WORLDVIEW

SUBMITTED IN PARTIAL FULFILLMENT FOR THE AWARD OF DEGREE OF:

BACHELOR OF TECHNOLOGY
(COMPUTER SCIENCE AND ENGINEERING)

Under the guidance of

Mr.M.N.Gupta
Mr.Amrit Nath Thulal
Ms.Meghna Sharma
Ms.Shaveta
Ms.Priyanka

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY
(AN INSTITUTE OF R.B.E.F)
AFFILIATED TO
GURU GOBIND SINGH
INDRAPRASTHA UNIVERSITY
NEW DELHI

Table of Contents

WORLDVIEW

Introduction To Solid Modeling

The potential to model a real-world object in a computer allows the development of many attractive capabilities and opens up an unimaginable plethora of possible applications. For instance in CAD/CAM, if a solid object can be modeled in a way that adequately captures its geometry, then a variety of useful operations can be performed before the object is manufactured. We may wish to determine whether two objects interfere with each other; for example, whether a robot arm will bump into objects in its environment, or whether a cutting tool will cut only the material it is intended to remove. Finite-element analysis is applied to solid models to determine response to factors such as stress and temperature through finite element modeling. A satisfactory representation for a solid object may even make it possible to generate instructions automatically for computer-controlled machine tolls to create that object. In addition, some graphical techniques, such as modeling refractive transparency, depend on being able to determine where a beam of light enters and exits a solid object. These applications are all examples of solid modeling. The need to model objects as solids has resulted in the development of a variety of specialized ways to represent them.

1. 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 parameterized in terms of transformations and other properties. 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. 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.

2. Sweep Representations:

Sweeping an object along a trajectory through space defines a new object, called a sweep. The simplest kind of a 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 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. Rotational sweeps are defined by rotating an area about an axis. Sweeps of solids are useful in modeling the region swept out by a machine-tool cutting head or robot following a path. Sweeps whose generating area or volume changes in size, shape, or orientation as they are swept and that follow an arbitrary curved trajectory are called general sweeps. General sweeps are particularly difficult to model efficiently. Also, general sweeps do not always generate solids. In general it is difficult to apply regularized Boolean set operations to sweeps without first converting to some other representation. Even simple sweeps are not closed under regularized Boolean set operations. Despite problems of closure and calculation, however, sweeps are a natural and intuitive way to construct a variety of objects. For this reason, many solid modeling systems allow users to construct objects as sweeps but store the objects in other representations.

3. Boundary Representations:

In this scheme of modeling, a solid is described 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. Determining what constitutes a face can be particularly difficult if curved surfaces are allowed. In this representation curved faces are often approximated with polygons.

4. Spatial-partitioning representations:

In spatial-partitioning representations, a solid is decomposed into a collection of adjoining, nonintersecting 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.

4.1 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. Although cell-decomposition representation of an object is unambiguous, it is not necessarily unique. Cell decompositions are also difficult to validate, since each pair of cells must be potentially be tested for intersection. Nevertheless, cell decomposition is an important representation for use in finite element analysis.

4.2 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 called voxels (volume elements), in analogy to pixels. 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 to only 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 bio-medical applications to represent volumetric data obtained from sources such as computerized axial tomography (CAT) scans. For all 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. If the cells are cubes, then the only solids 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.

4.3 Octrees: Octrees are a hierarchical variant of spatial-occupancy enumeration, designed to address that approach's demanding storage requirements. Octrees are in turn derived from quadtrees, a 2D representation format used to encode images. The fundamental idea behind both the quadtree and octree is the divide-and-conquer power of binary subdivision. A quadtree is derived by successively subdividing a 2D plane in both dimensions to form quadrants. When a quadtree is used to represent an area in the plane, each quadrant may be full, partially full, or empty (also called black, gray and white respectively), depending on how much of the quadrant intersects the area. A partially full quadrant is recursively subdivided into sub quadrants. Subdivision continues until all quadrants are homogeneous (either full or empty) or until a predefined cutoff depth is reached. The successive subdivisions can be represented as a tree with partially full quadrants at the internal nodes and full and empty quadrants at the leaves. With the exception of a few worst cases, it can be shown that the number of nodes in a quadtree or octree representation of an object is proportional to the object's perimeter or surface respectively. Thus any operation on one of these data structures that is linear in the number of nodes it contains also executes in time proportional to the size of its perimeter or area.

5. Constructive Solid Geometry:

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. 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 tree walk, to combine nodes from the leaves on up the tree. The complexity of this task depends on the representation in which the lead objects are stored and on whether a full representation of the composite object at the tree's root must actually be produced. 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 disjointed interiors. 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 sub trees, coupled with the relatively compact form in which models are stored, have made CSG one of the dominant solid modeling representations.

6. Comparison of Representations:

- **Accuracy:** Spatial-partitioning and polygonal b-rep methods produce only approximations for many objects. In some applications, such as finding a path for a robot, this is not a drawback, as long as the approximation is computed to an adequate (often relatively coarse) resolution. The resolution needed to produce visually pleasing graphics or to calculate object interactions with sufficient accuracy, however, may be too high to be practical. The smooth shading techniques do not fix the visual artifacts caused by the all-too-obvious polygonal edges. Therefore, systems that support high-quality graphics often use CSG with non-polyhedral primitives and b-reps that allow curved surfaces. Primitive instancing can also produce high-quality pictures, but does not allow two simpler objects to be combined with Boolean set operators.
- **Domain:** 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, although often only approximation. By providing other kinds of faces and edges in addition to polygons bounded by straight lines, b-reps can be used to represent a wide class of objects. Many b-rep systems, however,

are restricted to simple surface types and topologies. For example, they may be able to encode only combinations of quadrics that are 2-manifolds.

