MODELING OF COMPLEX THREE DIMENSIONAL OBJECTS WITH EMPHASIS ON BOOLEAN OPERATIONS AND A MODIFICATION OF THE RAY-CASTING METHOD

By

Michel R. Kokozaki

A Thesis

Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

Faculty of Natural and Applied Sciences

Notre Dame University

Louaizeh, Zouk Mosbeh, Lebanon

2000

MODELING OF COMPLEX THREE DIMENSIONAL OBJECTS WITH EMPHASIS ON BOOLEAN OPERATIONS AND A MODIFICATION OF THE RAY-CASTING METHOD

By

Michel R. Kokozaki

Approved by:

[illegible]

Khaldoun El-Khalidi: Assistant Professor of Computer Science.
Advisor.

[illegible]

Fouad Chedid: Associate Professor of Computer Science and Chairperson.
Member of Committee.

Hoda Maalouf

Hoda Maalouf: Assistant Professor of Computer Science.
Member of Committee.

[illegible]

Nasser Saad: Assistant Professor of Mathematics and Statistics.
Member of Committee.

Date of Thesis Defense: July 6, 2000

Notre Dame University

Abstract

MODELING OF COMPLEX THREE DIMENSIONAL OBJECTS WITH EMPHASIS ON BOOLEAN OPERATIONS AND A MODIFICATION OF THE RAY-CASTING METHOD

By

Michel R. Kokozaki

This thesis deals with the modeling of complex three-dimensional objects, and emphasizes on the Boolean operations between objects. It discusses a modification of the ray-casting method, which offers an improved performance over the regular ray-casting method.

A small simulation will be performed, on a personal computer, about how to draw three-dimensional primitives and shading them using the wire-frame technique, plus performing Boolean operations between two objects using the modified version of the ray-casting method. Data structures and the main drawing algorithms that were created for this purpose will also be discussed in order to make a clearer view of how these objects are represented on a computer screen.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

The author would like to thank Professor Khaldoun El-Khalidi for his support on the writing of this thesis.

In addition, the author wishes to express sincere appreciation to Professor Fouad Chedid for his advisory and assistance in the preparation of this manuscript, as well as Professor Jean Fares for his valuable guidance and tips.

Thanks also to Tony Salim, Joseph Daccache, and Naji El-Hayek; and thanks to all the people whom I may have forgotten to mention, and who helped me, directly or indirectly, with this thesis.

Special thanks to my parents whom, without them, I would not be acquiring my Master Thesis.

INTRODUCTION

Interactive graphics is a computer field that has been evolving exponentially since the computer itself was born. In the last few years, it has benefited from the steady and sometimes even spectacular reduction in the hardware price/performance ratio, and from the development of high-level, device-independent graphics packages that help make graphics programming rational and straightforward. Interactive graphics is now finally ready to fulfill its promise to provide us with pictorial communication and thus to become a major facilitator of man/machine interaction.

With the interactive graphics becoming more and more common in user interfaces and visualization of data and objects, the rendering of 3D objects has become dramatically more realistic, as evidenced by the mind-blowing computer-generated commercials and movie special effects. Techniques that were experimental in the early eighties are now standard practice, and more remarkable "photo realistic" effects are around the corner. The simpler kinds of pseudo-realism, which took hours of computer time per image in the early eighties, now are done routinely at animation rates (30 or more frames per second) on personal computers. Thus "real-time" vector displays in 1981 showed moving wire-frame objects made of tens of thousands of vectors without hidden-edge removal; in 1990 real-time raster displays can show not only the same kinds of line drawings but also moving objects composed of as many as one hundred thousand triangles rendered with Gouraud or Phong shading and specular highlights and with full hidden-surface removal. The highest performance systems provide real-time texture mapping, anti-aliasing, atmospheric attenuation for fog and haze, and other advanced effects.

Perhaps the most important new improvement in graphics is the increasing concern for modeling objects, not just for creating their pictures. Furthermore, interest is growing in describing the time-varying geometry and behavior of 3D objects. Thus graphics is increasingly concerned with simulation, animation, and a "back of physics" movement in both modeling and rendering in order to create objects that look and behave as realistically as possible.

As the tools and capabilities available become more and more sophisticated and complex, we need to be able to apply them effectively. Rendering is no longer the bottleneck. Therefore researchers are beginning to apply artificial intelligence techniques to assist in the design of object models, in motion planning, and in the layout of effective 2D and 3D graphical presentations.

Chapter 1

THE DEVELOPMENT OF 3D GRAPHICS

Computer graphics started with the display of data on hardcopy plotters and cathode ray tube (CRT) screens soon after the introduction of computers themselves. It has grown to include the creation, storage, and manipulation of models and images of objects. These models come from a diverse and expanding set of fields, and include physical, mathematical, engineering, architectural, and even conceptual (abstract) structures, natural phenomena, and so on. Computer graphics today is largely interactive: The user controls the contents, structure, and appearance of objects and of their displayed images by using input devices, such as a keyboard, mouse, or touch-sensitive panel on the screen. Because of the close relationship between the input devices and the display, the handling of such devices is included in computer graphics.

Until the early 1980s, computer graphics was a small, specialized field, largely because the hardware was expensive and graphics-based applications that were easy to use and cost-effective were few. Then, personal computers with built-in raster graphics displays, such as the Xerox Star and, later, the mass-produced, even less expensive Apple Macintosh and the IBM PC and its clones, popularized the use of bitmap graphics for user-computer interaction. A bitmap is a ones and zeros representation of the rectangular array of points (pixels or pels, short for "picture elements") on the screen. Once bitmap graphics became affordable, an explosion of easy-to-use and inexpensive graphics-based applications soon followed. Graphics-based user interfaces allowed millions of new users to control simple, low-cost application programs, such as spreadsheets, word processors, and drawing programs.

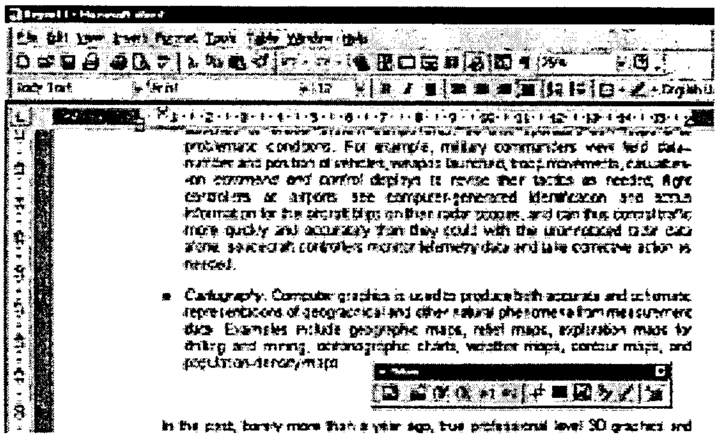
Even people who do not use computers in their daily work encounter computer graphics in television commercials and as cinematic special effects. Computer graphics is no longer a rarity. It is an integral part of all computer user interfaces, and is indispensable for visualizing two-dimensional (2D), three-dimensional (3D), and higher-dimensional objects: Areas as diverse as education, science, engineering, medicine, commerce, the military, advertising, and entertainment all rely on computer graphics. Learning how to program and use computers now includes learning how to use simple 2D graphics as a matter of routine.

__Differences Between Computer Graphics and Image Processing.__ Computer graphics concerns the pictorial synthesis of real or imaginary objects from their computer-based models, whereas the related field of image processing (also called picture processing) treats the converse process: the analysis of scenes, or the reconstruction of models of 2D or 3D objects from their pictures. Picture analysis is important in many areas: aerial surveillance photographs, slow-scan television images of the moon or of planets gathered from space probes, television images taken from an industrial robot's "eye", chromosome scans, X-ray images, computerized axial tomography (CAT) scans, and fingerprint analysis all exploit image-processing technology. Image processing has the sub areas image enhancement, pattern detection and recognition, and scene analysis and computer vision. Image enhancement deals with improving image quality by eliminating noise (extraneous or missing pixel data) or by enhancing contrast. Pattern detection and recognition deal with detecting and clarifying standard patterns and finding deviations (distortions) from these patterns. A particularly important example is optical character recognition (OCR) technology, which allows for the economical bulk input of pages of typeset, typewritten, or even hand-printed characters. Scene analysis and computer vision allow scientists to recognize and reconstruct a 3D model of a scene from several 2D images. An example is an industrial robot sensing the relative sizes, shapes, positions, and colors of parts on a conveyor belt.

Although both computer graphics and image processing deal with computer processing of pictures, they have until recently been quite separate disciplines. Now that they both use raster displays, however, the overlap between the two is growing, as is particularly evident in two areas. First, in interactive image processing, human input via menus and other graphical interaction techniques helps to control various sub-processes while transformations of continuous-tone images are shown on the screen in real time. For example, scanned-in photographs are electronically touched up, cropped, and combined with others (even with synthetically generated images) before publication. Second, simple image-processing operations are often used in computer graphics to help synthesize the image of a model. Certain ways of transforming and combining synthetic images depend largely on image-processing operations.

Representative Uses of Computer Graphics. Computer graphics is used today in many different areas of industry, business, government, education, entertainment, and, most recently, the home. The list of applications is enormous and is growing rapidly as computers with graphics capabilities become commodity products. Let's look at a representative sample of these areas:

User interfaces. As we mentioned, most applications that run on personal computers and workstations, and even those that run on terminals attached to time-shared computers and network compute servers, have user interfaces that rely on desktop window systems to manage multiple simultaneous activities, and on point-and-click facilities to allow users to select menu items, icons, and objects on the screen; typing is necessary only to input text to be stored and manipulated. Word-processing, spreadsheet, and desktop-publishing programs are typical applications that take advantage of such user-interface techniques. The following figure illustrates the User Interface.



problematic conditions. For example, military commanders were fed data—number and position of vehicles, weapons launched, troop movements, casualties—on command and control displays to revise their tactics as needed; flight controllers at airports see computer-generated identification and status information for the aircraft blips on their radar scopes, and can thus control traffic more quickly and accurately than they could with the unannotated radar data alone; spacecraft controllers monitor telemetry data and take corrective action as needed.

- Cartography. Computer graphics is used to produce both accurate and schematic representations of geographical and other natural phenomena from measurement data. Examples include geographic maps, relief maps, exploration maps for drilling and mining, oceanographic charts, weather maps, contour maps, and population-density maps

In the past, barely more than a year ago, true professional level 3D graphics and

Figure 1: A classic User Interface for the "Microsoft © Word" word processing application.

(Interactive) plotting in business, science, and technology. The next most common use of graphics today is probably to create 2D and 3D graphs of mathematical, physical, and economic functions; histograms, bar and pie charts; task-scheduling charts; inventory and production charts; and the like. All these are used to present meaningfully and concisely the trends and patterns gleaned from data, so as to clarify complex phenomena and to facilitate informed decision-making. Figure 2 shows a sample chart with some fictions statistics.

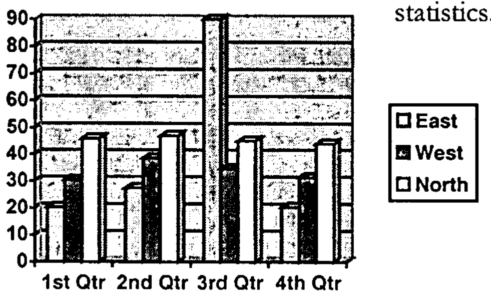


Figure 2: A sample chart.

Computer-aided drafting and design. In computer-aided design (CAD), interactive graphics is used to design components and systems of mechanical, electrical, electromechanical, and electronic devices, including structures such as buildings, automobile bodies, airplane and ship hulls, very-large-scale-integrated (VLSI) chips, optical systems, and telephone and computer networks. Sometimes, the user merely wants to produce the precise drawings of components and assemblies, as for online drafting or architectural blueprints. More frequently, however, the emphasis is on interacting with a computer-based model of the component or system being designed in order to test, for example, its structural, electrical, or thermal properties. Often, the model is interpreted by a simulator that feeds back the behavior of the system to the user for further interactive design and test cycles. After objects have been designed, utility programs can post-process the design database to make parts list, to process "bills of materials", to define numerical control tapes for cutting or drilling parts, and so on. See Figure 3.

