UNIT V FRACTALS

Computers are good at repetition. In addition, the high precision with which modern computers can do calculations allows an algorithm to take closer look at an object, to get greater levels of details.

Computer graphics can produce pictures of things that do not even exist in nature or perhaps could never exist. We will study the inherent finiteness of any computer generated picture. It has finite resolution and finite size, and it must be made in finite amount of time. The pictures we make can only be approximations, and the observer of such a picture uses it just as a hint of what the underlying object really looks like.

5.1 FRACTALS AND SELF-SIMILARITY

Many of the curves and pictures have a particularly important property called **self-similar**. This means that they appear the same at every scale: No matter how much one enlarges a picture of the curve, it has the same level of detail.

Some curves are **exactly self-similar**, whereby if a region is enlarged the enlargement looks exactly like the original.

Other curves are **statistically self-similar**, such that the wiggles and irregularities in the curve are the same "on the average", no matter how many times the picture is enlarged. Example: Coastline.

5.1.1 Successive Refinement of Curves

A complex curve can be fashioned recursively by repeatedly "refining" a simple curve. The simplest example is the Koch curve, discovered in1904 by the Swedish mathematician Helge von Koch. The curve produces an infinitely long line within a region of finite area.

Successive generations of the Koch curve are denoted K0, K1, K2….The zeroth generation shape K0 is a horizontal line of length unity.

Two generations of the Koch curve

To create K1 , divide the line K0 into three equal parts and replace the middle section with a triangular bump having sides of length 1/3.

CS2401 COMPUTER GRAPHICS UNIT V

2

The total length of the line is 4/3. The second order curve K2, is formed by building a bump on each of the four line segments of K1.

To form Kn+1 from Kn:

Subdivide each segment of Kn into three equal parts and replace the middle part with a bump in the shape of an equilateral triangle.

In this process each segment is increased in length by a factor of 4/3, so the total length of the curve is 4/3 larger than that of the previous generation. Thus Ki has total length of (4/3)i , which increases as i increases. As i tends to infinity, the length of the curve becomes infinite.

The first few generations of the Koch snowflake

The Koch snowflake of the above figure is formed out of three Koch curves joined together. The perimeter of the ith generations shape Si is three times length of a Koch curve and so is 3(4/3)i , which grows forever as i increases. But the area inside the Koch snowflake grows quite slowly. So the edge of the Koch snowflake gets rougher and rougher and longer and longer, but the area remains bounded.

Koch snowflake s3, s4 and s5

The Koch curve Kn is self-similar in the following ways: Place a small window about some portion of Kn, and observe its ragged shape. Choose a window a billion times smaller and observe its shape. If n is very large, the curve appears to be have same shape and roughness. Even if the portion is enlarged another billion times, the shape would be the same.

5.1.2 Drawing Koch Curves and Snowflakes

The Koch curves can be viewed in another way: Each generation consists of four versions of the previous generations. For instance K2 consists of four versions of K1 tied end to end with certain angles between them.

CS2401 COMPUTER GRAPHICS UNIT V

3

We call n the order of the curve Kn, and we say the order –n

Koch curve consists of four versions of the order (n-1) Koch curve.To

draw K2 we draw a smaller version of K1 , then turn left 60 , draw K1

again, turn right 120 , draw K1 a third time. For snowflake this routine is

performed just three times, with a 120 turn in between.

The recursive method for drawing any order Koch curve is given in the following pseudocode:

```
To draw Kn:
if ( n equals 0 ) Draw a straight line;
else {
Draw Kn-1;
Turn left 60 ;
Draw Kn-1;
Turn right 120 ;
Draw Kn-1;
Turn left 60 ;
Draw Kn-1;
}
```

Drawing a Koch Curve

```
Void drawKoch (double dir, double len, int n)
{
// Koch to order n the line of length len
// from CP in the direction dir
double dirRad= 0.0174533 * dir; // in radians
if (n ==0)
lineRel(len * cos(dirRad), len * sin(dirRad));
else {
n--; //reduce the order
len /=3; //and the length
drawKoch(dir, len, n);
dir +=60;
drawKoch(dir, len, n);
dir -=120;
drawKoch(dir, len, n);
dir +=60;
drawKoch(dir, len, n);
}
```

}

The routine drawKoch() draws Kn on the basis of a parent

line of length len that extends from the current position in the direction

CS2401 COMPUTER GRAPHICS UNIT V

4

dir. To keep track of the direction of each child generation, the parameter dir is passed to subsequent calls of Koch().