- **Uniqueness:** Only octree and spatial-occupancy-enumeration approaches guarantee the uniqueness of a representation. There is only one way to represent an object with a specified size and position. In the case of octrees, some processing must be done to ensure that the representation is fully reduced (i.e. no gray node has all black children or all white children). Primitive instancing does not guarantee uniqueness in general: for example a sphere may be represented by both a spherical and an elliptical primitive. If the set of primitives is chosen carefully, however, uniqueness can be ensured
- **Validity:** Among all representations, b-reps stand out as being the most difficult to validate. Not only may vertex, edge, and face data structures be inconsistent, but also faces or edges may intersect. In contrast, any BSP tree represents a valid spatial set, but not necessarily a bounded solid. Only simple local syntactic checking needs to be done to validate a CSG tree (which is always bounded, if its primitives are bounded) or an octree, and no checking is needed for spatial-occupancy enumeration.
- **Closure:** Primitives created using primitive instancing cannot be combined by all, and simple sweeps are not closed under Boolean operations. Therefore, neither is typically used as an internal representation in modeling systems. Although particular b-reps may suffer from closure problems under Boolean operations (e.g. the inability to represent other than 2-manifolds), these problem cases can often be avoided.
- **Compactness and efficiency:** Representation schemes are often classified by whether they produce “evaluated” or “unevaluated” models. Unevaluated models contain information that must be further processed (or evaluated) in order to perform basic operations, such as determining an object’s boundary. With regard to the use of Boolean operations, CSG creates unevaluated models, in that each time computations are performed, we must walk the tree, evaluating the expressions. Consequently, the advantages of CSG are its compactness and the ability to record Boolean operations and changes of transformations quickly, and to undo all of these quickly since they involve only

tree-node building. Octrees and BSP trees can also be considered unevaluated models, as can a sequence of Euler operators that creates a b-rep. B-reps and spatial-occupancy enumeration, on the other hand, are often considered evaluated models insofar as any Boolean operations used to create an object have already been performed. Note that the use of these terms is relative; if the operation to be performed is determining whether a point is inside an object, for example, more work may be done evaluating a b-rep than evaluating the equivalent CSG tree. A number of efficient algorithms exist for generating pictures of objects encoded using b-reps and CSG. Although spatial-occupancy enumeration and octrees can provide only course approximations for most objects, the algorithms used to manipulate them are in general simpler than the equivalents for other representations. They have thus been used in hardware-based solid modeling systems intended for applications in which the increased speed with which Boolean set operations can be performed on them outweighs the coarseness of the resulting images.

Purpose of WorldView

WorldView has been designed to be a solid modeling software, in which stress has been laid on speed and convenience of modeling, at times at the cost of approximation, rather than the capability to store fine details. It can also store color information of the solid. WorldView can be used to model any solid and to obtain that solid's view in full color from any desired angle in 3-Dimension space. Currently it also supports a few operations on the solid like replacing some specific kind of cell with another.

When a human is asked to recall a scene he/she does not remember the fine details, only approximate positions of the objects, and an abstract view of the objects themselves. When a human sees a tree, he/she does not remember the number of leaves on that tree, their orientation or texture or their exact measurements. Humans only store a faint idea of what the tree looked like, and its approximate size. Such dramatic simplification allows a corresponding dramatic decrease in the amount of processing required to "understand" the scene. Despite the enormous processing capabilities that computers have, humans simply outclass computers when it comes to image processing. As a result, humans are still needed in possibly perilous exploration programmes. Space exploration employs robots only when it becomes impossible or prohibitively expensive/ impractical to send humans. WorldView can thus be thought of as a first step towards an attempt to adopt nature's brilliant strategy for image processing.

Hardware/Software Description

WorldView has been implemented using the JAVA programming language. The Graphical User Interface (GUI) is based entirely on JAVA Swing. The software was tested extensively on an Intel Pentium II (233 MHz) system with 192 MB SDRAM and a 2Mb VGA Video Card running Microsoft Windows ME. Any computer system capable of running the Java Virtual machine (JVM) will hence, be capable of running WorldView. The whole software is fully portable to any machine without the need for any change in code or recompilation.

System Requirements:

1. Intel Pentium II (233 MHz) and above
2. 32MB RAM
3. 2 MB Video Card
4. 2MB free hard disk space

Operating System Requirements

1. Microsoft Windows or,
2. Linux or,
3. Mac OS

WorldView Software Design

WorldView implements the Spatial Occupancy Enumeration representation for solid modeling along with a few enhancements to extend its capabilities. These enhancements over ordinary Spatial Occupancy Enumeration are:

- The solid is decomposed into entities called *blocks.* Computer memory is allocated only for blocks and not for the whole solid, thus drastically reducing space requirements.
- The resolution (number of voxels/unit volume) of each block in the solid can be different. Hence portions of the solid that need more detail can be stored at higher resolutions, and portions that do not, at lower resolutions.
- Each voxel is allocated a user-defined number of bits so that not only the presence/absence of a cell is stored but also its color information. Setting the number of bits/voxel can easily set the number of colors supported.