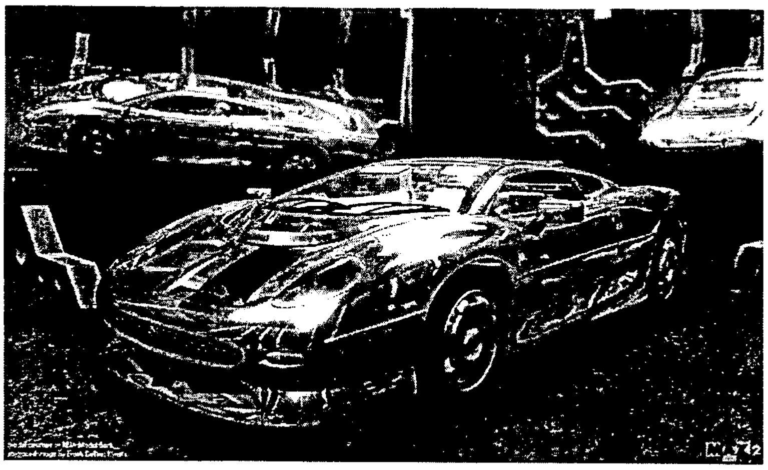


Figure 3: A Jaguar modeled using a very popular CAD program: 3D Studio Max.

Simulation and animation for scientific visualization and entertainment. Computer-produced animated movies and displays of the time-varying behavior of real and simulated objects are becoming increasingly popular for scientific and engineering visualization. We can use them to study abstract mathematical entities as well as mathematical models of such phenomena as fluid flow, relativity, nuclear and chemical reactions, physiological systems and organ function, and deformation of mechanical structures under various kinds of loads. Another advanced-technology area is interactive cartooning. The simpler kinds of systems for producing "flat" cartoons are becoming cost-effective in creating routine "in-between" frames that interpolate between two explicitly specified "key frames". Cartoon characters will increasingly be modeled in the computer as 3D shape descriptions whose movements are controlled by computer commands, rather than by the figures being drawn manually by cartoonists. Television commercials featuring flying logos and more exotic visual trickery have become a common, as have elegant special effects in movies. Sophisticated mechanisms are available to model the objects and to represent light and shadows.

Art and commerce. Overlapping the previous category is the use of computer graphics in art and advertising; here, computer graphics is used to produce pictures that express a

Teletext and Videotext terminals in public places such as museums, transportation terminals, supermarkets, and hotels, as well as in private homes, offer much simpler but still informative pictures that let users orient themselves, make choices, or even "teleshop" and conduct other business transactions. Finally, slide production for commercial, scientific, or educational presentations is another cost-effective use of graphics, given the steepy rising labor costs of the traditional means of creating such material. See Figure 4.

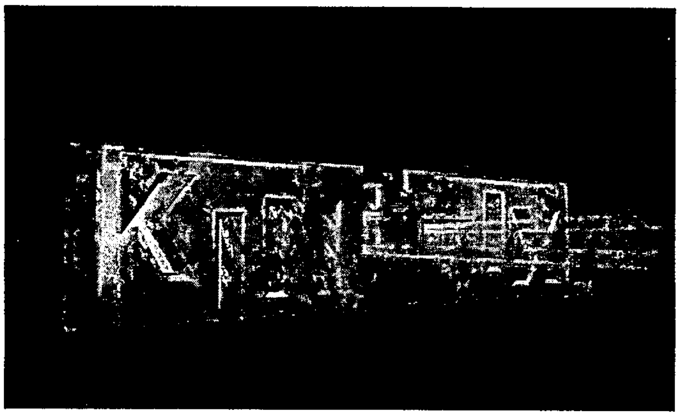


Figure 4: A slogan featuring a logo for the Kinetix Company, creators of 3D Studio.

Process control. Whereas flight simulators or arcade games let users interact with a simulation of a real or artificial world, many other applications enable people to interact with some aspect of the real world itself. Status displays in refineries, power plants, and computer networks show data values from sensors attached to critical system components, so that operators can respond to problematic conditions. For example, military commanders view field data--number and position of vehicles, weapon launched, troop movements, casualties--on *command and control* displays to revise their tactics as needed; flight controllers at airports see computer-generated identification and status information for the aircraft blips on their radar scopes, and can thus control traffic more quickly and accurately than they could with the unannotated radar data alone; spacecraft controllers monitor telemetry data and take corrective action as needed.

Cartography. Computer graphics is used to produce both accurate and schematic representations of geographical and other natural phenomena from measurement data. Examples include geographic maps, relief maps, exploration maps for drilling and mining, oceanographic charts, weather maps, contour maps, and population-density maps.

Today's 3D Workstation. In the past, barely more than 5 years ago, true professional level 3D graphics and animation was confined almost exclusively to very expensive workstations produced by Silicon Graphics Incorporated. The most important and most sophisticated professional-level packages used in Hollywood film effects--Alias and SoftImage--were available only for SGI machines. There were exceptions. AutoDesk, makers of AutoCAD, produced their original 3D Studio for Intel machines. Electric Image produced a rendering and animation package (without modeling) for the Macintosh. These products developed a strong professional following, but there remained a clear distinction between these "PC" or "desktop" products, and the true workstation applications used on SGI. Given the price of both the workstations and the software, professional level 3D was largely out of reach for the vast majority of otherwise interested people.

Today, as these words are being written, fully professional 3D has moved irrevocably to a PC platform that is already affordable to those with a serious interest. Affordable means a complete system running from $4,000 to about $10,000. These are machines that run 3D Studio MAX, LightWave, and SoftImage, all using Windows NT as an operating system. With a set-up like this, and the skills to use this software, the committed individual will find his or her way into professional 3D graphics and animation.

Hardware wise, you always want the ability to add the second processor, and there is no performance cost to adding it later rather than including it from the start.

The harder question is whether to buy that second Pentium right away. Windows NT made multi-threaded applications possible on the PC. To multi-thread an application means to write it such that it can allocate tasks between different processors, and where this is done completely and correctly, a dual processor machine comes very close to doubling its power. So the question comes down to the choice of application.

64 MB of RAM is the minimum and 128 is recommended. Professional 3D animation packages are memory-intensive far beyond any other kind of program you may be familiar with using. The moment you run out of RAM and start swapping out to your hard drive, things become unbearably slow.

If RAM is important, cache memory is even more so. All the Pentium Pro systems include 256KB of Level 2 cache per processor. The new Pentium II and Pentium III systems have 512KB.

The hard drive should be ultra-wide SCSI. Speed is very important here to improve the inevitable swapping from "virtual memory" and for screen playback of animation files.

A 21-inch monitor is obviously far superior to a 17 inch one because professional 3D applications necessarily divide the screen into multiple windows for simultaneous viewing from different positions. But a 21-inch monitor still costs $1,000 more than a 17 inch one, and even if you have that money, you may want it for a second processor or a beefier graphics card. On the other hand, you can't simply "upgrade" a monitor in the way you can add more RAM or a second Pentium. In any event, look for a refresh rate no slower than 75Hz. Intensive hours of 3D graphics work can be hard on the eyes, and noticeable flicker on the screen promotes fatigue.

Chapter 2

REPRESENTING PRIMITIVE OBJECTS IN 3D SPACE

Here is a three-dimensional coordinate system.

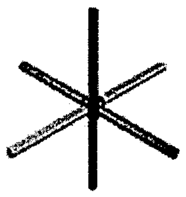


Figure 5: A 3D coordinate system.

We are looking at the origin (0,0,0) from above and somewhat over to the left so that we can see the whole scene unobstructed. The blue axis is the vertical one, called y. Positive y values are up and negative ones are down. Let's assume that our axes extend exactly 1 unit from the origin. Thus the point (0,1,0) is at the top end of the y (blue) axis, and (0, -1, 0) is at the bottom end.

The yellow axis is the horizontal one, called x. (1,0,0) is at the right tip of this axis and (-1,0,0) is at the left tip as we look from the front. The green axis is z, the depth axis. (0,0, -1) is at the far end away from us, while (0,0,1) is at the tip nearest to us as we look from the front. Notice that depth increases in this way as the z value decreases. This creates what is called a "right-handed" coordinate system, and is the most common convention in use today.

We will create the simplest possible object by placing points in the space. A cube can be completely described if we know the location of each of its eight corners. We will create a cube to exactly enclose our 3 axes. The first point we will place is the one nearest to the viewpoint we established in the last picture.

Let's change view to look down from the top. The ability to move between different views of the same scene is the most essential skill to develop, and it doesn't come naturally to anyone. Notice that our first (pink) point is placed at 1 on the z-axis and -1 on the x-axis. From the top view, we can't tell its location on the y-axis.

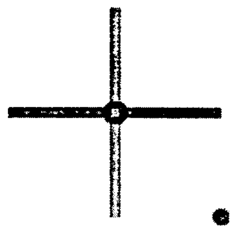


Figure 6: The top view.

Now let's switch to a front view. Now we can see that the point is placed at 1 on the y-axis. The point is therefore at (-1,1,1).

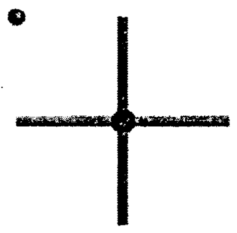


Figure 7: The front view.

Let us look at a third view from the left side to confirm this location. It takes time to get accustomed to thinking and seeing in this way.

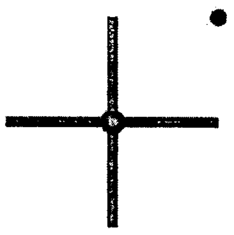


Figure 8: The left view.

Let's swing back to a viewpoint close to where we began. You can readily see how difficult it would be to determine the location of the pink point from this view alone.

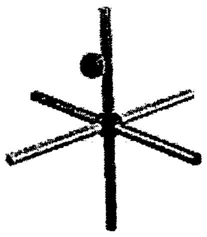


Figure 9: A perspective view.

Building an Object in 3D Space. Next we add three other points to define the four corners of the upper corners of the cube.

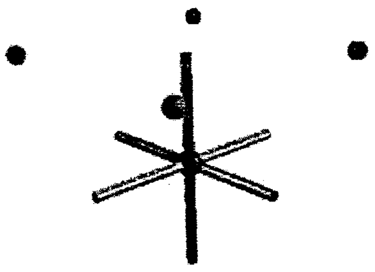


Figure 10: Adding upper rectangle points to draw the cube.

All of these points sit on the same plane, the plane in which the y value is always equal to 1. The four points are (1,1,1), (1,1, -1), (-1,1, -1) (-1,1,1). This is how we learn to work in 3D space.

Let's add the lower group of points to make all the corners of the cube. The new points are all on the plane y = -1.

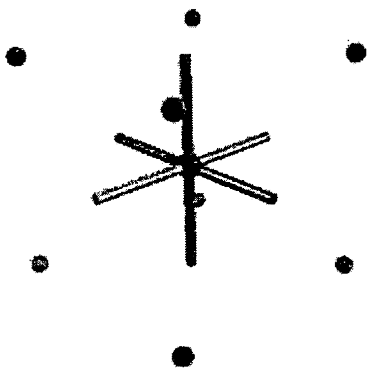


Figure 11: Adding lower rectangle points to draw a cube.

Now how do we go from the points alone to a solid object? We use the points to represent the vertices (corners) of six separate squares that form the surface of the cube. The points are used to define linked polygons that define the shape of the cube. Using points, and connecting them to create polygons, we create a Model that can be viewed, or "rendered" by 3D software.

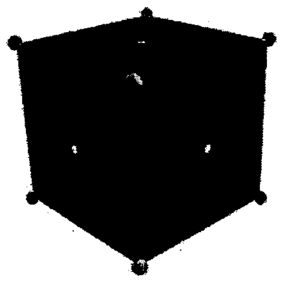


Figure 12: The final rendered box model

Notice how the very tips of the 3 axes are visible in this rendering. This helps us to understand where the cube is located in our space. To clarify things even more, we will make the cube translucent, so that the axes may be seen within.

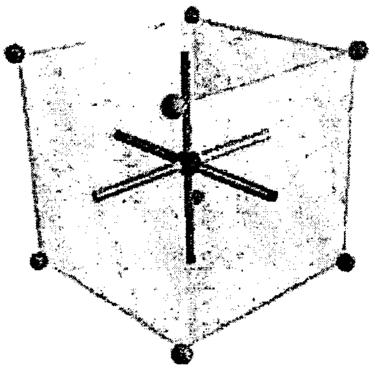


Figure 13: The cube after being made a little transparent to show the axes.

