to-one double-layer relationship on layer entities provides a simple software configuration of a double-buffered raster memory. The resulting graphics package contains several other concepts (viewports, for example) and relations as shown in Figure 2.
Fig. 2.
Data model diagram for the Tiger-R graphics package. Notation is derived from ER-diagrams [CHEN76]. Rectangles represent entities, ovals represent associated attributes, and connecting lines with diamonds and cardinality figures represent relationships.
4. A Recursive Algorithm for Layering the Video Look-Up Table

During the design of this software (called Tiger-R), a simple algorithm for setting the video look-up table was developed. The algorithm, described in the C programming language in Figure 3, is based on a record type, "layer", which represents the layer entity. Layers are linked together in a singly linked list ordered by decreasing visual priority. The first layer in the list will be shown on top of any other layer.

```c
typedef unsigned char Utiny;    /* Type for unsigned byte data */
typedef int Colindex;           /* Type for colour data */

/* Layer entity: */
struct layer {
    struct layer *lay_next;    /* Link to layer behind */
    float lay_visib;           /* Visibility (0.0 to 1.0) */
    Colindex lay_transp;       /* Transparent colour index */
    Utiny lay_offset;          /* Offset of lowest plane in layer */
    Utiny lay_planes;          /* Number of planes in layer */
    struct rgb *lay_col;       /* Private colour table for layer */
};

/* Video look-up table entry: */
struct rgb {
    Utiny red;                 /* Red colour component (0 to 255) */
    Utiny green;               /* Green ditto */
    Utiny blue;                /* Blue ditto */
};

extern struct rgb Background;  /* Background colour */
extern struct rgb Vlt[1 << PLANES]; /* Global video look-up table */
extern struct layer *Frontlay;  /* Pointer to front layer */
```

Fig. 3.
The data structure declarations.

Attributes consist of

1. two numbers indicating which frame buffer planes are allocated to "layer" and
2. a pointer to the layer-specific (private) colour table segment.
As the sum of private colour table entries will never exceed the number of entries in the look-up table, space for the entries can be allocated from a fixed-size table.

The appearance may be controlled by a visibility (weight) attribute. At a visibility value of 1.0, the top layer is fully oblique. As the visibility attribute is decreased, the top layer fades, and the layer underneath shows until the top layer reaches a transparent state.

![Diagram](image)

**Fig. 4.**
Data structure for the example in Figure 1 with the overlay plane made translucent (visibility = 0.7).

It is often desirable to define a certain colour index as transparent. This step is analogous to the mixing register in video mixers. Thus, whatever the layer visibility attribute state, the layer is transparent where the frame buffer contains this colour value. As can be seen in Figures 1 and 4, the 0-values are transparent in the otherwise red layer.

The algorithm traverses the layer list from front to back. Each layer updates the global video look-up table according to its visibility and its private colour table and recursively invokes the algorithm for colouring the background. The recursive procedure for computing the video look-up table is shown in Figure 5. It is invoked by $\text{CompCol}(0, \text{Frontlay}, 1.0)$;
/* Recursive procedure for dividing frame buffer into layered structure, by means of setting colour look-up table. */
CompCol(colind, laypt, visib)
Colindex colind; /* Video look-up table index */
struct layer *laypt; /* Pointer to current layer */
float visib; /* Weight factor for this layer and background */
{
  Colindex colour; /* Index for private colour */
  if (laypt == Frontlay) /* Front layer - initialize look-up table */
    Vlt[0].red = Vlt[0].green = Vlt[0].blue = 0;
  if (laypt != NULL) { /* Normal layer - not background */
    /* Loop backwards through private colour table */
    for (colour = 1 << laypt->lay_planes; --colour >= 0; ) {
      if (laypt->lay_visib > 0.0 &&
          colour != laypt->lay_transp) {
        Vlt[colind + (colour << laypt->lay_offset)].red =
          Vlt[colind].red + (Utiny)(visib * laypt->lay_visib *
            laypt->lay_col[colour].red);
        .
        .
        /* Colour background */
        CompCol(colind + (colour << laypt->lay_offset),
            laypt->lay_next, visib * (1.0 - laypt->lay_visib));
      } else {
        Vlt[colind + (colour << laypt->lay_offset)].red =
          Vlt[colind].red;
        .
        .
        /* Colour background */
        CompCol(colind + (colour << laypt->lay_offset),
            laypt->lay_next, visib);
      }
    }
  } else { /* Background reached - end recursion */
    Vlt[colind].red += (Utiny)(visib * Background.red);
    .
    .
  }
}

Fig. 5.
The recursive colour assignment procedure.