The WorldView Renderer implements Visible Surface Ray Tracing (ray casting) to obtain axonometric orthographic parallel projections of the desired view of the modeled solid i.e. the direction of projection and normal to the projection plane are in the same direction, but it is possible to keep the projection planes at an angle to the principal axis. Also, rays are cast at the resolution of the solid itself, and not at the resolution of the viewport. Hence, it operates at object-precision rather than image-precision.

Considerable thought and effort has been spent on carefully deciding the modules comprising Worldview. Worldview has been designed to be easily extensible, to incorporate into itself the many possible modeling representations. Future efforts to extend its capabilities will be considerably eased because of the choice of modules used. The modules are:

1. **Internal Data Structures:** This is the module, which stores the data required for completely describing the object in the given representation. This module is responsible for translating the data describing the object in the particular representation, to and from a form, which can be stored in the constrained environments of a computer. Time needed to access the stored data is a major concern while implementing this module. This module is also responsible for ascertaining the validity of the data. It is also possible to incorporate into this module, implementations of algorithms for various transformations to enhance the overall performance of the software.

2. **Renderer:** The renderer is responsible for obtaining the appropriate data from the internal data structure and painting the graphical image on a transparent canvas. The renderer only obtains the specifications of the view in 3-dimensions desired by the user, and produces the corresponding 2-dimensional planar view. The renderer is also responsible for interpreting the color-information stored in the internal data structure. It also fixes the size and background of the viewport over which the solid's required view is drawn.

WorldView also provides the capability to retrieve previously modeled solids from the back storage like hard disks or CD-ROMS.

Programming Code

Class point

Member Method	Description
public point()	Constructor for obtaining a point initialized to (0,0,0)
public point(int x, int y, int z)	Constructor for obtaining a point initialized to (x,y,z)
public void set(int x, int y, int z)	Re-initialize the point object to (x,y,z)

Class Block

Member Method	Description
public Block(int nbits, int x, int y, int z)	Create a Block which stores nbits number of bits per voxel and has dimensions (x,y,z)
Public Block(int nbits, point p)	Create a Block which stores nbits number of bits per voxel and has dimensions (p.x,p.y,p.z)
public Block(FileInputStream fin) throws IOException	Create a Block by reading it from the FileInputStream fin
public void setData(long data, int x, int y, int z)	Stores the first nbits bits of data into the Block at position (x,y,z)
public long getData(int x, int y, int z)	Returns the data stored in the Block at position (x,y,z)

Class Spocen

Member Method	Description
public void addBlock(Block toadd, point p)	Adds the Block toadd to the Spocen object at point p
public void addBlock(Block toadd, point p1, point p2)	Adds the Block toadd to the Spocen object such that it extends from point p1 to point p2
public void addBlock(Block toadd, int x1, int y1, int z1, int x2, int y2, int z2)	Add the Block toadd to the Spocen object such that it extends from (x1,y1,z1) to (x2,y2,z2)
Public void remBlock(int pos)	Removes the Block stored at position pos
public void remBlock(int x, int y, int z)	Removes the Block which contains the point (x,y,z)
public void replace(long toreplace, long with)	Replaces all occurrences of data toreplace with the data with
public void setData(long data, int x, int y, int z)	Stores the first nbits bits in data at the position (x,y,z) only if a Block exists there

public long getData(int x, int y, int z)	Returns the data stored at the position (x,y,z). If no Block exists there, then returns 0
public void writeToFile(File fname)	Stores all the contents of the data in the file specified by fname
public void readFromFile(File fname)	Retrieves and stores all the data of the Spocen object stored in the file fname
public point getDimension()	Returns the maximum extent of the Spocen object as a point object.
public void setDesc(String desc)	Sets the descriptive string for the Spocen object to desc
public String getDesc()	Returns the descriptive string for the Spocen object
public int getNbits()	Returns the maximum number of bits nbits stored in any Block of the Spocen object
public int getCount()	Returns the number of Blocks that have been added to the Spocen object
public String getBlockDesc(int pos)	Returns the descriptive string for the Block stored at position pos
public point[] getPoints(int pos)	Returns a point object array of length 2 storing the two points determining the extent of the Block stored at position pos
public point getBlockDim(int pos)	Returns the dimensions of the Block stored at position pos as a point object.

Important Code Snippets

Class Block

public void setData(long data, int posx, int posy, int posz)

```
{
	long temp=((y*posz+posy)*x+posx)*nbits;
	long stored=data;
	long temp1=arr[(int)((temp/dsize)/len)][((int)temp/dsize)%len];

	long filter=0;
	for(int i=0; i<nbits; i++)
	{
		filter=filter<<1; filter++;
	}

	filter=filter<<temp%dsize;
	data=data<<temp%dsize;

	filter=~filter;
	temp1=temp1&filter;
	temp1=temp1|data;
	arr[(int)((temp/dsize)/len)][((int)temp/dsize)%len]=temp1;

	if(temp%dsize>dsize-nbits)
	{
		temp1=arr[(int) ((temp/dsize+1)/len) ][((int)(temp/dsize)+1)%len];
		filter=0;
		for(int i=0; i<nbits-dsize+temp%dsize; i++)
		{
			filter=filter<<1; filter++;
		}
		filter=~filter;
		temp1=temp1&filter;
		stored=stored>>>(dsize-temp%dsize);
		temp1=temp1|stored;
		arr[(int) ((temp/dsize+1)/len) ][((int)(temp/dsize)+1)%len]=temp1;
	 }
	}

	public long getData(int posx, int posy, int posz)
	{
		long temp=((y*posz+posy)*x+posx)*nbits;
		long temp1=arr[(int)((long)temp/dsize)/len][((int)temp/dsize)%len];
		long toret;

		long filter=0;
		for(int i=0; i<nbits; i++)
		{
```

```
                filter=filter<<1; filter++;
            }

            filter=filter<<temp%dsize;
            toret=temp1&filter;
            toret=toret>>>temp%dsize;

            if(temp%dsize>dsize-nbits)
            {
            temp1=arr[(int) ((((long)temp/dsize)+1)/len) ][((int)(temp/dsize)+1)%len];
                filter=0;
                for(int i=0; i<nbits-dsize+temp%dsize; i++)
                {
                    filter=filter<<1; filter++;
                }
                temp1=temp1&filter;
                temp1=temp1<<(dsize-temp%dsize);
                toret=toret|temp1;
            }

            return toret;
     }
```