Now that we see how a 3D object is created by using points to create surface polygons (in this case, squares), we can return to the ideas of transformation. By transforming the upper group of points downward in y (from y=1 to y=.5) and transforming the lower group upward in y (from y=-1 to y=-.5), the cube is compressed vertically. This demonstrates how scaling (resizing) is achieved by transformation of coordinates.

Figure 14: A scale transformation.

By transforming all the points together, we can rotate the object around the z-axis.

Figure 15: A rotation transformation.

Finally, we translate (move) the entire object back away from our point of view.

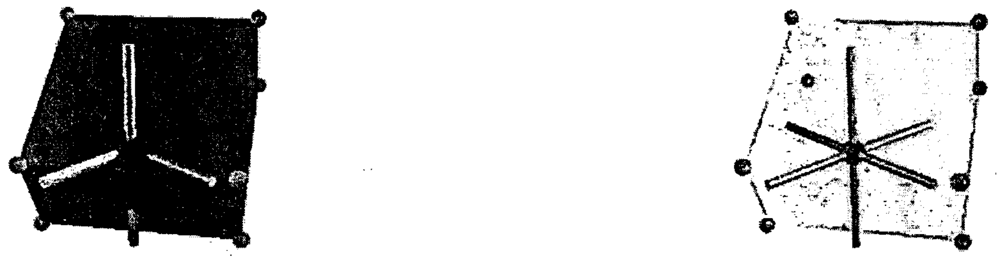
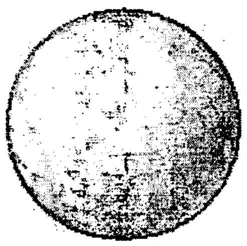


Figure 16: A translation transformation.

<u>Building Smooth Surfaces.</u> A sphere is the ultimate smooth surface. If we create it out of a mesh of flat polygons, we get something like this.

Figure 17: A sphere.



A close-up view shows a kind of faceted ball, which is not smooth.

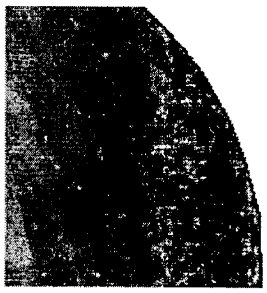


Figure 18: A close view to a sphere.

Each little four-sided polygon on the surface has its own normal. That is to say that each polygon faces a slightly different direction toward the light. Now, we might imagine dividing the ball up into smaller and smaller units, and if the squares were many and small enough, a good illusion would be created of a smooth surface. All computer graphics is based on this kind of digital principle. We can scan a continuous tone picture into a bitmap composed of discrete little points very convincingly. But we need not go this far with our 3D model of a sphere. We can keep the relatively rough polygonal model we have and make it look smooth when it renders.

This is how the very same model looks when rendered with "smooth shading" instead of "flat shading."

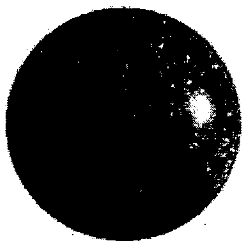


Figure 19: The same sphere rendered using smooth shading

In flat shading, all of the points on a polygon surface have the same normal. They all point in the same direction. In smooth shading, the lighting for every point on the surface of a polygon is computed separately, and the normal is adjusted for each point so that there is a slight change of direction against the light at each pixel as it is drawn. To achieve smooth shading, the idea of a normal is expanded from what we already understand. We are no longer concerned with the normals of the individual faces, but rather of the normals at the points (the vertices) where these faces meet. The normal at a vertex is determined by averaging the normals of all the polygons that share it. Thus there is a kind of direction toward the light at a given point on the surface of the sphere.

This idea may seem very peculiar, but it is a core concept found again and again in mathematics. If we pick a point on a sphere, we can imagine a plane that is tangent to that point--a plane that embeds that point alone and no other on the sphere. Once we have a plane, we can easily imagine a normal to that plane, and that normal is, in effect the normal to the point on the sphere. The following figure should strengthen the idea of the continuous change of normals across the surface of a sphere.

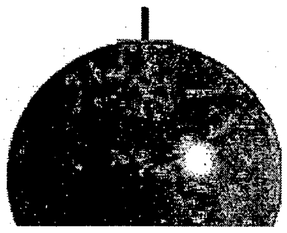


Figure 20: A tangent plane to any point on the sphere, with its normal.

<u>Primitive Objects.</u> Each application will have its own special approaches to modeling. But most of the basic modeling concepts transcend individual applications and are found in one form or another on all. The first step in all modeling is working with Primitives. Every application will generate certain basic models automatically. Just as a vector graphics program like Corel Draw will generate 2D circles and squares, a 3D application will always generate spheres, cubes, cylinders, pyramids, and usually cones.

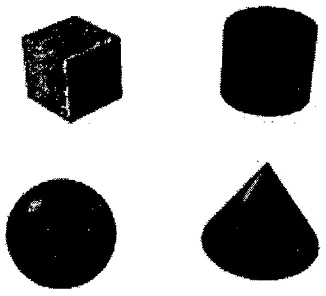


Figure 21: Some basic primitives.

In the high-end programs, the artist can manipulate the individual points and polygons on these primitives, so that a sphere primitive may form the starting point for a highly detailed human head. In all applications, however, the geometry of the primitive can be manipulated as a whole, and a very broad range of effects is possible from such simple controls.

Primitives can be merged in space to create composite objects.

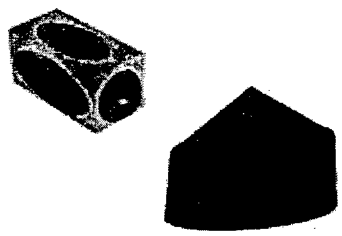


Figure 22: Combing multiple objects to create composite objects.

<u>Rendering Techniques.</u> Recalling that deep question of philosophy, which asks whether a tree, falling in a deserted forest, makes a sound. Just so, our 3D models can't be seen unless there is someone there to see them. In this case, however, that "someone" is not a person, but a hypothetical camera, and the process of seeing is called

Rendering. But the models themselves are just the data necessary to produce a rendered image, and the rendering process itself is half the story.

To strip the rendering process to its bare essentials, a hypothetical camera is placed in the same 3D coordinate space that contains our models. It therefore has a location in (x, y, z) coordinates. It is a single point that represents the spot from which an "eye" looks at the scene, and so it is often called the viewpoint. Like a real eye and real camera, it must have an orientation. It must be looking in a certain direction. And it must also have a field of vision, an angle projecting out from the viewpoint. Objects that fall within this angle can be seen, and those falling outside it cannot. This is all exactly like a real camera and like our own eyes, although a camera with a zoom lens can expand and contract its field of vision without moving the camera itself. So can our hypothetical camera in 3D coordinate space.

In the rendering process, the camera "takes a picture" of the objects in the scene as seen from the camera's viewpoint, given its direction and field of view. The rendering is achieved mathematically, by tracing lines from the vertices of all the polygons in all the objects in the scene back to the viewpoint of the camera. This enables the "rendering engine," as the software that produces the rendering is often called, to reconstruct the polygons as they would be projected on a flat surface, just as light is focused though the center of a camera lens onto the film plane. A major aspect of this process is determining which polygons (surfaces) on the objects are obstructed by other polygons from the camera's viewpoint. Surfaces that are behind other surfaces obviously should not be rendered.

Once the rendering engine has determined which polygons are visible and how they should be projected on the rendering surface, called the "viewing plane," they must be drawn as pixels to produce a bitmap. Each pixel in a bitmap must be assigned a color. How does the rendering engine assign a color?

Here we approach one of the most fascinating aspects of 3D graphics. 3D graphics applications model reality in two distinct ways. The most obvious is in geometry. A 3D object is no mere flat representation, but rather like a sculpture that can be viewed from different directions to reveal its full three-dimensional substance. But 3D graphics also models the way that light interacts with objects and with our eyes. Objects don't have just a color the way they do in painting and 2D computer graphics. They have surface qualities that reveal themselves under the particular lighting in the scene. If there is no lighting, the object is rendered black regardless of what color it would appear in light.

The following is a rendering of a sphere. There is no lighting in the scene, and therefore the sphere is completely black. Notice that the background is white. In 3D graphics applications, the background of a scene is typically assigned a fixed color (or an image) that may have no relationship with the scene.

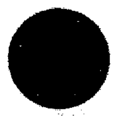


Figure 23: A rendering of a sphere without any lighting.

Now we add a light, actually a couple of lights. A spotlight is pointing down on the object from above and behind, reflecting off the surface of the sphere. This simple highlight gives the viewer a completely different reading of the scene.

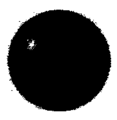


Figure 24: The same sphere with a couple of lights added.

The object is now perceived of (barely) as a 3D sphere, and not just a flat circle. The color of the sphere is black, but it is made of a somewhat shiny material, like a billiard ball. The highlight, called a "specular reflection" is white; telling the viewer the light shining on the object is white. To compare, look at a rendering using a yellow spotlight.

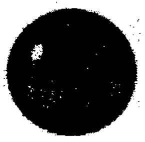


Figure 25: The same sphere with a yellow spotlight.

The object being black, the only visual clue to its three-dimensional nature is the highlight. Shadows cannot be seen on a black surface. If we change the object's surface to a light blue, the shading across the surface (revealed as darker shades of blue and gray) do much more to reveal the shape.

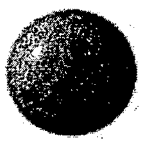


Figure 26: The same sphere with a light blue surface color.

Chapter 3

DIFFERENT MODELING TECHNIQUES

Let us now talk about the different techniques used in modeling three-dimensional objects. We have considered using primitive, and then editing or deforming them to get the geometry we want. An enormous amount of modeling is possible from this approach. But often we must create even the basic geometry from scratch. This is one of the most creative aspects of 3D graphics, both for the programmers developing the tools and for the artists using them.

Primitive Instancing. In primitive instancing, the modeling system defines a set of primitive 3D solid shapes that are relevant to the application area. These primitives are typically parameterized not just in terms of transformations, but on other properties as well. For example, one primitive object may be a regular pyramid with a user-defined number of faces meeting at the apex. Primitive instances are similar to parameterized objects, except that the objects are solids. A parameterized primitive may be thought of as defining a family of parts whose members vary in a few parameters, an important CAD concept known as *group technology*. Primitive instancing is often used for relatively complex objects, such as gears or bolts, that are tedious to define in terms of Boolean combinations of simpler objects, yet are readily characterized by a few high-level parameters. For example, a gear may be parameterized by its diameter or number of teeth, as shown in the figure below.

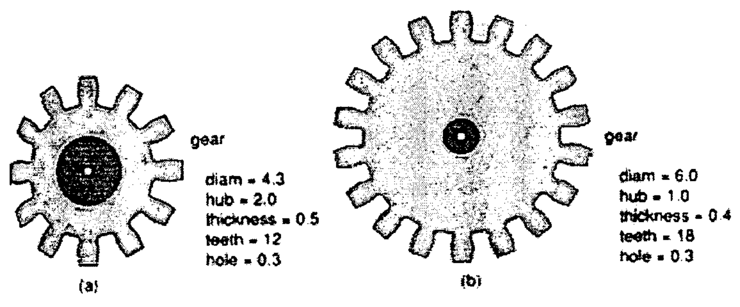


Figure 27: Two gears defined by primitive instancing.

Although we can build up a hierarchy of primitive instances, each leaf-node instance is still a separately defined object. In primitive instancing, no provisions are made for combining objects to form a new higher-level object, using, for example, the regularized Boolean set operations. Thus, the only way to create a new kind of object is to write the code that defines it. Similarly, the routines that draw the objects or determine their mass properties must be written individually for each primitive.

Sweep Representations. Sweeping an object along a trajectory through space defines a new object, called a *sweep*. The simplest kind of sweep is defined by a 2D area swept along a linear path normal to the plane of the area to create a volume. This is known as a *translational sweep* or *extrusion* and is a natural way to represent objects made by extruding metal or plastic through a die with the desired cross-section. In these simple cases, each sweep's volume is simply the swept object's area times the length of the sweep. Simple extensions involve scaling the cross-section as it is swept to produce a tapered object or sweeping the cross-section along a linear path that is not normal to it. *Rotational sweeps* are defined by rotating an area about an axis. The illustration below shows two objects and simple translational and rotational sweeps generated using them.