5.3 Creating An Image By Means of Iterative Function Systems

Another way to approach infinity is to apply a transformation to a picture again and again and examine the results. This technique also provides an another method to create fractal shapes.

5.3.1 An Experimental Copier

We take an initial image I0 and put it through a special photocopier that produces a new image I1. I1 is not a copy of I0 rather it is a superposition of several reduced versions of I0. We then take I1 and feed it back into the copier again, to produce image I2. This process is repeated , obtaining a sequence of images I0, I1, I2,… called the **orbit of I0**.

Making new copies from old

In general this copier will have N lenses, each of which perform an affine mapping and then adds its image to the output. The collection of the N affine transformations is called an "iterated function system".

An iterated function system is a collection of N affine transformations Ti, for i=1,2,…N.

5.3.2 Underlying Theory of the Copying Process

Each lens in the copier builds an image by transforming every point in the input image and drawing it on the output image. A black and white image I can be described simply as the set of its black points:

I = set of all black points = { (x,y) such that (x,y) is colored black }

I is the input image to the copier. Then the ith lens characterized by transformation Ti, builds a new set of points we denote as Ti(I) and adds them to the image being produced at the current iteration. Each added set Ti(I) is the set of all transformed points I:

Ti(I) = { (x',y') such that (x',y') = Ti(P) for some point P in I }

CS2401 COMPUTER GRAPHICS UNIT V

5

Upon superposing the three transformed images, we obtain the output image as the union of the outputs from the three lenses:

Output image = T1(I) U T2(I) U T3(I)

The overall mapping from input image to output image as W(.). It maps one set of points – one image – into another and is given by:

W(.)=T1(.) U T2(.) U T3(.)

For instance the copy of the first image I0 is the set W(I0).

Each affine map reduces the size of its image at least slightly, the orbit converge to a unique image called the **attractor** of the IFS. We denote the attractor by the set A, some of its important properties are:

1. The attractor set A is a fixed point of the mapping W(.), which we write as W(A)=A. That is putting A through the copier again produces exactly the same image A.

The iterates have already converged to the set A, so iterating once more makes no difference.

2. Starting with any input image B and iterating the copying process enough times, we find that the orbit of images always converges to the same A.

If Ik = W (k)(B) is the kth iterate of image B, then as k goes to infinity Ik becomes indistinguishable from the attractor A.

5.3.3 Drawing the kth Iterate

We use graphics to display each of the iterates along the orbit. The initial image I0 can be set, but two choices are particularly suited to the tools developed: I0 is a polyline. Then successive iterates are collections of polylines. I0 is a single point. Then successive iterates are collections of points.

Using a polyline for I0 has the advantage that you can see how each polyline is reduced in size in each successive iterate. But more memory and time are required to draw each polyline and finally each polyline is so reduced as to be indistinguishable from a point.

Using a single point for I0 causes each iterate to be a set of points, so it is straight forward to store these in a list. Then if IFS consists of N affine maps, the first iterate I1 consists of N points, image I2 consists of N2 points, I3 consists of N3 points, etc.

Copier Operation pseudocode(recursive version)

```
void superCopier( RealPolyArray pts, int k)
{ //Draw kth iterate of input point list pts for the IFS
```

int i;

RealPolyArray newpts; //reserve space for new list

if(k==0) drawPoints(pts);

else for(i=1; i<=N; i++) //apply each affine

{

CS2401 COMPUTER GRAPHICS UNIT V

6

newpts.num= N * pts.num; //the list size grows fast

for(j=0; j<newpts.num; j++) //transforms the jth point

transform(affines[i], pts.pt[j], newpts.pt[j]);

superCopier(newpts, k – 1);

}

}

If k=0 it draws the points in the list

If k>0 it applies each of the affine maps Ti, in turn, to all of the points, creating a new list of points, newpts, and then calls superCopier(newpts, k – 1);

To implement the algorithm we assume that the affine maps are stored in the global array Affine affines[N].

Drawbacks

Inefficient

Huge amount of memory is required.

5.3.4 The Chaos Game

The Chaos Game is a nonrecursive way to produce a picture of the attractor of an IFS.