Colour components are added to the video look-up tables as the list is traversed. Each layer updates video look-up table entries according to its
allocated planes, unaware of which planes are allocated to layers behind it. Video look-up table entries spread to the whole table as the last layer is visited. (See Figure 6.)

\[ vlt[0] = \{0, 0, 0\} \]

\[
\begin{align*}
vlt[0 + 1] &= vlt[0] + 0.7 \cdot \text{lay}_1 \rightarrow \text{lay}_\text{col}[1] = \{178, 0, 0\} \\
vlt[1 + 6] &= vlt[1] + 0.3 \cdot \text{lay}_2 \rightarrow \text{lay}_\text{col}[3] = \{254, 76, 76\} \\
vlt[1 + 4] &= vlt[1] + 0.3 \cdot \text{lay}_2 \rightarrow \text{lay}_\text{col}[2] = \{229, 51, 51\} \\
vlt[1 + 2] &= vlt[1] + 0.3 \cdot \text{lay}_2 \rightarrow \text{lay}_\text{col}[1] = \{203, 25, 25\} \\
vlt[1 + 0] &= vlt[1] + 0.3 \cdot \text{lay}_2 \rightarrow \text{lay}_\text{col}[0] = \{178, 0, 0\}
\end{align*}
\]

\[ vlt[0 + 0] = vlt[0] = \{0, 0, 0\} \]

\[
\begin{align*}
vlt[0 + 6] &= vlt[0] + \text{lay}_2 \rightarrow \text{lay}_\text{col}[3] = \{255, 255, 255\} \\
vlt[0 + 4] &= vlt[0] + \text{lay}_2 \rightarrow \text{lay}_\text{col}[2] = \{170, 170, 170\} \\
vlt[0 + 2] &= vlt[0] + \text{lay}_2 \rightarrow \text{lay}_\text{col}[1] = \{85, 85, 85\} \\
vlt[0 + 0] &= vlt[0] + \text{lay}_2 \rightarrow \text{lay}_\text{col}[0] = \{0, 0, 0\}
\end{align*}
\]

Fig. 6.

Updates to the video look-up table as the recursive algorithm is applied to the data structure in Figure 4. Because the back layer is completely oblique, recursion to background is not shown.

The displayed algorithm linearly interpolates the colour components between the layer and the background according to the layer's visibility. Within the framework of the algorithm, other colour-setting strategies can also be implemented (additive and subtractive colour mixture, etc.).

5. Layers, Display Managers and GKS

The Graphical Kernel System [ENDERLE84] is of large and growing importance in the graphics community. As a result, the possible relationships between a layer facility and the GKS need to be discussed.

Graphics systems, including GKS, prefer to view a raster display device as a uniform frame buffer surface that can be divided into viewports at the programmer's will. When a graphics system is run under an operating system with an integrated display manager, the two display structures will conflict. The most complete solution to this conflict would be to integrate the concepts of a display manager's window with the viewport of the graphics package. Currently, however, the semantics of the two display structures are different. Normally, this conflict is resolved either by mapping the entire display screen
of the graphics package into the window handed over from the display manager, or by suspending the display manager and borrowing the complete display surface while the graphics program is active.

The layer concept also conflicts with the philosophy of GKS. Programs using GKS cannot divide the frame buffer into layers because there is no mechanism that selects bit planes for output. Neither is there any way of specifying how the different logical surfaces overlay each other. The former could, however, be implemented as an "escape" function, but would obstruct the portability of GKS programs. It would perhaps be better to integrate the layer concept with a display manager, allowing a layer specification to be handed over to a graphics program together with the window area. Moreover, overlapping display windows can be implemented very efficiently in a display manager by means of layers.

6. Conclusions

There are still unsolved problems concerning the integration of standard graphics systems with layers and display managers' windows. At RUNIT (the computing centre at the University of Trondheim) solutions to these problems are being pursued in two projects: an implementation of an extended version of GKS and a joint project with ICAN (Interactive Computer Aids of Norway) for developing a display management system for the ICAN Kernel Raster Workstation. Internationally, much related research is going on in the field of user interface management [ROSENTHAL83, ACQUAH82]. An eventual unified model of the relationship between layers, windows, and standard graphics systems would strongly encourage the production of portable, user-friendly graphics applications software.

7. Afterwords, 1989

Since the article was originally written in 1984, workstation technology has made large progress, both in terms of hardware and software. We see that windowing systems are having a big break-through, and we see emerging industry standards for windowing systems, i.e. the X Window System [SCHEIFLER86].
Though GKS is being interfaced with windowing systems, for instance by means of an X device driver, GKS has not been integrated with windowing concepts. Another candidate for graphic system standard, PHIGS, is currently being integrated with the X window system. The new system is being called PEX [PEX88].

For simple two-dimensional applications, X or X Toolkit may be sufficient and efficient as a programming basis. For applications with more advanced graphic requirements, PEX may prove to be the choice. Though being the only officially adopted graphic standard, the alliance of a windowing system with PHIGS may leave GKS out in the cold.

The layering approach has not been adopted by neither GKS nor X. In X it is possible to have some control over the allocation of frame buffer planes, but no structured approach as the one described here is available. Implementing colour layers is still the responsibility of the application program, which means that the described algorithm is still of importance.
APPENDIX A. REFERENCES.


ICAN83 "ICAN Kernel Raster Workstation," ICAN a/s, Moloveien 1, PO Box 31, N-3191 Horten.