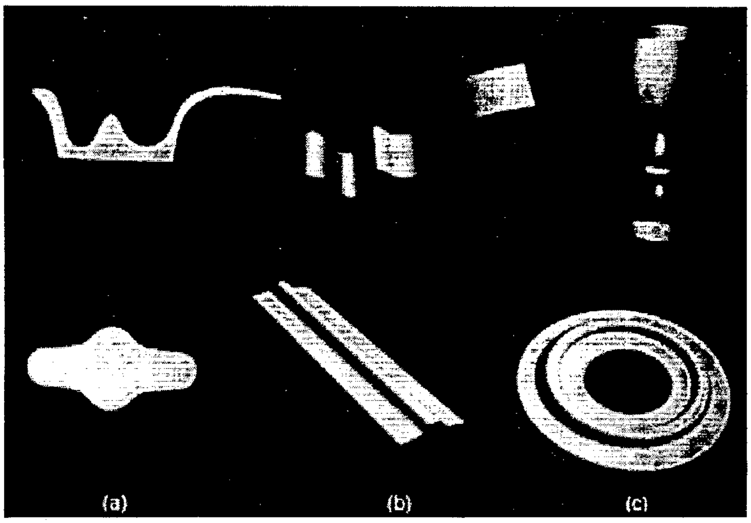


Figure 28: Sweeps. Here in (a) we have 2D areas used to define translational sweeps (b) and rotational sweeps (c).

The object being swept does not need to be 2D. Sweeps of solids are useful in modeling the region swept out by a machine-tool cutting head or robot following a path, as shown in the next figure. Sweeps whose generating area or volume changes in size, shape, or orientaion as they are swept and that follow an arbitrary curved trajectory are called *general sweeps*. General sweeps of 2D cross-sections are known as generalized cylinders in computer vision and are usually modeled as parameterized 2D cross-sections swept at right angles along an arbitrary curve. General sweeps are particularly difficult to model efficiently. For example, the trajectory and object shape may make the swept object intersect itself, making volume calculations complicated. As well, general sweeps do not always generate solids. For example, sweeping a 2D area in its own plane generates another 2D area.

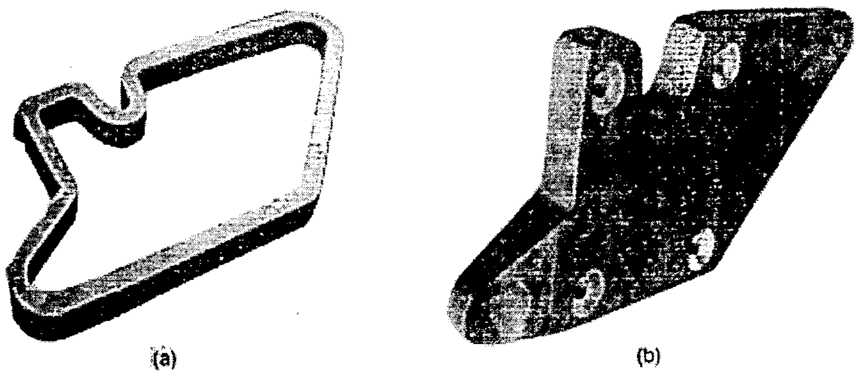


Figure 29: Path of a cutting tool (a), modeled as a solid sweep, is used to define model of an aircraft part in (b).

Boundary Representations. *Boundary Representations*, also known as *b-reps*, describe an object in terms of its surface boundaries: vertices, edges, and faces. Some b-reps are restricted to planar, polygonal boundaries, and may even require faces to be convex polygons or triangles. Alternatively, they can also be represented as surface patches if the algorithms that process the representation can treat the resulting intersection curves, which will, in general, be of higher order than the original surfaces. B-reps grew out of the simple vector representations and are used in many current modeling systems. Because of their importance in graphics, a number of efficient techniques have been developed to create smooth shaded pictures of polygonal objects.

Many b-rep systems support only solids whose boundaries are *2-manifolds*. By definition, every point on a 2-manifold has some arbitrarily small neighborhood of points around it that can be considered topologically the same as a disk in the plane. This means that there is a continuous one-to-one correspondence between the neighborhood of points and the disk, as shown in the figures in (a) and (b) below. For example, if more than two faces share an edge, as in (c), any neighborhood of a point on that edge contains points from each of those faces. It is intuitively obvious that there is no continuous one-to-one correspondence between this neighborhood and a disk in the plane, although the mathematical proof is by no means trivial. Thus, the surface in (c) is not a 2-manifold.

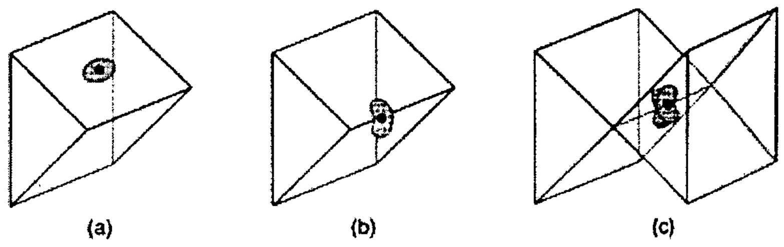


Figure 30: A 2-manifold surface.

Polyhedra and Euler's Formula. A *polyhedron* is a solid that is bounded by a set of polygons whose edges are each a member of an even number of polygons (exactly two polygons in the case of 2-manifolds) and that satisfies some additional constraints. A *simple polyhedron* is one that can be deformed into a sphere; that is, a polyhedron that, unlike a torus, has no holes. The b-rep of a simple polyhedron satisfies Euler's formula, which expresses an invariant relationship among the number of vertices, edges, and faces of a simple polyhedron:

$$V - E + F = 2$$

Where V is the number of vertices, E is the number of edges, and F is the number of faces. The following figure shows some simple polyhedra and their numbers of vertices, edges, and faces. Note that the formula still applies if curved edges and nonplanar faces are allowed. Euler's formula by itself states necessary but not sufficient conditions for an object to be a simple polyhedron. One can construct objects that satisfy the formula but do not bound a volume, by attaching one or more dangling faces or edges to an otherwise valid solid. Additional constraints are needed to guarantee that the object is a solid: each edge must connect two vertices and be shared by exactly two faces, at least three edges must meet at each vertex, and faces must not interpenetrate.

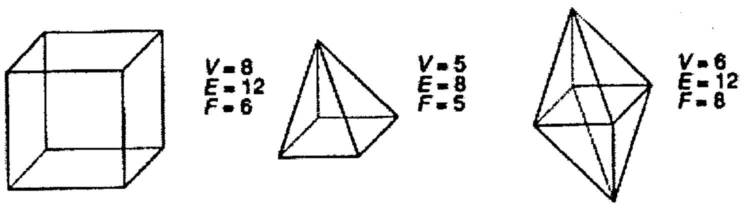


Figure 31: Some simple polyhedra with their V – E + F = 2.

A generalization of Euler's formula applies to 2-manifolds that have faces with holes:

$$V - E + F - H = 2(C - G)$$

Where H is the number of holes in the faces, G is the number of holes that pass through the object, and C is the number of separate components (parts) of the object, as shown below. If an object has a single component, its G is known as its *genus*; if it has multiple components, then its G is the sum of the genera of its components. As before, additional constraints are also needed to guarantee that the objects are solids.

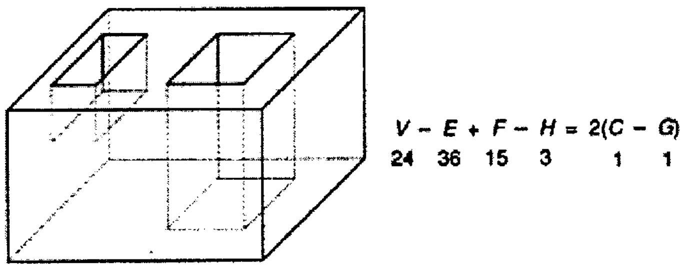


Figure 32: A polyhedron with two holes in its top face and one hole in its bottom face.

Spatial-Partitioning Representations. In *spatial-partitioning* representations, a solid is decomposed into a collection of adjoining, non-intersecting solids that are more primitive than, although not necessarily of the same type as, the original solid. Primitives may vary in type, size, position, parameterization, and orientation, much like the different-shaped blocks in a child's block set. How far we decompose objects depends on how primitive the solids must be in order to perform readily the operations of interest.

Cell Decomposition. One of the most general forms of spatial partitioning is called *cell decomposition*. Each cell-decomposition system defines a set of primitive cells that are typically parameterized and are often curved. Cell decomposition differs from primitive instancing in that we can compose more complex objects from simple, primitive ones in a bottom-up fashion by "gluing" them together. The *glue* operation can be thought of as a restricted form of union in which the objects must not intersect. Further restrictions on gluing cells often require that two cells share a single point, edge, or face. Although cell-decomposition representation of an object is unambiguous, it is not necessarily unique, as shown below.

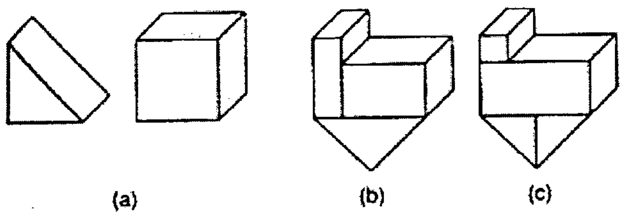


Figure 33: A non-unique cell-decomposition representation of an object. The cells in (a) may be transformed to construct the same object shown in (b) and (c) in different ways.

Cell decompositions are also difficult to validate, since each pair of cells must potentially be tested for intersection. Nevertheless, cell decomposition is an important representation for use in finite element analysis.

Spatial-Occupancy Enumeration. *Spatial-Occupancy enumeration* is a special case of cell decomposition in which the solid is decomposed into identical cells arranged in a fixed, regular grid. These cells are often voxels (volume elements), in analogy to pixels. The figure below shows an object represented by spatial-occupancy enumeration.

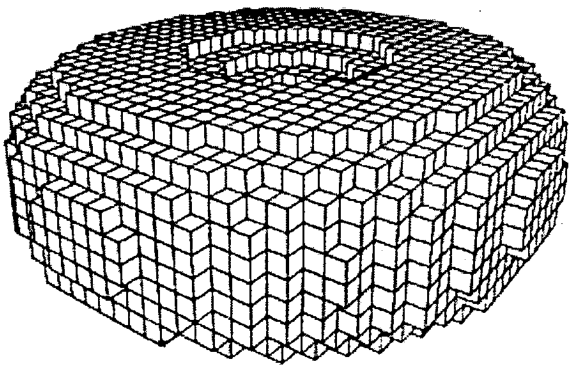


Figure 34: A torus represented by spatial-occupancy enumeration.

The most common cell type is the cube, and the representation of space as a regular array of cubes is called a *cuberille*. When representing an object using spatial-occupancy enumeration, we control only the presence or absence of a single cell at each position in the grid. To represent an object, we need only to decide which cells are occupied and which are not. The object can thus be encoded by a unique and unambiguous list of occupied cells. It is easy to find out whether a cell is inside or outside of the solid, and determining whether two objects are adjacent is simple as well. Spatial-occupancy enumeration is often used in biomedical applications to represent volumetric data obtained from sources such as computerized axial tomography (CAT) scans.

For all of its advantages, however, spatial-occupancy enumeration has a number of obvious failings that parallel those of representing a 2D shape by a 1-bit-deep bitmap. There is no concept of partial occupancy. Thus, many solids can be only approximated; the torus of the previous figure is an example. If the cells are cubes, then the only objects that can be represented exactly are those whose faces are parallel to the cube sides and whose vertices fall exactly on the grid. Like pixels in a bitmap, cells may in principle be made as small as desired to increase the accuracy of the representation. Space becomes an important issue, however, since up to n^3 occupied cells are needed to represent an object at a resolution of n voxels in each of the three dimensions.

<u>**Boolean Set Operations.**</u> No matter how we represent objects, we would like to be able to combine them in order to make new ones. One of the most intuitive and popular methods for combining objects is by Boolean set operations, such as union, difference, and intersection, as shown in the illustration below.