The process for drawing the Sierpinski gasket

Set corners of triangle :p[0]=(0,0), p[1]=(1,0), p[2]=(.5,1)

Set P to one of these, chosen randomly;

do {

Draw a dot at P;

Choose one of the 3 points at random;

Set newPt to the midpoint of P and the chosen point;

Set P= newPt;

} while(!bored);

A point P is transformed to the midpoint of itself and one of

the three fixed points: p[0], p[1] or p[2]. The new point is then drawn as a

dot and the process repeats. The picture slowly fills in as the sequence of

dots is drawn.

The key is that forming a midpoint based on P is in fact

applying an affine transformation. That is

P =

2

1

(P + p[…]) (find the midpoint of P and p[..])

Can be written as

P =

2

1

0

0

2

1

P +

2

1

p[..]

So that P is subjected to the affine map, and then the transformed

version is written back into P. The offset for this map depends on which

point p[i[is chosen.

CS2401 COMPUTER GRAPHICS UNIT V

7

Drawing the Sierpinski gasket

One of the three affine maps is chosen at random each time, and the previous point P is subjected to it. The new point is drawn and becomes the next point to be transformed.

Also listed with each map is the probability pri that the map is chosen at each iteration.

Starting with a point P0, the sequence of iterations through this system produces a sequence of points P0, P1,.., which we call the orbit of the system with starting point P0. This is a random orbit, different points are visited depending on the random choices made at each iteration.

The idea behind this approach is that the attractor consists of all points that are reachable by applying a long sequence of affines in the IFS. The randomness is invoked to ensure that the system is fully exercised, that every combination of affine maps is used somewhere in the process.

Pseudocode for playing the Chaos Game

```
void chaosGame(Affine aff[], double pr[], int N)
{
RealPoint P = { 0,0 ,0,0}; //set some initial point
int index;
do {
index = chooseAffine(pr , N); // choose the next affine
P = transform(aff[ondex], P);
drawRealDot(P); // draw the dot
} while (!bored);
}
```

The function chaosGame() plays the chaos game. The function draws the attractor of the IFS whose N transforms are stored in the array aff[]. The probabilities to be used are stored in an array pr[]. At each iteration one of the N affine maps is chosen randomly by the function chooseAffine() and is used to transform the previous point into the next point.

Adding Color

The pictures formed by playing the Chaos Game are bilevel, black dots on a white background. It is easy to extend the method so

CS2401 COMPUTER GRAPHICS UNIT V

8

that it draws gray scale and color images of objects. The image is viewed as a collection of pixels and at each iteration the transformed point lands in one of the pixels. A counter is kept for each pixel and at the completion of the game the number of times each pixel has been visited is converted into a color according to some mapping.

5.3.4 Finding the IFS; Fractal Image Compression

Dramatic levels of image compression provide strong motivation for finding an IFS whose attractor is the given image. A image contains million bytes of data, but it takes only hundreds or thousands of bytes to store the coefficients of the affine maps in the IFS.

Fractal Image Compression and regeneration

The original image is processed to create the list of affine maps, resulting in a greatly compressed representation of the image.

In the decompression phase the list of affine maps is used and an algorithm such as the Chaos Game reconstructs the image. This compression scheme is lossy, that is the image I' that is generated by the game during decompression is not a perfect replica of the original image I.

5.4 THE MANDELBROT SET

Graphics provides a powerful tool for studying a fascinating collection of sets that are the most complicated objects in mathematics.

Julia and Mandelbrot sets arise from a branch of analysis known as iteration theory, which asks what happens when one iterates a function endlessly. Mandelbrot used computer graphics to perform experiments.

5.4.1 Mandelbrot Sets and Iterated Function Systems

A view of the Mandelbrot set is shown in the below figure. It is the black inner portion, which appears to consist of a cardoid along with a number of wartlike circles glued to it.

Its border is complicated and this complexity can be explored by zooming in on a portion of the border and computing a close up view. Each point in the figure is shaded or colored according to the outcome of an experiment run on an IFS.