void MemAlloc(long space) //Allocate space bits to arr in rows of len

```
    {
            try
            {
                arr=new long[(int)Math.ceil(((double)space/dsize)/len)][len];
                System.out.println("Allocated Memory : "+(space*8/dsize)+" bytes
in rows of "+len);
            }
            catch(OutOfMemoryError e)
            {
                len=len/2;
                if(len>Math.sqrt(space)) MemAlloc(space);
                else System.out.println("Failed in allocating memory");
            }
    }
```

Class Spocen

```
    public void addBlock(Block toadd, point p1, point p2)
    {
            if(count==blocks.length-1)
            {
                Block temp[]=new Block[blocks.length+blocksinthing];
                point temparr[][]=new point[parr.length+blocksinthing][2];
                double tempscal[][]=new double[scale.length+blocksinthing][3];

                for(int i=0; i<count; i++)
```

```
                        {
                                temp[i]=blocks[i];
                                temparr[i][0]=parr[i][0];       temparr[i][1]=parr[i][1];
                                tempscal[i][0]=scale[i][0];
                                tempscal[i][1]=scale[i][1];
                                tempscal[i][2]=scale[i][2];
                        }
                        blocks=temp;  parr=temparr; scale=tempscal;
                }
                blocks[count]=toadd;
                if(nbits<toadd.nbits) nbits=toadd.nbits;
                parr[count][0]=new point(p1.x, p1.y, p1.z); parr[count][1]=new point(p2.x,
p2.y, p2.z);
                if(parr[count][0].x>x) x=parr[count][0].x;
                if(parr[count][0].y>y) y=parr[count][0].y;
                if(parr[count][0].z>z) z=parr[count][0].z;
                if(parr[count][1].x>x) x=parr[count][1].x;
                if(parr[count][1].y>y) y=parr[count][1].y;
                if(parr[count][1].z>z) z=parr[count][1].z;
                scale[count][0]=((double)(toadd.x)/(p2.x-p1.x+1));
                scale[count][1]=((double)(toadd.y)/(p2.y-p1.y+1));
                scale[count][2]=((double)(toadd.z)/(p2.z-p1.z+1));
                if(scale[count][0]<0) scale[count][0]*=-1;
                if(scale[count][1]<0) scale[count][1]*=-1;
                if(scale[count][2]<0) scale[count][2]*=-1;
                count++;

        }
}
```

public void setData(long data, int x, int y, int z)

```
        {
                for(int i=0; i<count; i++)
                {

        if((parr[i][0].x<=x&&parr[i][1].x>=x)|(parr[i][1].x<=x&&parr[i][0].x>=x))

        if((parr[i][0].y<=y&&parr[i][1].y>=y)|(parr[i][1].y<=y&&parr[i][0].y>=y))

        if((parr[i][0].z<=z&&parr[i][1].z>=z)|(parr[i][1].z<=z&&parr[i][0].z>=z))
                {
                        blocks[i].setData(data, (int)scale[i][0]*(x-parr[i][0].x),
(int)scale[i][1]*(y-parr[i][0].y), (int)scale[i][2]*(z-parr[i][0].z) );
                        return;
                }
         }
```

public long getData(int x, int y, int z)

```
        {
                for(int i=0; i<count; i++)
                {

        if((parr[i][0].x<=x&&parr[i][1].x>=x)|(parr[i][1].x<=x&&parr[i][0].x>=x))

        if((parr[i][0].y<=y&&parr[i][1].y>=y)|(parr[i][1].y<=y&&parr[i][0].y>=y))

        if((parr[i][0].z<=z&&parr[i][1].z>=z)|(parr[i][1].z<=z&&parr[i][0].z>=z))
                {       return blocks[i].getData( (int)scale[i][0]*(x-parr[i][0].x),
(int)scale[i][1]*(y-parr[i][0].y), (int)scale[i][2]*(z-parr[i][0].z) );
                }
         }
         return 0;
        }
```