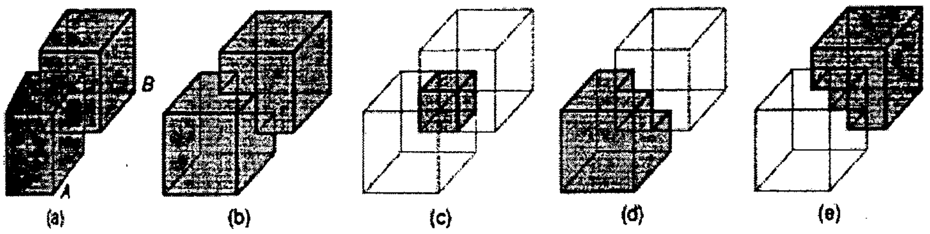


Figure 35: (a) The objects A and B, (b) A 4 B, (c) A 3 B, (d) A - B, (e) B - A.

These operations are the 3D equivalents of the familiar 2D Boolean operations. Applying an ordinary Boolean set operation to two solid objects, however, does not necessarily yield a solid object. For example, the ordinary Boolean intersections of the cubes in the below illustration through (e) are a solid, a plane, a line, a point, and the null object, respectively.

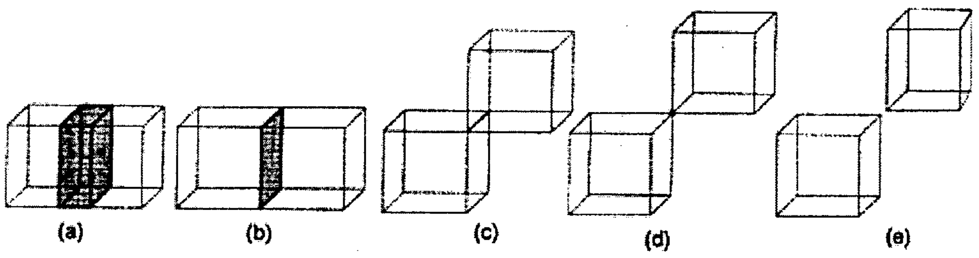


Figure 36: The ordinary Boolean intersection of two cubes may produce: (a) A solid, (b) a plane, (c) a line, (d) a point, or (e) the null set.

Regularized Boolean Set Operators. Rather than using the ordinary Boolean set operators, we will instead use the *regularized* Boolean set operators, denoted 4*, 3*, and -*, and defined such that operations on solids always yield solids. For example, the regularized Boolean intersection of the objects shown in the above illustration is the same as their ordinary Boolean intersection in cases (a) and (e), but is empty in (b) through (d).

To explore the difference between ordinary and regularized operators, we can consider any object to be defined by a set of points, partitioned into interior points and boundary points, as illustrated below.

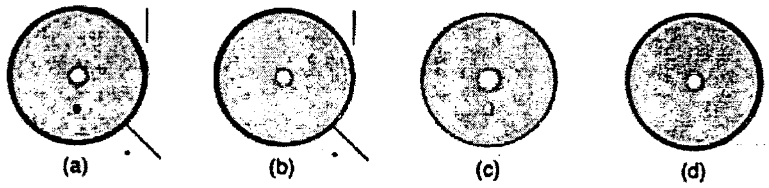


Figure 37: Difference between ordinary and regularized operators.

In (a), Boundary points are those points whose distance from the object and the object's complement is zero. Boundary points need not be part of the object. A closed set contains all its boundary points, whereas an open set contains none. The union of a set with the set of its boundary points is known as the set's closure, as shown in (b), which is a self closed set. The boundary of a closed set is the set of its boundary points, whereas the interior, shown in (c), consists of all of the set's other points, and thus is the complement of the boundary with respect to the object. The regularization of a set is defined as the closure of the set's interior points. In (d) we see the closure of the object in (c) and, therefore, the regularization of the object in (a). A set that is equal to its own regularization is known as a regular set. Note that a regular set can contain no boundary point that is not adjacent to some interior point; thus, it can have no "dangling" boundary points, lines, or surfaces. We can define each regularized Boolean set operator in terms of the corresponding ordinary Boolean set operator as:

$$A \text{ op}^* B = \text{closure (interior } (A \text{ op } B))$$

Where op is one of 4, 3, or -. The regularized Boolean set operators produce only regular sets when applied to regular sets.

We now compare the ordinary and regularized Boolean set operations as performed on regular sets. Consider the two objects in (a) of the illustration below, positioned as shown in (b). The ordinary Boolean intersection of two objects contains the intersection of the interior and boundary of each object with the interior and boundary of the other, as shown in (c). In contrast, the regularized Boolean intersection of two objects. Shown in (d), contains the intersection of their interiors and the intersection of the interior of each with the boundary of the other, but only a subset of the intersection of their boundaries. The criterion used to define this subset determines how regularized Boolean intersection differs from ordinary Boolean intersection, in which all parts of the intersection of the boundaries are included. Intuitively, a piece of the boundary-boundary intersection is included in the regularized Boolean intersection if and only if the interiors of both objects lie on the same side of this piece of shared boundary. Since the interior points of both objects that are directly adjacent to that piece of boundary are in the intersection, the boundary piece must also be included to maintain closure. Consider the case of a piece of shared boundary that lies in coplanar faces of two polyhedra. Determining whether the interiors lie on the same side of a shared boundary is simple if both objects are defined such that their surface normals point outward (or inward). The interiors are on the same side if the normals point in the same direction. Thus, segment AB in (d) is included. Remember that those parts of one object's boundary that intersect with the other object's interior, such as segment BC, are always included.

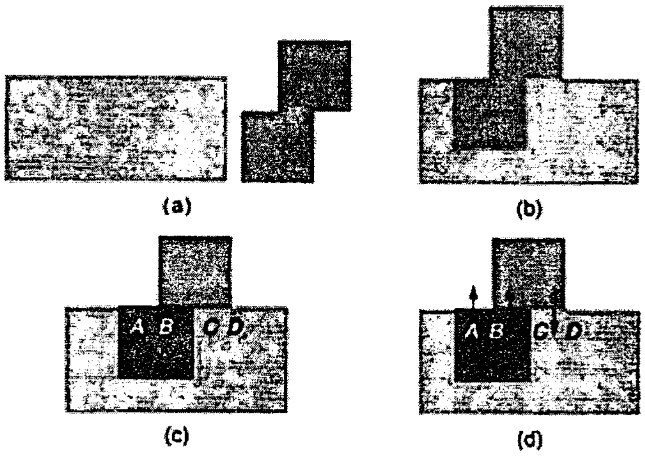


Figure 38: Comparing the ordinary and regularized Boolean set operations.

Consider what happens when the interiors of the objects lie on the opposite sides of the shared boundary, as is the case with segment CD. In such cases, none of the interior points adjacent to the boundary are included in the intersection. Thus, the piece of shared boundary is not adjacent to any interior points of the resulting object and therefore is not included in the regularized intersection. This additional restriction on which pieces of shared boundary are included ensures that the resulting object is a regular set. The surface normal of each face of the resulting object's boundary is the surface normal of whichever surface(s) contributed that part of the boundary. Having determined which faces lie in the boundary, we include an edge or vertex of the boundary-boundary intersection in the boundary of the intersection if it is adjacent to one of these faces.

The result of each regularized operator may be defined in terms of the ordinary operators applied to the boundaries and interiors of the objects. The table below shows how the regularized operators are defined for any objects A and B; and the figure below the table shows the results of performing the operations. A_b and A_i are the A's boundary and interior, respectively. A_b 3 B_b same is that part of the boundary shared by A and B for which Ai and Bi lie on the same side. This is the case for some point b on the shared boundary if at least one point i adjacent to it is a member of both A_i and B_i. A_b 3 B_b diff is that part of the boundary shared by A and B for which A_i and B_i lie on opposite sides.

This is true for b if it is adjacent to no such point *i*. Each regularized operator is defined by the union of the sets associated with those rows that have a * in the operator's column.

Set	$A \cup^* B$	$A \cap^* B$	$A -^* B$
$A_i \cap B_i$	●	●	
$A_i - B$	●		●
$B_i - A$	●		
$A_b \cap B_i$		●	
$B_b \cap A_i$		●	●
$A_b - B$	●		●
$B_b - A$	●		
$A_b \cap B_b$ *same*	●	●	
$A_b \cap B_b$ *diff*			●

Table 1: Defining the regularized operators for any objects A and B

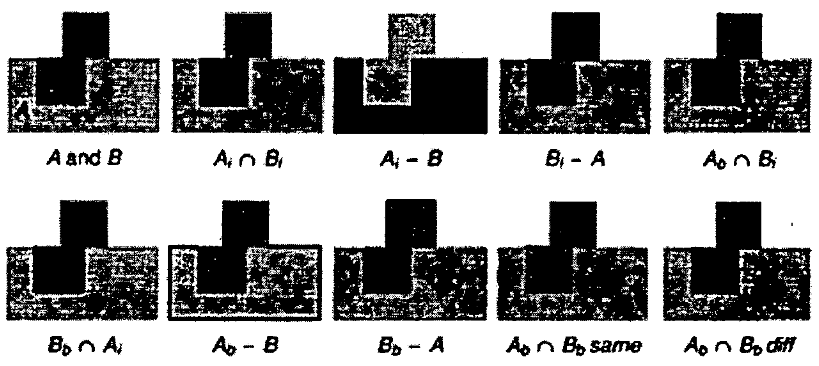


Figure 39: The results of performing the Boolean operations.

Note that, in all cases, each piece of the resulting object's boundary is on the boundary of one or both of the original objects. When computing A 4* B or A 3* B, the surface normal of a face of the result is inherited from the surface normal of the corresponding face of one or both original objects. In the case of A -* B, however, the surface normal of each face of the result at which B has been used to excavate A must point in the opposite direction from B's surface normal at that face. This corresponds to the boundary pieces A_b 3 B_b diff and B_b 3 A_i. Alternatively, A -* B may be rewritten as A

3* ~~B~~. We can obtain ~~B~~ (the complement of B) by complementing B's interior and reversing the normals of its boundary.

The regularized Boolean set operators have been used as a user-interface technique to build complex objects from simple ones in most of the representation schemes we shall discuss. They are also included explicitly in one of the schemes, constructive solid geometry.

Chapter 4

WORK EXPERIENCE

After presenting the different modeling techniques, we emphasize the Boolean operations method, and suggest an improvement to it. My main work study is presented in this chapter, along with some simulation algorithms to emphasize this study.

Constructive Solid Geometry (CSG). Constructive Solid Geometry, or simply CSG, is another technique for solid modeling where the volumes occupied by overlapping 3D objects using set operations are combined together. This modeling method creates a new volume by applying the union, intersection, or difference to two specified volumes. The illustrations below show examples for forming new shapes using the set operations.

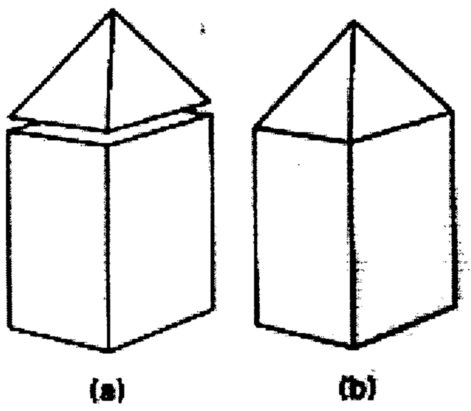


Figure 40: Forming new shapes using the set operations.

In this picture, part (a) shows a block and a pyramid, which are placed adjacent to each other. Specifying the union operation, we obtain the combined object shown in part (b).

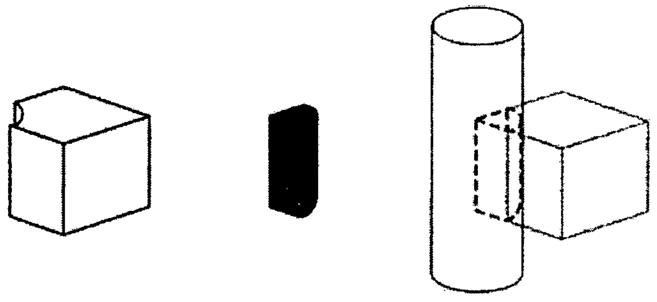


Figure 41: Showing the intersection and difference operations.

In the right part of the above figure, we see a block and a cylinder with overlapping volumes. Using the intersection operation, we obtain the resulting solid shown in the middle of the picture. With a difference operation, we can get the solid shown in the leftmost part of the image.

A CSG application starts with an initial set of 3D objects (primitives), such as blocks, pyramids, cylinders, cones, spheres, and closed spline surfaces.