CS2401 COMPUTER GRAPHICS UNIT V

9

The Mandelbrot set

The Iterated function systems for Julia and Mandelbrot sets

The IFS uses the simple function

f(z) = z2 + c --------------------------------(1)

where c is some constant. The system produces each output by squaring its input and adding c. We assume that the process begins with the starting value s, so the system generates the sequence of values or orbit

d1= (s)2 + c

d2= ((s)2 + c)2 + c

d3= (((s)2 + c)2 + c)2 + c

d4= ((((s)2 + c)2 + c)2 + c)2 + c ------------------------------(2)

The orbit depends on two ingredients the starting point s the given value of c

Given two values of s and c how do points dk along the orbit behaves as k gets larger and larger? Specifically, does the orbit remain finite or explode. Orbits that remain finite lie in their corresponding Julia or Mandelbrot set, whereas those that explode lie outside the set.

When s and c are chosen to be complex numbers , complex arithmetic is used each time the function is applied. The Mandelbrot and Julia sets live in the complex plane – plane of complex numbers.

CS2401 COMPUTER GRAPHICS UNIT V

10

The IFS works well with both complex and real numbers.

Both s and c are complex numbers and at each iteration we square the

previous result and add c. Squaring a complex number z = x + yi yields

the new complex number:

(x + yi)2 = (x2 – y2) + (2xy)i

----------------------------------(3)

having real part equal to x2 – y2 and imaginary part equal to

2xy.

Some Notes on the Fixed Points of the System

It is useful to examine the fixed points of the system

f(.) =(.)2 + c . The behavior of the orbits depends on these fixed points

that is those complex numbers z that map into themselves, so that

z2 + c = z. This gives us the quadratic equation z2 – z + c = 0 and the fixed

points of the system are the two solutions of this equation, given by p+, p- = *c*

4

1

2

1

--------------------------------(4)

If an orbit reaches a fixed point, p its gets trapped there
forever. The fixed point can be characterized as **attracting** or **repelling**.
If an orbit flies close to a fixed point p, the next point along the orbit will
be forced
closer to p if p is an attracting fixed point
farther away from p if p is a repelling a fixed point.
If an orbit gets close to an attracting fixed point, it is sucked
into the point. In contrast, a repelling fixed point keeps the orbit away
from it.

5.4.2 Defining the Mandelbrot Set

The Mandelbrot set considers different values of c, always
using the starting point s =0. For each value of c, the set reports on the
nature of the orbit of 0, whose first few values are as follows:
orbit of 0: 0, c, c2+c, (c2+c)2+c, ((c2+c)2+c)2 +c,……..
For each complex number c, either the orbit is **finite** so that
how far along the orbit one goes, the values remain finite or the orbit
explodes that is the values get larger without limit. The Mandelbrot set
denoted by M, contains just those values of c that result in finite orbits:
The point c is in M if 0 has a finite orbit.
The point c is not in M if the orbit of 0 explodes.

Definition:

The Mandelbrot set M is the set of all complex numbers c
that produce a finite orbit of 0.
If c is chosen outside of M, the resulting orbit explodes. If c
is chosen just beyond the border of M, the orbit usually thrashes around

the plane and goes to infinity.
If the value of c is chosen inside M, the orbit can do a variety
of things. For some c's it goes immediately to a fixed point or spirals into
such a point.
CS2401 COMPUTER GRAPHICS UNIT V
11

5.4.3 Computing whether Point c is in the Mandelbrot Set

A routine is needed to determine whether or not a given
complex number c lies in M. With a starting point of s=0, the routine
must examine the size of the numbers dk along the orbit. As k increases
the value of *k d* either explodes(c is not in M) or does not explode(c is
in M).
A theorem from complex analysis states that if *k d* exceeds
the value of 2, then the orbit will explode at some point. The number of
iterations *k d* takes to exceed 2 is called the **dwell** of the orbit.
But if c lies in M, the orbit has an infinite dwell and we can't
know this without it iterating forever. We set an upper limit Num on the
maximum number of iterations we are willing to wait for.
A typical value is Num = 100. If *k d* has not exceeded 2 after
Num iterates, we assume that it will never and we conclude that c is in
M. The orbits for values of c just outside the boundary of M have a large
dwell and if their dwell exceeds Num, we wrongly decide that they lie
inside M. A drawing based on too small value of Num will show a
Mandelbrot set that is slightly too large.

dwell() routine

```
int dwell (double cx, double cy)
{ // return true dwell or Num, whichever is smaller
#define Num 100 // increase this for better pictures
double tmp, dx=cx, dy=cy, fsq=cx *cx + cy * cy;
for(int count=0; count<=Num && fsq <=4; count++)
{
```

tmp = dx; //save old real part

dx = dx * dx – dy * dy +cx; //new real part

dy = 2.0 * tmp * dy + cy; //new imaginary part

fsq = dx * dx + dy * dy;

}

return count; // number of iterations used

}

For a given value of c = cx + cyi, the routine returns the
number of iterations required for *k d* to exceed 2.
At each iteration, the current dk resides in the pair (dx,dy)
which is squared using eq(3) and then added to (cx,cy) to form the next
d value. The value *k d* 2 is kept in fsq and compared with 4. The dwell()
function plays a key role in drawing the Mandelbrot set.