Class Renderer

```
        void showview()
        {
         pntr.setColor(Color.WHITE);
         pntr.fillRect( 0, 0, view.getWidth(), view.getHeight() );
         ldata=0;
         for(int plx=-1*range; plx<range; plx++)
         {
          for(int ply=-1*range; ply<range; ply++)
          {
           for(int plz=-1*range; plz<range; plz++)
           {
            if(l*plx+m*ply+n*plz==0) //if plx,ply,plz is on the plane passing through origin
            {
             for(int r=((int)Math.ceil(dist)); r>=0; r--)
             {
              if(thethng.getData((int) (r*l+plx), (int) (r*m+ply), (int) (r*n+plz))!=0)
              {
               //Write data at appropriate position
              drawData(thethng.getData((int) (r*l+plx), (int)(r*m+ply), (int)(r*n+plz)), plx,
ply, plz);
              break;
              }
             }
            }
           }
          }
        }
        nowadd();
       }

        Color translateColor(long data)
        {
```

```
 if(nbits%4!=0) return Color.BLACK;
 int r, g, b, a, filter=0;
 for(int i=0; i<nbits/4; i++)
 {
        filter=filter<<1; filter++;
 }
 float max=filter;
 b=(int) data&filter;
 filter=filter<<(nbits/4);       g=(int) data&filter;    g=g>>>(nbits/4);
 filter=filter<<(nbits/4);       r=(int) data&filter;    r=r>>>(nbits/2);
 filter=filter<<(nbits/4);       a=(int) data&filter;    a=a>>>(nbits*3/4);
 return new Color(r/max, g/max, b/max, 1.0f-a/max);
}
```

IMAGES

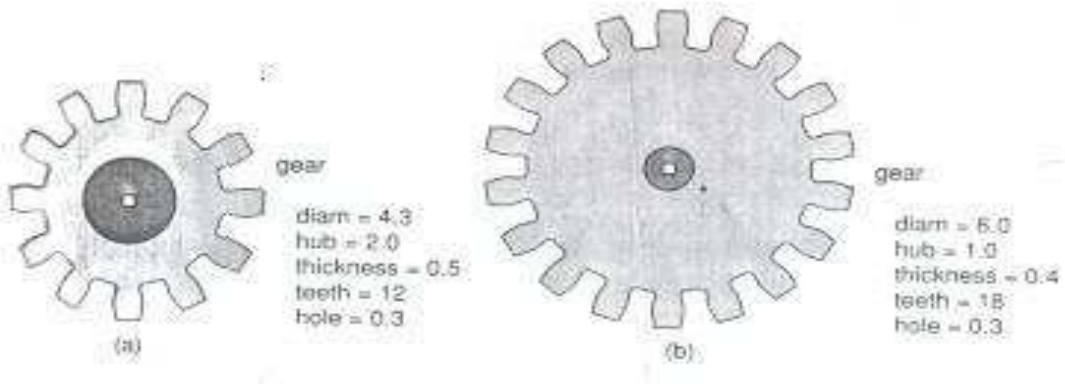


Figure: Primitive Instancing

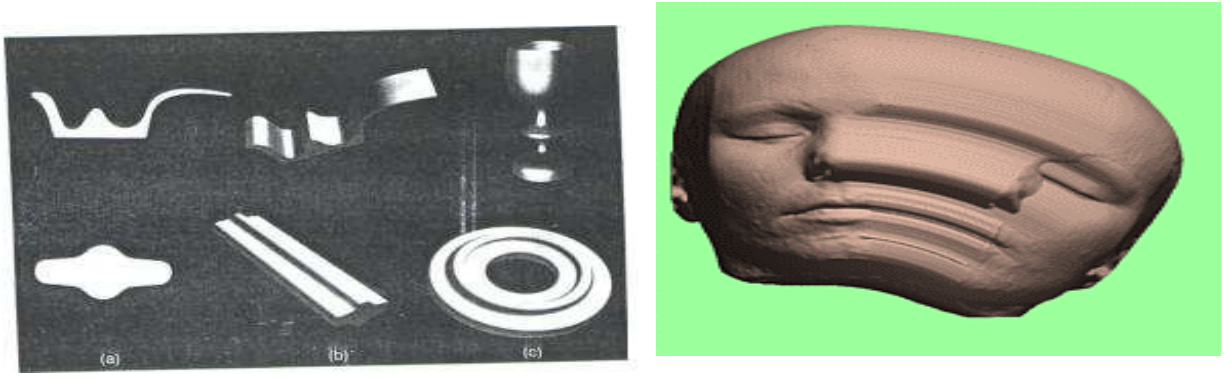


Figure: Sweep Representations

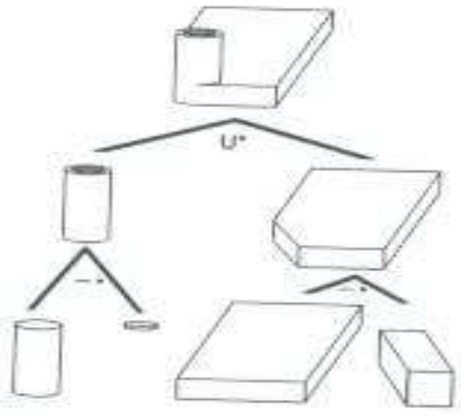


Figure: Constructive Solid Geometry (CSG)