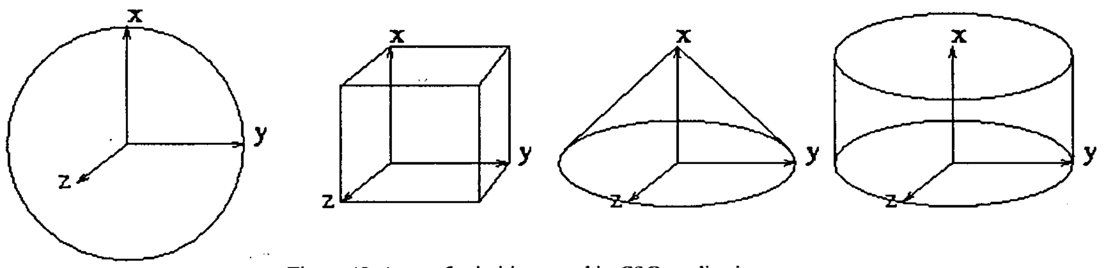


Figure 42: A set of primitives used in CSG applications.

The primitives can be provided by the CSG package as menu selections, or the primitives themselves could be formed using sweep methods, spline constructions, or other modeling procedures. To create a new 3D shape using CSG methods, we first select two primitives and drag them into position in some region of space. Then we select an operation (union, intersection, or difference) for combining the volumes of the two primitives. Now we have a new object, in addition to the primitives, that we can use to form other objects. We continue to construct new shapes, using combinations of primitives and the objects created at each step, until we have the final shape. An object designed with this procedure is represented with a binary tree.

Binary Tree Representation of CSG. In constructive solid geometry (CSG), simple primitives are combined by means of regularized Boolean set operators that are included directly in the representation. An object is stored as a tree with operators at the internal nodes and simple primitives at the leaves as shown in both illustrations below.

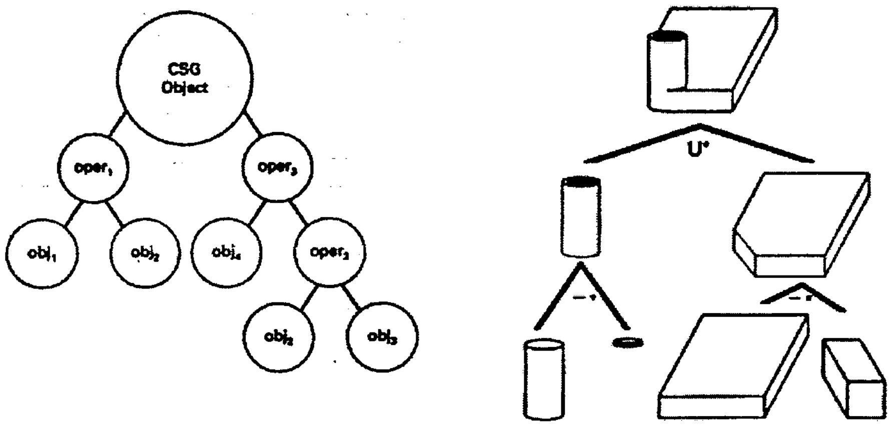


Figure 43: Tree representations for a CSG object.

Some nodes represent Boolean operators, whereas others perform translation, rotation, and scaling. Since Boolean operations are not, in general, commutative, the edges of the tree are ordered.

To determine physical properties or to make pictures, we must be able to combine the properties of the leaves to obtain the properties of the root. The general processing strategy is a depth-first free walk, to combine nodes from the leaves on up the tree. The complexity of this task depends on the representation in which the leaf objects are stored and on whether a full representation of the composite object at the tree's root must actually be produced. In some implementations, the primitives are simple solids, such as cubes or spheres, ensuring that all regularized combinations are valid solids as well. In other systems, primitives include half-spaces, which themselves are not bounded solids. For example, a cube can be defined as the intersection of six half-spaces, or a finite cylinder as in infinite cylinder that is capped off at the top and bottom by planar half-spaces. Using half-spaces introduces a validity problem, since not all combinations produce solids. Half-spaces are useful, however, for operations such as slicing an object by a plane, which might otherwise be performed by using the face of another solid object. Without half-spaces, extra overhead is introduced, since the regularized Boolean set operations must be performed with the full object doing the slicing, even if only a single slicing face is of interest.

We can think of the cell-decomposition and spatial-occupancy enumeration techniques as special cases of CSG in which the only operator is the implicit glue operator: the union of two objects that may touch, but must have disjoint interiors (i.e., the objects must have a null regularized Boolean intersection).

CSG does not provide a unique representation. This can be particularly confusing in a system that lets the user manipulate the leaf objects with tweaking operators. Applying the same operation to two objects that are initially the same can yield two different results. Nevertheless, the ability to edit models by deleting, adding, replacing, and modifying subtrees, coupled with the relatively compact form in which models are stored, have made CSG one of the dominant solid modeling representations.

Ray-Casting Methods Used for Boolean Operations. Ray-casting methods are commonly used to implement constructive solid geometry operations when objects are described with boundary representations. We apply ray casting by constructing composite objects in world coordinates with the *xy* plane corresponding to the pixel plane of a video monitor. This plane is then referred to as the "firing plane" since we fire a ray from each pixel position through the objects that are to be combined (see illustration below). We then determine surface intersections along each ray path, and sort the intersection points according to the distance from the firing plane. The surface limits for the composite object are then determined by the specified set operation.

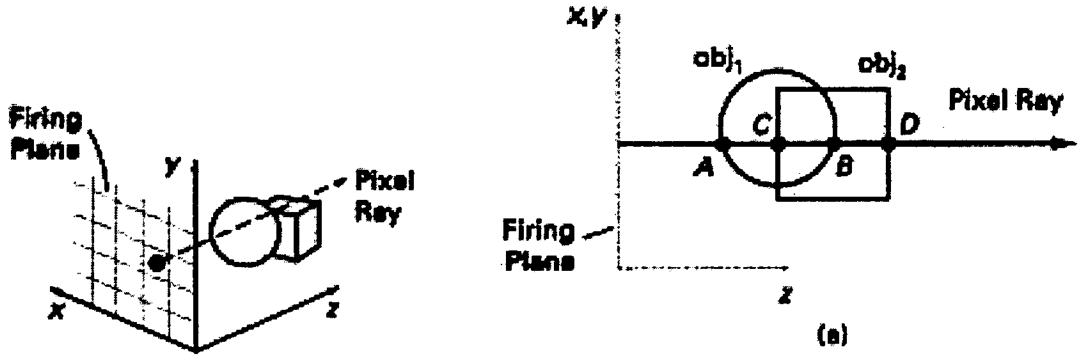


Figure 44: Using ray-casting methods to implement constructive solid geometry.

An example of the ray-casting determination of surface limits for a CSG object is given in the right side of the illustration above, which shows *yz* cross sections of two primitives and the path of a pixel ray perpendicular to the firing plane.

<u>Operation</u>	<u>Surface Limits</u>
Union	A, D
Intersection	C, B
Difference (obj2 – obj1)	B, D

Table 2: Showing the surface limits of the objects in figure 36.

- For the union operation, the new volume is the combined interior regions occupied by either or both primitives.

- For the intersection operation, the new volume is the interior region common to both primitives.

- ➢ And a difference operation subtracts the volume of one primitive from the other.

Each primitive can be defined in its own local (modeling) coordinates. Then, a composite shape can be formed by specifying the modeling-transformation matrices that would place two primitives in an overlapping position in world coordinates. The inverse of these modeling matrices can then be used to transform the pixel rays to modeling coordinates, where the surface-intersection calculations are carried out for the individual primitives. Then surface intersection for the two objects are sorted and used to determine the composite object limits according to the specified set operation. This procedure is repeated for each pair of objects that are to be combined in the CSG tree for a particular object.

Once a CSG object has been designed, ray casting is used to determine physical properties, such as volume and mass. To determine the volume of the object, we can divide the firing plane into any number of small squares, as shown in the illustration below.

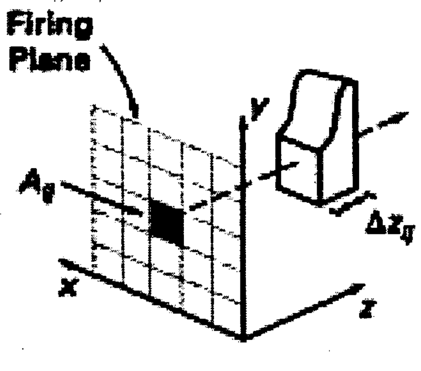


Figure 45: Determining the volume of the object.

__Volume and Mass Calculation of the Resulting Objects.__ We can then approximate the volume V_{ij} of the object for a cross-sectional slice with area A_{ij} along the path of a ray from the square at position (i, j) as:

$$V_{ij} \approx A_{ij} \Delta z_{ij}$$

Where Δz_{ij} is the depth of the object along the ray from position (i, j). If the object has internal holes, Δz_{ij} is the sum of the distances between pairs of intersection points along the ray. The total volume of the CSG object is then calculated as:

$$V \approx \sum_{i,j} Vij$$

Given the density function, $\rho(x, y, z)$, for the object, we can approximate the mass along the ray from position (i, j) as:

$$m_{ij} \approx A_{ij} \int \rho(x_{ij}, y_{ij}, z)dz$$

Where the one-dimensional integral can often be approximated without actually carrying out the integration, depending on the form of the density function. The total mass of the CSG object is then approximated as:

$$m \approx \sum_{i,j} Mij$$

Other physical properties, such as center of mass and moment of inertia, can be obtained with similar calculations. We can improve the approximate calculations for the values of the physical properties by taking finer subdivisions in the firing plane.

In chapter 3, we talked about the different modeling techniques, mainly primitive instancing, sweeps, b-reps, spatial partitioning (including cell decomposition and spatial-occupancy enumeration). In this chapter we talked in detail about Boolean operations between objects and constructive solid geometry (CSG).

<u>**Comparison of the Different Modeling Representations.**</u> We will now compare these different modeling techniques and show their advantages and disadvantages, in the following table:

Accuracy	**Domain**	**Uniqueness**	**Validity**	**Closure**	**Compactness and Efficiency**
Spatial-partitioning and **polygonal b-rep** methods produce only approximations for many objects. Systems that support high-quality graphics often use **CSG** with non-polyhedral primitives and **b-reps** that allow curved surfaces.	The domain of objects that can be represented by both **primitive instancing** and **sweeps** is limited. In comparison, **spatial-partitioning** approaches can represent any solid. **B-reps** can be used to represent a very wide class of objects.	Only the **spatial-occupancy enumeration** approach guarantees the uniqueness of a representation: there is only one way to represent an object with a specified size and position. **Primitive instancing** does not guarantee uniqueness in general.	Among all the representations, **b-reps** stand out as being the most difficult to validate. Only simple local syntactic checking needs to be done to validate a **CSG** tree. No checking is needed for **spatial-occupancy enumeration**.	Primitives created using **primitive instancing** cannot be combined at all, and simple **sweeps** are not closed under Boolean operations. Therefore, neither is typically used as an internal representation in modeling systems.	The advantages of **CSG** are its compactness and the ability to record Boolean operations and changes of transformations quickly. More work may be done evaluating a **b-rep** than evaluating the equivalent **CSG**.

Table 3: Comparison of different modeling representations.

When we talk about modeling, we surely need to talk about a way to design, model, and structure our geometrical models. In this chapter, I will present the data structures and classes that I have created in order to model and display 2D and 3D models correctly on the computer screen.

In order to display models on the screen, we have to present a certain inter-relation between the various objects.

Two dimensional objects, like points, lines, circles, ellipses, and the like, are the foundation for what will be later called "3D Models", or primitives. For example, lines and rectangles will be used to draw 3D boxes; circles and ellipses will be used to draw spheres, etc...

Data Definitions of 2D Primitives. I will present the following definitions of 2D objects' data structures, together with the definitions of the classes that I have created for them, which are demonstrated in the Appendix.

- A point:

Property	**Property Description**	**Method**	**Method Description**
x-coordinate (integer)	The x-coordinate of the point.	Draw	Draws the point on its container.
y-coordinate (integer)	The y-coordinate of the point.	Clear	Deletes the point from its container.
Container (object)	The object that is containing the point.		

Table 4: The point data structure.

- A polygon (including lines):

Property	**Property Description**	**Method**	**Method Description**
Collection of points	These points delimit the polygon.	Draw	Draws the polygon on its container.
Container (object)	The object that is containing the polygon.	Clear	Deletes the polygon from its container.

DrawStyle (integer)	Determines the style of the drawing (solid, dotted, etc...).		
ClosePolygon (Boolean)	Determine whether we want to close the polygon.		