5.4.4 Drawing the Mandelbrot Set

To display M on a raster graphics device. To do this we set
up a correspondence between each pixel on the display and a value of c,
CS2401 COMPUTER GRAPHICS UNIT V
12
and the dwell for that c value is found. A color is assigned to the pixel, depending on whether the dwell is finite or has reached its limit.

The simplest picture of the Mandelbrot set just assign black to points inside M and white to those outside. But pictures are more appealing to the eye if a range of color is associated with points outside M. Such points all have dwells less than the maximum and we assign different colors to them on the basis of dwell size.

Assigning colors according to the orbit’s dwell

The figure shows how color is assigned to a point having dwell d. For very small values of d only a dim blue component is used. As d approaches Num the red and green components are increased up to a maximum unity. This could be implemented in OpenGL using:

float v = d / (float)Num;

glColor3f(v * v, v*, v, 0.2); // red & green at level v-squared

We need to see how to associate a pixel with a specific complex value of c. A simple approach is suggested in the following figure.

Establishing a window on M and a correspondence between points and pixels.

The user specifies how large the desired image is to be on the screen that is the number of rows, **rows** the number of columns, **cols**

CS2401 COMPUTER GRAPHICS UNIT V

13

This specification determines the aspect ratio of the image

:R= cols/rows. The user also chooses a portion of the complex plane

to be displayed: a rectangular region having the same aspect ratio as

the image. To do this the user specifies the region's upper left hand

corner P and its width W. The rectangle's height is set by the required

aspect ratio. The image is displayed in the upper left corner of the

display.

To what complex value c= cx + cyi, does the center of the i,

jth pixel correspond? Combining we get

cij = *W*

cols

j

W P

cols

i

P x y

2

1

2 ,

1

------------------------(5)

for i = 0,…….,cols-1 and j=0,…..,rows-1.

The chosen region of the Mandelbrot set is drawn pixel by

pixel. For each pixel the corresponding value of c is passed to dwell(),

and the appropriate color associated with the dwell is found. The pixel
is then set to this color.

Pseudocode for drawing a region of the Mandelbrot set

```
for(j=0; j<rows; j++)
for(i=0; i<cols; i++)
{
find the corresponding c value in equation (5)
estimate the dwell of the orbit
find Color determined by estimated dwell
setPixel( j , k, Color);
}
```

A practical problem is to study close up views of the
Mandelbrot set, numbers must be stored and manipulated with great
precision.

Also when working close to the boundary of the set , you
should use a larger value of Num. The calculation times for each
image will increase as you zoom in on a region of the boundary of M.
But images of modest size can easily be created on a microcomputer
in a reasonable amount of time.

5.5 JULIA SETS

Like the Mandelbrot set, Julia sets are extremely
complicated sets of points in the complex plane. There is a different Julia
set, denoted Jc for each value of c. A closely related variation is the **filledin Julia set**, denoted by Kc, which is easier to define.

5.5.1 The Filled-In Julia Set Kc

In the IFS we set c to some fixed chosen value and examine
what happens for different starting point s. We ask how the orbit of
CS2401 COMPUTER GRAPHICS UNIT V
14
starting point s behaves. Either it explodes or it doesn't. If it is finite , we say the starting point s is in Kc, otherwise s lies outside of Kc.

Definition:

The filled-in Julia set at c, Kc, is the set of all starting points whose orbits are finite.

When studying Kc, one chooses a single value for c and considers different starting points. Kc should be always symmetrical about the origin, since the orbits of s and –s become identical after one iteration.