SPATIAL-PARTITIONING REPRESENTATIONS

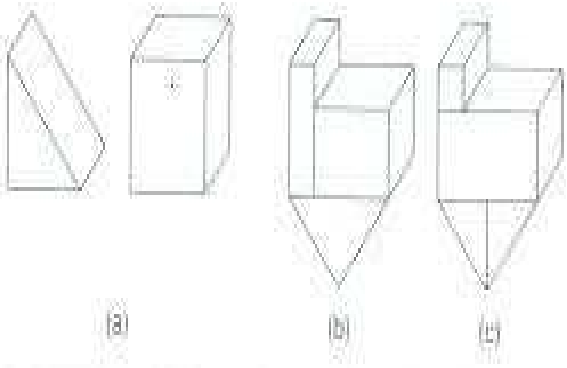


Figure: Cell Decomposition

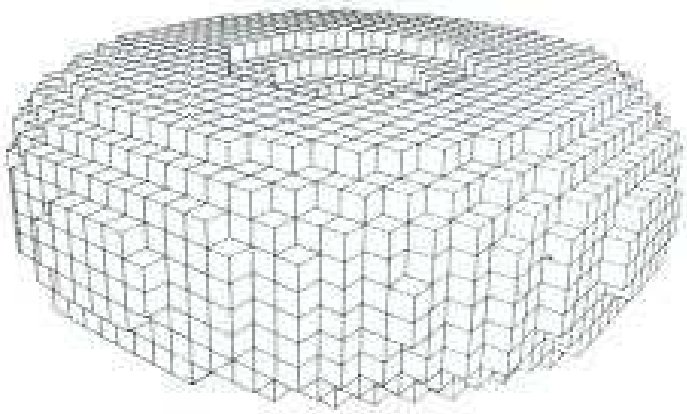
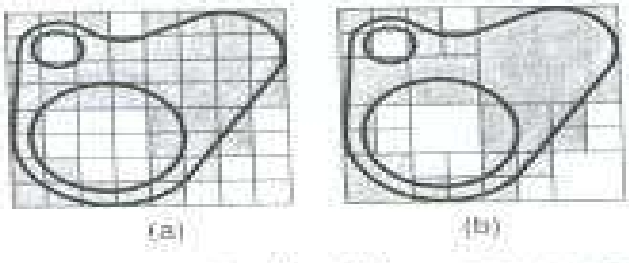


Figure: Spatial-Occupancy Enumeration representation of a torus



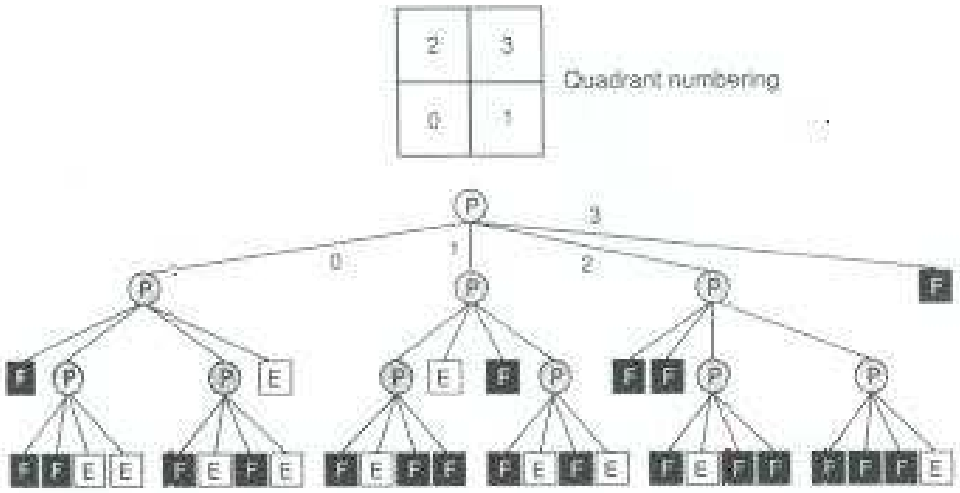
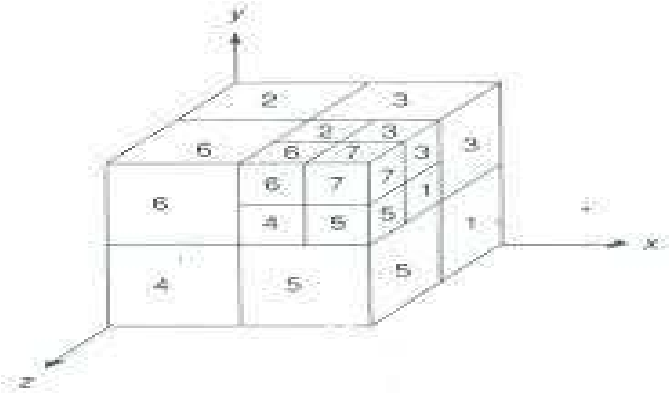