Table 5: The polygon data structure.

➢ An ellipse (including circles):

Property	**Property Description**	**Method**	**Method Description**
Center point	The center point of the ellipse.	Draw	Draws the ellipse on its container.
Radius (single)	The radius of the ellipse.	Clear	Deletes the ellipse from its container.
StartAngle (single)	The starting angle, in radians, of an ellipse arc.		
EndAngle (single)	The ending angle, in radians, of an ellipse arc.		

AspectRatio (single)	The aspect ratio of the vertical to horizontal axes		
DrawStyle (integer)	Determines the style of the drawing (solid, dotted, etc...).		
Container (object)	The object that is containing the ellipse.		

Table 6: The ellipse data structure.

If we are to draw a perfect circle, we set its "AspectRatio" property to 1, and the "Radius" property will set the circle's radius.

If we are to draw an ellipse, we set its "AspectRatio" to a value other than 1. This property will then be the ratio between the vertical radius and the horizontal one, and the "Radius" property will then describe the greater between the vertical and the horizontal radii.

The "StartAngle" and "EndAngle" properties are, respectively, the start and end angle in radians, if we want to draw an arc instead of a closed ellipse or circle.

The ellipse also takes as properties, a center point, a draw style, and a container object.

<u>Data Definitions of 3D Primitives.</u> After these definitions of 2D objects, I have created so far the following 3D object models.

➢ A box:

Property	**Property Description**	**Method**	**Method Description**
Collection of points	The 4 points that delimit the box.	Draw	Draws the box on its container.
Container (object)	The object that is containing the box.	Clear	Deletes the box from its container.

Table 7: The box data structure.

I have found some contribution while working on the box object, which is drawing 3D boxes with only 4 input points, as follows:

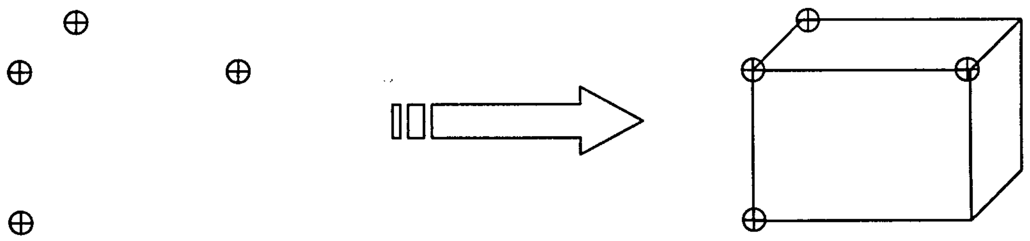


Figure 46: Drawing a box using 4 input vertices.

➢ A sphere:

Property	Property Description	Method	Method Description
Center point	The center point of the sphere.	Draw	Draws the sphere on its container.
Radius (single)	The radius of the sphere.	Clear	Deletes the sphere from its container.
Container (object)	The object that is containing the sphere.	IsInside	Determines if a given point is inside a sphere or not.

Table 8: The sphere data structure.

➢ A pyramid:

Property	Property Description	Method	Method Description
Collection of points	The 4 points that delimit the pyramid.	Draw	Draws the pyramid on its container.
Container (object)	The object that is containing the pyramid.	Clear	Deletes the pyramid from its container.

Table 9: The pyramid data structure.

Here also, I have found some contribution while working on the pyramid object, which is drawing 3D pyramids with only 4 input points, as follows:

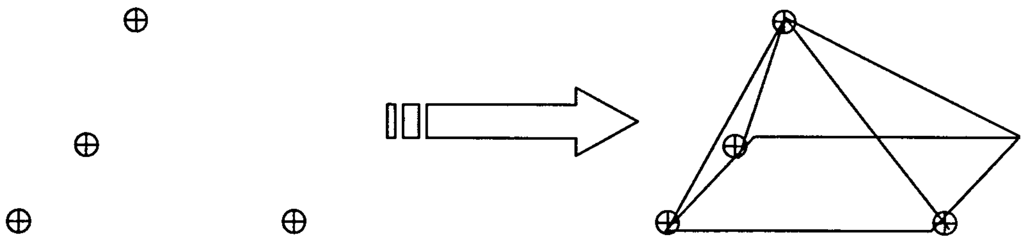


Figure 47: Drawing a pyramid using 4 input vertices.

In addition to this, all the 2D and 3D primitives have a Name property, which is a string data type. This property is referred to later in the implementation of the Boolean operations.

I will now talk about collections of classes. For example, we have a points collection, containing a set of points, and this collection has properties and methods just like any other class. This is the outline definition of the points collection as I defined it:

1. An Add method to add a point to the collection
2. A Remove method to remove a specific point.
3. A RemoveAll method to remove all the points from the collection.

Just like this collection of points, we have a collection of polygons, a collection of ellipses, a collection of pyramids, a collection of boxes, and a collection of spheres.

<u>Simulating the Boolean Operations Between 2 Objects.</u> I will now present a simple simulation that features a modification of the ray-casting method for the Boolean operations between 2 objects, and talk about the algorithm that I have enhanced to improve performance.

Suppose we have 2 objects on the computer screen, as follows:

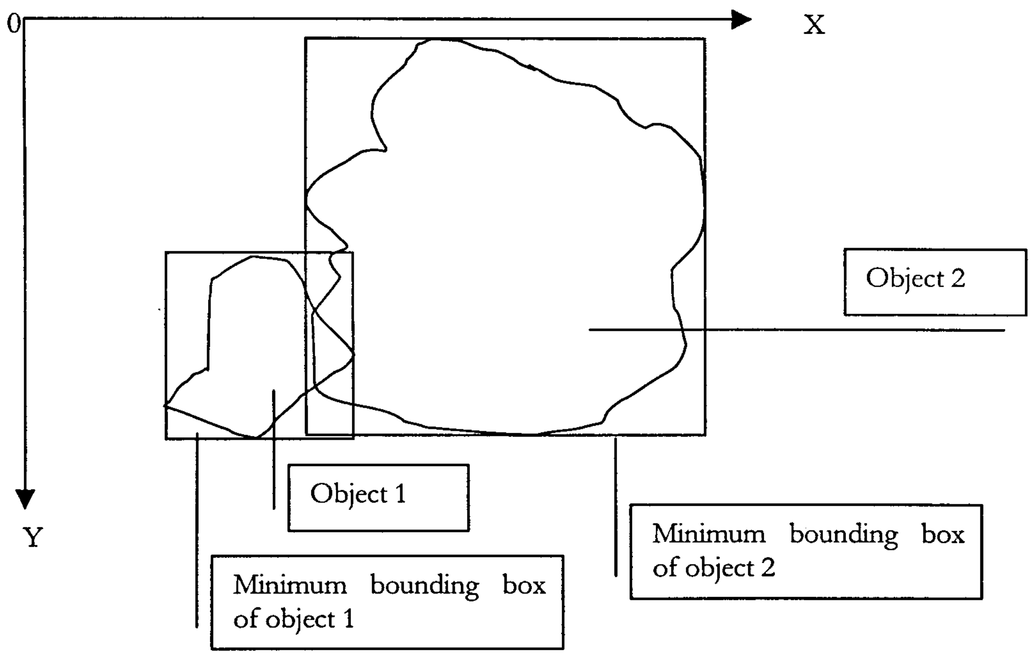

0
X
Y
Object 2
Object 1
Minimum bounding box of object 1
Minimum bounding box of object 2

Figure 48: 2 objects on a computer screen.

- For intersection, or 3, we find the minimum bounding box of the smallest object (in volume). And then, for each point inside this bounding box, if this point belongs to object 1 **<u>and</u>** object 2, then this point belongs to their intersection, and so on. By choosing the smallest object, we minimize the number of vertices that are to be tested for their inclusion inside that object, and we do not need to test all the pixels on the screen at all, like the regular ray-casting method suggests. The resulting intersection object is shaded below.

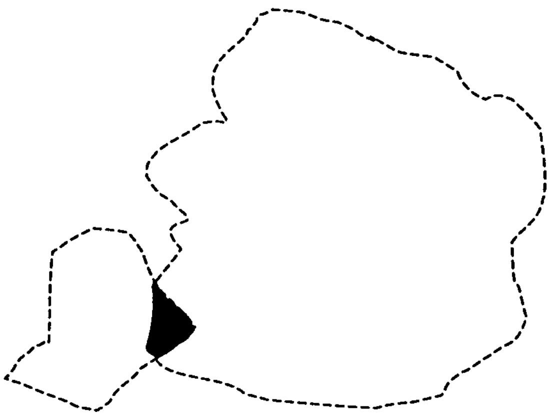


Figure 49: The resulting Boolean intersection object.

- As for union, or 4, we find the minimum bounding box for both objects. And then, for each point inside each bounding box, beginning with the smallest one, if this point belongs to object 1 **or** object 2, then this point belongs to their union. We then consider the other object, but now we take all the pixels that are inside the bounding box not contained inside the first bounding box, in order to avoid repetitive pixel comparisons. This way also, we are minimizing the number of vertices that are to be tested for their inclusion inside the objects, and we do not need to test all the pixels on the screen at all. The resulting union object is shaded below.

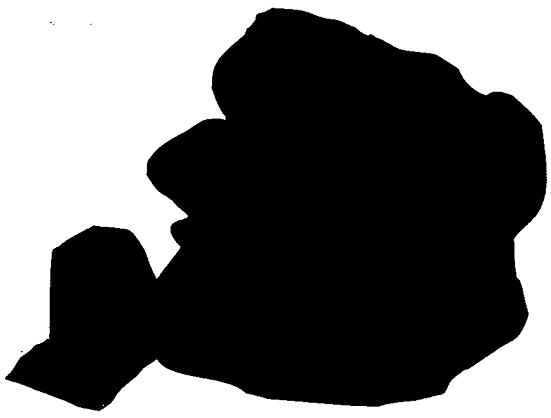


Figure 50: The resulting Boolean union object.

- For difference, or -, we find the minimum bounding box for the object that we want to subtract from, object 1 – object 2 here. And then, for each point inside that bounding box, if this point belongs to object 1 **and not** to object 2, then this point belongs to their difference. Here also, we are minimizing the number of vertices that are to be tested for their inclusion inside the objects, and we do not need to test all the pixels on the screen nor on the subtracted from object. The resulting difference object is shaded below.

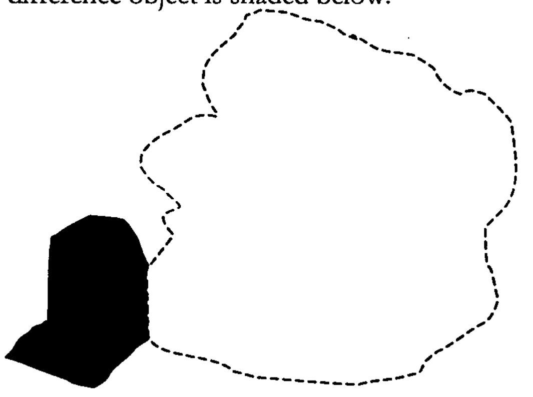
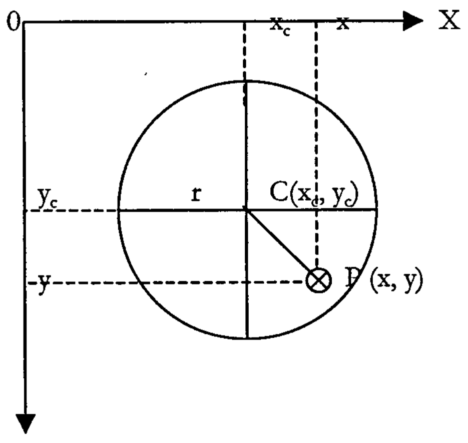


Figure 51: The resulting Boolean difference object.

In my simulation project, I have tried this technique on spheres, as shown in the Appendix. But first of all, let me define a method to find out if a point P(x, y) is inside a given sphere of center C(x_c, y_c) and radius r, as illustrated below:



Y

Figure 52: Determining if a point is inside a sphere.

In order for P to be inside the sphere, we just need to prove that CP <= r.

But we have $CP^2 = |x - x_c|^2 + |y - y_c|^2$.

So $CP = \sqrt{|x - xc|2 + |y - yc|2}$

And the condition becomes as follows:

$\sqrt{|x - xc|2 + |y - yc|2} <= r$.