5.5.2 Drawing Filled-in Julia Sets

A starting point s is in Kc, depending on whether its orbit is finite or explodes, the process of drawing a filled-in Julia set is almost similar to Mandelbrot set. We choose a window in the complex plane and associate pixels with points in the window. The pixels correspond to different values of the starting point s. A single value of c is chosen and then the orbit for each pixel position is examined to see if it explodes and if so, how quickly does it explodes.

Pseudocode for drawing a region of the Filled-in Julia set

```
for(j=0; j<rows; j++)
for(i=0; i<cols; i++)
{
find the corresponding s value in equation (5)
estimate the dwell of the orbit
find Color determined by estimated dwell
setPixel( j , k, Color);
}
```

The dwell() must be passed to the starting point s as well as c. Making a high-resolution image of a Kc requires a great deal of computer time, since a complex calculation is associated with every pixel.

5.5.3 Notes on Fixed Points and Basins of Attraction

If an orbit starts close enough to an attracting fixed point, it is sucked into that point. If it starts too far away, it explodes. The set of points that are sucked in forms a so called **basin of attraction** for the fixed point p. The set is the filled-in Julia set Kc. The fixed point which lies inside the circle |z|= ½ is the attracting point.

All points inside Kc, have orbits that explode. All points inside Kc, have orbits that spiral or plunge into the attracting fixed point. If the starting point is inside Kc, then all of the points on the orbit must also be inside Kc and they produce a finite orbit. The repelling fixed point is on the boundary of Kc.

Kc for Two Simple Cases

The set Kc is simple for two values of c:

1. **c=0:** Starting at any point s, the orbit is simply s, s2,s4,…….,s2k,…,
so the orbit spirals into 0 if |s|<1 and explodes if |s|>1. Thus K0
is the set of all complex numbers lying inside the unit circle, the
CS2401 COMPUTER GRAPHICS UNIT V
15
circle of radius 1 centered at the origin.

2. **c = -2:** in this case it turns out that the filled-in Julia set consists
of all points lying on the real axis between -2 and 2.

For all other values of c, the set Kc, is complex. It has been
shown that each Kc is one of the two types:

Kc is connected or

Kc is a Cantor set

A theoretical result is that Kc is connected for precisely those
values of c that lie in the Mandelbrot set.

5.5.4 The Julia Set Jc

Julia Set Jc is for any given value of c; it is the boundary of
Kc. Kc is the set of all starting points that have finite orbits and every
point outside Kc has an exploding orbit. We say that the points just along
the boundary of Kc and "on the fence". Inside the boundary all orbits
remain finite; just outside it, all orbits goes to infinity.

Preimages and Fixed Points

If the process started instead at f(s), the image of s, then the
two orbits would be:

s, f(s), f2(s), f3(s),…. (orbit of s)

or

f(s), f2(s), f3(s), f4(s),…. (orbit of f(s))

which have the same value forever. If the orbit of s is finite,

then so is the orbit of its image f(s). All of the points in the orbit , if
considered as starting points on their own, have orbits with thew same
behavior: They all are finite or they all explode.
Any starting point whose orbit passes through s has the
same behavior as the orbit that start at s: The two orbits are identical
forever. The point “**just before**” s in the sequence is called the **preimage**
of s and is the inverse of the function f(.) = (.)2 + c. The inverse of f(.) is
z c , so we have
two preimages of z are given by *z c* ------------------(6)
To check that equation (6) is correct, note that if either
preimage is passed through (.)2 + c, the result is z. The test is illustrated
in figure(a) where the orbit of s is shown in black dots and the two
preimages of s are marked. The two orbits of these preimages “join up”
with that of s.
Each of these preimages has two preimages and each if
these has two, so there is a huge collection of orbits that join up with the
orbit of s, and thereafter committed to the same path. The tree of
preimages of s is illustrated in fig(B): s has two parent preimages, 4
grandparents, etc. Going back k generations we find that there are 2k
preimages.
CS2401 COMPUTER GRAPHICS UNIT V
16
Orbits that coincide at s
The Julia set Jc can be characterized in many ways that are
more precise than simply saying it is the “boundary of” Kc. One such
characterization that suggests an algorithm for drawing Jc is the
following:
The collection of all preimages of any point in Jc is dense in Jc.
Starting with any point z in Jc, we simply compute its two
parent preimages, their four grandparent preimages, their eight greatgrandparent
ones, etc. So we draw a dot at each such preimage, and the

display fills in with a picture of the Julia set. To say that these dots are
dense in Jc means that for every point in Jc, there is some preimage that
is close by.