Figure: Quadtree Representation



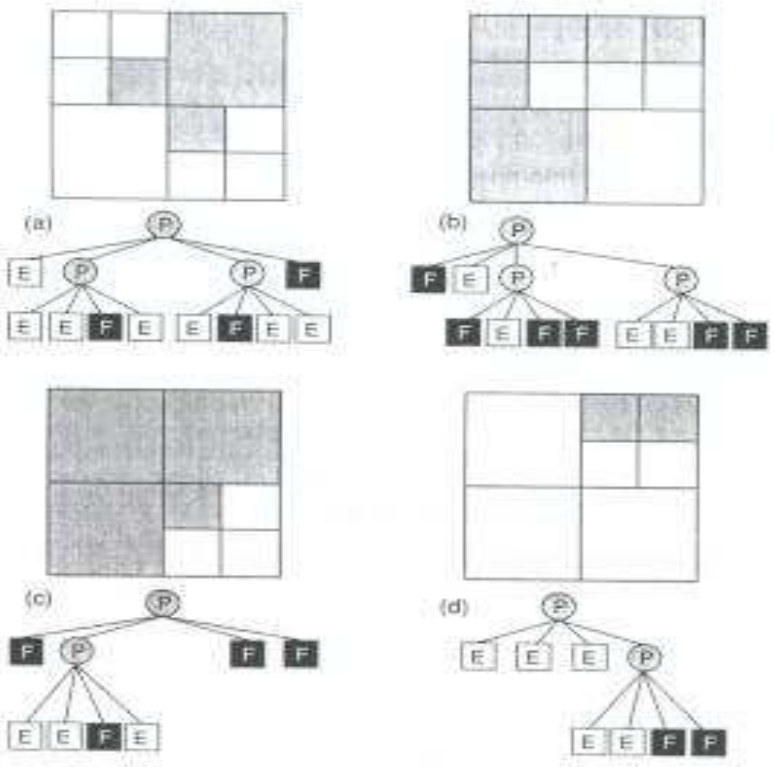


Figure : Octree Representation

CONCLUSION

WorldView was successfully tested on many machines, and was found to be reasonably fast. JAVA lended itself gracefully to the task, which comprised of intensive computation and innumerable memory accesses. The Graphical User Interface (GUI) provided by Swing, also, was easy to use and looked beautiful. The portability provided by JAVA also proved to be very useful as the same software ran on all machines.

FUTURE SCOPE OF THE PROJECT

WorldView has tremendous future scope.

Possible future extensions are:

- Modeler: The modeler will be used by the user, to interactively create WorldView solid objects using a sleek Graphical User Interface (GUI).
- 3-Dimensional Games: Contrary to normal Spatial Occupancy Enumeration, WorldView can be configured to use amazingly small amounts of memory if the scene to be rendered is chosen appropriately. It can thus be used a 3D engine for games on mobile devices.
- Path Finder: An algorithm may be implemented to find a path between any two points in the virtual world modeled using WorldView.
- Scene Analysis and Computer Vision: The software will allow the computer to recognize and reconstruct 3D models of a scene from several of its 2D images.
- Object Recognition: A modeled 3D object may be compared to a stored library of solids and be matched with them in order to recognize that object.
- Finite Element Analysis: With appropriately designed internal data structures, the analysis of any solid can be done using the finite element method.

References/Bibliography

1. Computer Graphics: principles & practice, Second Edition in C, by James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes. Pearson Education ISBN 81-7808-038-9
2. Computer Graphics, C Version (2nd Edition) by Donald Hearn and M. Pauline Baker. Prentice Hall ISBN 0135309247.
3. The Complete Reference Java 2, Fourth Edition, by Herbert Schildt. Tata McGraw-Hill Publishing, ISBN 0-07-043592-8.
4. Graphic Java 2, Volume 2, by David Geary. Prentice Hall Publications ISBN 0130796670.
5. Calculus 9th Edition by George B. Thomas, Jr. and Ross L. Finney with Maurice D. Weir. Pearson Education ISBN 81-7808-160-1
6. Advances Engineering Mathematics, by Dr. A.B. Mathur and V.P. Jaggi. Khanna Publishers.

APPENDIX: Reasons for using JAVA

2.1. Object-Oriented

Java is an *object-oriented* programming language. In an object-oriented system, a *class* is a collection of data and methods that operate on that data. Taken together, the data and methods describe the state and behavior of an *object*. Classes are arranged in a hierarchy, so that a subclass can inherit behavior from its super class. A class hierarchy always has a root class; this is a class with very general behavior. Java comes with an extensive set of classes, arranged in *packages* that can be used in programs. For example, Java provides classes that create graphical user interface components (the java.awt package), classes that handle input and output (the java.io package), and classes that support networking functionality (the java.net package). The Object class (in the java.lang package) serves as the root of the Java class hierarchy.

2.2. Interpreted

Java is an interpreted language: the Java compiler generates byte-codes for the Java Virtual Machine (JVM), rather than native machine code. To actually run a Java program, the Java interpreter executes the compiled byte-codes. Because Java byte-codes are platform-independent, Java programs can run on any platform that the JVM (the interpreter and run-time system) has been ported to. In an interpreted environment, the standard "link" phase of program development pretty much vanishes. If Java has a link phase at all, it is only the process of loading new classes into the environment, which is an incremental, lightweight process that occurs at run-time. This is in contrast with the slower and more cumbersome compile-link-run cycle of languages like C and C++.

2.3. Architecture Neutral and Portable

Because Java programs are compiled to an *architecture neutral* byte-code format, a Java application can run on any system, as long as that system implements the Java Virtual Machine. As an application developer in today's software market, development of applications that can run on PCs, Macs, and UNIX workstations is extremely desirable. With multiple flavors of UNIX, Windows 95, and Windows NT

on the PC, and the new PowerPC Macintosh, it is becoming increasingly difficult to produce software for all of the possible platforms. Application written in Java, however, can run on all platforms. While it is technically possible to write non-portable programs in Java, it is relatively easy to avoid the few platform-dependencies that are exposed by the Java API and write truly portable or "pure" Java programs. Sun's new "100% Pure Java" program helps developers ensure (and certify) that their code is portable. Programmers need only to make simple efforts to avoid non-portable pitfalls in order to live up to Sun's trademarked motto "Write Once, Run Anywhere."

2.4. Dynamic and Distributed

Java is a *dynamic* language. Any Java class can be loaded into a running Java interpreter at any time. These dynamically loaded classes can then be dynamically instantiated. Native code libraries can also be dynamically loaded. Java is also called a *distributed* language. This means, simply, that it provides a lot of high-level support for networking. For example, the URL class and OA related classes in the java.net package make it almost as easy to read a remote file or resource as it is to read a local file. Similarly, in Java 1.1, the Remote Method Invocation (RMI) API allows a Java program to invoke methods of remote Java objects, as if they were local objects. (Java also provides traditional lower-level networking support, including data grams and stream-based connections through sockets.)