P is located outside of the sphere if:

$\sqrt{|x - xc|2 + |y - yc|2} > r$.

Chapter 5

CONCLUSION

In this thesis, we have presented the basic aspects of computer graphics for modeling of three-dimensional objects. To cite, we talked about primitive instancing, sweep representations, boundary representations, spatial-partitioning enumeration, and Boolean operations between objects.

Different Boolean operation methods were discussed, mainly regularized Boolean set operators, Constructive Solid Geometry (CSG), and the binary tree representation of objects. We also discussed methods for calculating the volume and the mass of the resulting objects.

A modification of the ray-casting method is also discussed, which uses the bounding boxes of the 3D objects and makes the calculations in them. In particular, we simulated an operation for finding the resulting objects derived from the Boolean operations between two spheres.

Future research may deal with more improvements to the various drawing objects, like cones and cylinders. More improvements can be made to the Boolean operations between objects by modifying the ray-casting methods in order to let it tolerate some model accuracy in favor of the speed of calculations, with the object's quality staying slightly the same.

GLOSSARY

3D Coordinate Space. There are six directions ranged about us in three pairs: left and right--the horizontal directions, up and down--the vertical directions, forward and backwards (or front and behind)--for which we have no general name. We have a pure Cartesian space of 3 dimensions, and call the dimensions X, Y and Z. We choose a point in this space and call it the origin. As the origin, it is the location where X=0, Y=0, and Z=0, and the point is designated as (0,0,0). We run three axes right through this point, the X, Y, and Z-axes, each perpendicular to the other two. Now we can designate the exact location of any point in our space relative to the origin. For example, a point at (3,2,1) can be reached by starting at the origin (0,0,0) moving 3 units of length (perhaps inches) in the X direction, then moving 2 units in the Y direction, and finally 1 unit in the Z direction. The numbers are called "coordinates" and therefore the defined space is called a 3D coordinate space. The coordinates can be negative as well as positive

Boolean Operations. Boolean Operations are modeling methods that make use of two objects that overlap and therefore share part of the same space. In Boolean union, the geometry of the overlapping area is eliminated and a single object is created from the two using all of the exposed surface area. Union is generally used to merge objects that are most easily built from component parts that have been modeled separately. Boolean subtraction is used to sculpt out the overlapping volume from one object or the other. After the operation, one object is left, minus its overlapping region with the other object. Boolean intersection preserves the overlapping region only, eliminating all the rest of both objects.

Minimum Bounding Box. It is the box surrounding a three-dimensional object, having the minimum volume.

Normals. Normals can be associated with the flat surfaces of the polygons, and also with the individual points that make up the vertices where polygons meet on the surface of a model. This technique is used in rendering to create the appearance of curved surfaces rather those flat, faceted sides. Such vertex normals can be directly assigned in the model file, but are usually computed during rendering by averaging the normals of the adjacent polygons.

Orthogonal Viewing. An orthogonal view or projection eliminates the effect of distance from a viewpoint, and therefore provides a useful means of locating points and objects in 3D space. An orthogonal view effectively eliminates one dimension. For example, when working in a front orthogonal view, points can be moved in the x and y dimensions, but not in z.

Primitives. Primitives are the basic 3D geometric shapes, like spheres, cubes, cylinders (sometimes called disks), cones, and pyramids, that are automatically generated by 3D modeling applications, and which therefore need not be constructed from scratch. The most considerable amount of modeling begins with primitives, which are then edited and used with other primitives to create more complex objects.

Rendering. Rendering is the process of producing bitmapped images from a view of 3D models in a 3D scene. An animation is a series of such renderings, each with the scene slightly changed.

Scene. A scene is a file containing all the information necessary to identify and position all of the models, lights and cameras for rendering. A scene can be identified with the 3D coordinate space in which rendering takes place. This space is often called the "global" coordinate space; as opposed to the "local" coordinate spaces associated with each individual object in the scene.

Surfacing. Surfacing (sometimes called shading) is the process of assigning values to the surfaces of objects. These values generally control the manner in which the surface interacts with light in the scene to create the object's color, specularity (highlights), reflective qualities, transparency, and (if the surface is at all transparent) refraction. Surfacing controls those qualities that suggest the material that an object is made of, whether wood or plastic or metal, and the art of surfacing is coming to understand how the range of surfacing parameters interact to create realistic or imaginative effects.

Transformation of Coordinates. It is the most general term for movement, rotation, and scaling of objects by changing their coordinates.

BIBLIOGRAPHY

[1] Hearn, Donald and Baker, M. Pauline. *Computer Graphics*, 2nd ed. Prentice Hall International, 1994.

[2] Foley, James D., Van Dam, Andries, Feiner, Steven K., and Hughes, John F. C*omputer Graphics Principles and Practice,* 2nd ed. Addison-Wesley Publishing Company, 1993.

[3] Elliott, Steven and Miller, Philip. *Inside 3D Studio MAX 2,* Vol. 1. New Riders Publishing, 1998.

[4] Yasumuro, Y., Quian Chen, Chuhara, K. *3D Modeling of Human Hand With Motion Constraints.* IEEE. Graduate School of Information Sciences, Nama Institute of Sciences and Technology. Japan.

[5] Kappel, M.R. *An Ellipse-Drawing Algorithm for Raster Displays.* NATO ASI Series, Springer-Verlag, Berlin, 1985.

[6] International Standards Organization, *International Standard Information Processing Systems-Computer Graphics-Graphical Kernel System for Three Dimensions (GKS-3D) Functional Description,* ISO Document Number 8805:1988(E), American National Standards Institute, New York, 1988.

[7] Polevoi, Robert. *Through the Looking Glass.* 1997.

[8] Polevoi, Robert. *Intersecting Spaces.* 1997.

[9] Polevoi, Robert. *Today's 3-D Workstation.* 1997.Polevoi, Robert. *Let There Be Light.* 1997.

[10] Polevoi, Robert. *Boolean Operations.* 1997.

[11] University of Waterloo. *Modeling and CSG, CS488/688: Introduction to Interactive Computer Graphics.* 1999.

[12] University of Waterloo. *Surface Information, CS488/688: Introduction to Interactive Computer Graphics.* 1999.

[13] In Kyu Park, Sank Uk Lee. *Geometric Modeling from Scattered 3D Range Data.* Real Time Vision Lab, Seoul national university, South Korea.

[14] Kaplan, G. and E. Lerner. *Realism in Synthetic Speech.* IEEE Spectrum, April 1985.

[15] International Standards Organization, *Information Processing Text and Office Systems Standard Page Description Language (SPDL),* ISO Document Number JTC1 SC18/WG8N561, American National Standards Institute, New York.

[16] Barsky, B.A. *A Description and Evaluation of Various 3-D Models.* IEEE Computer Graphics and Applications. 1984.

Appendix

SOURCE CODE AND OUTPUT

In this appendix, I will present the main algorithm for the Boolean operations of the simulation program, which is written in Visual Basic.

This pseudo-code is executed when the "OK" button is pressed on the Boolean operations window.

```
Select case of Boolean operation

Case of "Intersection"

    Find the minimum bounding box of the smallest sphere.

    For I = x-coordinate of top-leftmost pixel to bottom-rightmost

        For J = y-coordinate of top-leftmost pixel to bottom-rightmost

            If that (I, J) pixel is in that sphere and in the other one then

                We have found an intersection pixel.

            End If

        Next J

    Next I
```

Case of "Union"

```
Find the minimum bounding box of the smallest sphere.
For I = x-coordinate of top-leftmost pixel to bottom-rightmost
    For J = y-coordinate of top-leftmost pixel to bottom-rightmost
        If that (I, J) pixel is in that sphere or in the other one then
            We have found a union pixel.
        End If
    Next J
Next I
Move to the other bounding box
For I = x-coordinate of top-leftmost pixel to bottom-rightmost
    For J = y-coordinate of top-leftmost pixel to bottom-rightmost
        If that (I, J) pixel is in that sphere or in the other one
    without being in the first bounding box then
            We have found a union pixel.
        End If
    Next J
Next I
```

Case of "Difference"

```
    Find the minimum bounding box of the first sphere.

    For I = x-coordinate of top-leftmost pixel to bottom-rightmost

        For J = y-coordinate of top-leftmost pixel to bottom-rightmost

            If that (I, J) pixel is in that sphere **and not** in the other one then

                We have found a difference pixel.

            End If

        Next J

    Next I
```

Here are some sample output screens of the simulation:

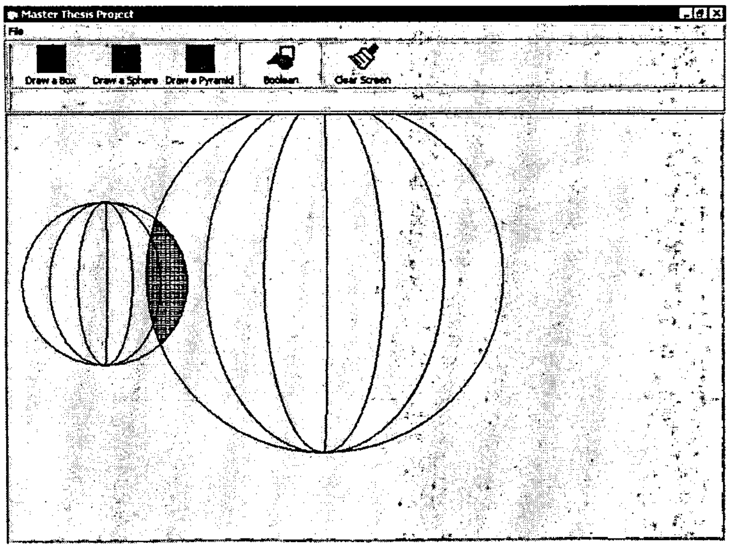


Figure 53: The intersection.

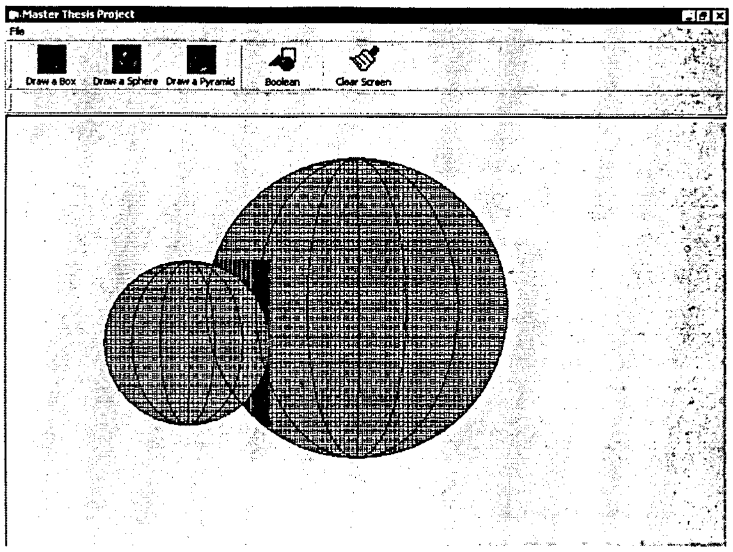


Figure 54: The union.

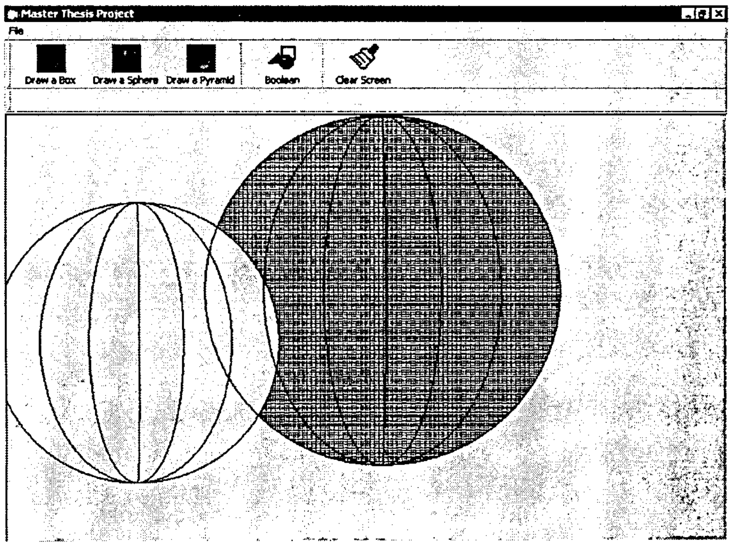


Figure 55: The difference.