Drawing the Julia set Jc

To draw Jc we need to find a point and place a dot at all of
the point's preimages. Therea re two problems with this method:

1. finding a point in Jc
2. keeping track of all the preimages

An approach known as the backward-iteration method
overcomes these obstacles and produces good result. The idea is simple:
Choose some point z in the complex plane. The point may or may not be
in Jc. Now iterate in backward direction: at each iteration choose one of
the two square roots randomly, to produce a new z value. The following
pseudocode is illustrative:

```
do {
if ( coin flip is heads z= $z c$ );
else z = $z c$ ;
draw dot at z;
} while (not bored);
```

The idea is that for any reasonable starting point iterating
backwards a few times will produce a z that is in Jc. It is as if the
backward orbit is sucked into the Julia set. Once it is in the Julia set, all
CS2401 COMPUTER GRAPHICS UNIT V
17
subsequent iterations are there, so point after point builds up inside Jc, and a picture emerges.

5.6 RANDOM FRACTALS

Fractal is the term associated with randomly generated curves and surfaces that exhibit a degree of self-similarity. These curves are used to provide "naturalistic" shapes for representing objects such as coastlines, rugged mountains, grass and fire.

5.6.1 Fractalizing a Segment

The simplest random fractal is formed by recursively roughening or fractalizing a line segment. At each step, each line segment is replaced with a "random elbow".

The figure shows this process applied to the line segment S having endpoints A and B. S is replaced by the two segments from A to C and from C to B. For a fractal curve, point C is randomly chosen along the perpendicular bisector L of S. The elbow lies randomly on one or the other side of the "parent" segment AB.

Fractalizing with a random elbow

Steps in the fractalization process

Three stages are required in the fractalization of a segment. In the first stage, the midpoint of AB is perturbed to form point C. In the second stage , each of the two segment has its midpoints perturbed to

CS2401 COMPUTER GRAPHICS UNIT V

18

form points D and E. In the third and final stage, the new points F…..I

are added.

To perform fractalization in a program

Line L passes through the midpoint M of segment S and is

perpendicular to it. Any point C along L has the parametric form:

C(t) = M + (B-A) t ----------------------------------(7)

for some values of t, where the midpoint M= (A+B)/2.

The distance of C from M is |B-A||t|, which is proportional

to both t and the length of S. So to produce a point C on the random

elbow, we let t be computed randomly. If t is positive, the elbow lies to

one side of AB; if t is negative it lies to the other side.

For most fractal curves, t is modeled as a Gaussian random

variable with a zero mean and some standard deviation. Using a mean of

zero causes, with equal probability, the elbow to lie above or below the

parent segment.

Fractalizing a Line segment

```
void fract(Point2 A, Point2 B, double stdDev)
// generate a fractal curve from A to B
```

```
double xDiff = A.x – B.x, yDiff= A.y –B.y;
Point2 C;
if(xDiff * XDiff + YDiff * yDiff < minLenSq)
cvs.lintTo(B.x, B.y);
else
{
stdDev *=factor; //scale stdDev by factor
double t=0;
// make a gaussian variate t lying between 0 and 12.0
for(int i=0; I, 12; i++)
t+= rand()/32768.0;
t= (t-6) * stdDev; //shift the mean to 0 and sc
C.x = 0.5 *(A.x +B.x) – t * (B.y – A.y);
C.y = 0.5 *(A.y +B.y) – t * (B.x – A.x);
fract(A, C, stdDev);
fract(C, B, stdDev);
}
```

The routine fract() generates curves that approximate actual
fractals. The routine recursively replaces each segment in a random
elbow with a smaller random elbow. The stopping criteria used is: When
the length of the segment is small enough, the segment is drawn using
cvs.lineTo(), where cvs is a Canvas object. The variable t is made to be
approximately Gaussian in its distribution by summing together 12
uniformly distributed random values lying between 0 and 1. The result
has a mean value of 6 and a variance of 1. The mean value is then
shifted to 0 and the variance is scaled as necessary.

The depth of recursion in fract() is controlled by the length of
the line segment.