2.5. Simple

Java is a *simple* language. The Java designers were trying to create a language that a programmer could learn quickly, so the number of language constructs has been kept relatively small. In order to keep the language both small and familiar, the Java designers removed a number of features available in C and C++. These features are mostly ones that led to poor programming practices or were rarely used. For example, Java does not support the goto statement; instead, it provides labelled break and continue statements and exception handling. Java does not use header files and it eliminates the C preprocessor. Because Java is object-oriented, C constructs like struct and union have been removed. Java also eliminates the operator overloading and multiple inheritance features of C++.

Perhaps the most important simplification, however, is that Java does not use pointers. Pointers are one of the most bug-prone aspects of C and C++ programming. Since Java does not have structures, and arrays and strings are objects, there's no need for pointers. Java automatically handles the referencing and dereferencing of objects. Java also implements automatic garbage collection, so the programmer doesn't have to worry about memory management issues. All of this frees the programmer from having to worry about dangling pointers, invalid pointer references, and memory leaks.

2.6. Robust

Java has been designed for writing highly reliable or *robust* software. Java certainly doesn't eliminate the need for software quality assurance; it's still quite possible to write buggy software in Java. However, Java does eliminate certain types of programming errors, which makes it considerably easier to write reliable software.

Java is a strongly typed language, which allows for extensive compile-time checking for potential type-mismatch problems. Java is more strongly typed than C++, which inherits a number of compile-time laxities from C, especially in the area of function declarations. Java requires explicit method declarations; it does not support C-style implicit declarations. These stringent requirements ensure that the compiler can catch method invocation errors, which leads to more reliable programs.

One of the things that make Java simple is its lack of pointers and pointer arithmetic. This feature also increases the robustness of Java programs by abolishing an entire class of pointer-related bugs. Similarly, all accesses to arrays and strings are checked at run-time to ensure that they are in bounds, eliminating the possibility of overwriting memory and corrupting data. Casts of objects from one type to another are also checked at run-time to ensure that they are legal. Finally, and very importantly, Java's automatic garbage collection prevents memory leaks and other pernicious bugs related to memory allocation and deallocation. Exception handling is another feature in Java that makes for more robust programs.

2.7. Secure

One of the most highly touted aspects of Java is that it's a *secure* language. This is especially important because of the distributed nature of Java. Java was designed with security in mind, and provides several layers of security controls that protect against malicious code, and allow users to comfortably run untrusted programs such as applets.

At the lowest level, security goes hand-in-hand with robustness. Java programs cannot forge pointers to memory, or overflow arrays, or read memory outside of the bounds of an array or string. These features are one of Java's main defenses against malicious code. By totally disallowing any direct access to memory, an entire huge, messy class of security attacks is ruled out.

The second line of defense against malicious code is the byte-code verification process that the Java interpreter performs on any untrusted code it loads. These verification steps ensure that the code is well formed - that it doesn't overflow or underflow the stack or contain illegal byte-codes, for example. If the byte-code verification step was skipped, inadvertently corrupted or maliciously crafted byte-codes might be able to take advantage of implementation weaknesses in a Java interpreter.

Another layer of security protection is commonly referred to as the "sandbox model": untrusted code is placed in a "sandbox," where it can play safely, without doing any damage to the "real world," or full Java environment. When an applet, or other untrusted code, is running in the sandbox, there are a number of restrictions on what it can do. The most obvious of these restrictions is that it has no access whatsoever to the local file system. There are a number of other restrictions in the sandbox as well. These restrictions are enforced by a Security Manager class. The model works because all of the core Java classes that perform sensitive operations, such as file system access, first ask permission of the currently installed Security Manager. If the call is being made, directly or indirectly, by untrusted code, the security manager throws an exception, and the operation is not permitted.

Finally, in Java 1.1, there is another possible solution to the problem of security. By attaching a digital signature to Java code, the origin of that code can be established in a cryptographically secure and unforgeable way.

2.8. High-Performance

Java is an interpreted language, so it is never going to be as fast as a compiled language like C. Java 1.0 was said to be about 20 times slower than C. Java 1.1 is nearly twice as fast as Java 1.0, however, so it might be reasonable to say that compiled C code runs ten times as fast as interpreted Java byte-codes. But this speed is more than adequate to run interactive, GUI and network-based applications, where the application is often idle, waiting for the user to do something, or waiting for data from the network. Furthermore, the speed-critical sections of the Java run-time environment, that do things like string concatenation and comparison, are implemented with efficient native code.

As a further performance boost, many Java interpreters now include "just in time" compilers that can translate Java byte-codes into machine code for a particular CPU at run-time. The Java byte-code format was designed with these "just in time" compilers in mind, so the process of generating machine code is fairly efficient and it produces reasonably good code. In fact, Sun claims that the performance of byte-codes converted to machine code is nearly as good as native C or C++.

2.9. Multithreaded

Java is a *multithreaded* language; it provides support for multiple threads of execution (sometimes called lightweight processes) that can handle different tasks. An important benefit of multithreading is that it improves the interactive performance of graphical applications for the user.

Java makes programming with threads much easier, by providing built-in language support for threads. The java.lang package provides a Thread class that supports methods to start and stop threads and set thread priorities, among other things. The Java language syntax also supports threads directly with the synchronized keyword. This keyword makes it extremely easy to mark sections of code or entire methods that should only be run by a single thread at a time.