CS2401 COMPUTER GRAPHICS UNIT V

19

5.6.2 Controlling the Spectral Density of the Fractal Curve

The fractal curve generated using the above code has a
“power spectral density” given by

S(f)= 1/f β

Where β the power of the noise process is the parameter the
user can set to control the jaggedness of the fractal noise. When β is 2,
the process is known as Brownian motion and when β is 1, the process is
called “1/f noise”. 1/f noise is self similar and is shown to be a good
model for physical process such as clouds. The fractal dimension of such
processes is:

2

5

D

In the routine fract(), the scaling factor factor by which the
standard deviation is scaled at each level based on the exponent β of the
fractal curve. Values larger than 2 leads to smoother curves and values
smaller than 2 leads to more jagged curves. The value of factor is given
by:

factor = 2 (1 – β/2)

The factor decreases as β increases.

Drawing a fractal curve(pseudocode)

```
double MinLenSq, factor; //global variables
void drawFractal (Point2 A, Point2 B)
{
double beta, StdDev;
User inputs beta, MinLenSq and the the initial StdDev
factor = pow(2.0, (1.0 – beta)/ 2.0);
cvs.moveTo(A);
fract(A, B, StdDev);
}
```

In this routine factor is computed using the C++ library
function pow(…).

One of the features of fractal curves generated by
pseudorandom –number generation is that they are repeatable. All that is
required is to use the same seed each time the curve is fractalized. A
complicated shape can be fractalized and can be stored in the database
by storing only
the polypoint that describes the original line segments
the values of minLenSq and stdDev and
the seed.
An extract replica of the fractalized curve can be regenerated
at any time using these informations.
CS2401 COMPUTER GRAPHICS UNIT V
20

5.7 INTERSECTING RAYS WITH OTHER PRIMITIVES

First the ray is transformed into the generic coordinates of the object and then the various intersection with the generic object are computed.

1) Intersecting with a Square

The generic square lies in the z=0 plane and extends from -1 to 1 in both x and y. The square can be transformed into any parallelogram positioned in space, so it is often used in scenes to provide this, flat surfaces such as walls and windows. The function hit(1) first finds where the ray hits the generic plane and then test whether this hit spot also lies within the square.

2) Intersecting with a Tapered Cylinder

The side of the cylinder is part of an infinitely long wall with a radius of L at z=0,and a small radius of S at z=1.This wall has the implicit form as

F(x, y, z)=x2 + y2- (1 + (S - 1) z)2, for 0 < z < 1

If S=1, the shape becomes the generic cylinder, if S=0 , it becomes the generic cone. We develop a hit () method for the tapered cylinder, which also provides hit() method for the cylinder and cone.

3) Intersecting with a Cube (or any Convex Polyhedron)

The convex polyhedron, the generic cube deserves special attention. It is centered at the origin and has corner at (±1, ±1, ±1) using all right combinations of +1 and -1.Thus,its edges are aligned with coordinates axes, and its six faces lie in the plan.

The generic cube is important for two reasons. A large variety of intersecting boxes can be modeled and placed in a scene by applying an affine transformation to a generic cube. Then, in ray tracing each ray can be inverse transformed into the generic cube's coordinate system and we can use a ray with generic cube intersection routine. The generic cube can be used as an extent for the other generic primitives in the sense of a bounding box. Each generic primitives, such as the cylinder, fits snugly inside the cube.

4) Adding More Primitives

To find where the ray S + ct intersects the surface, we substitute S + ct for P in F(P) (the explicit form of the shape)

d(t) = f(S + ct)

This function is positive at these values of t for which the ray is outside the object. zero when the ray coincides with the surface of the object and negative when the ray is inside the surface.

The generic torus has the implicit function as

F(P) = (Px2+Py2) - d)2 + Pz2 - 1

So the resulting equation d(t)=0 is quartic.

CS2401 COMPUTER GRAPHICS UNIT V

21

For quadrics such as the sphere, d(t) has a parabolic shape, for the torus, it has a quartic shape. For other surfaces d(t) may be so complicated that we have to search numerically to locate t's for which d(.) equals zero. The function for super ellipsoid is

d(t) = ((Sx + Cxt)n +(Sy + Cyt)n)m/n + (Sy + Cyt)m -1

where n and m are constant that govern the shape of the surface.

5.8 ADDING SURFACE TEXTURE

A fast method for approximating global illumination effect is environmental mapping. An environment array is used to store background intensity information for a scene. This array is then mapped to the objects in a scene based on the specified viewing direction. This is called as environment mapping or reflection mapping.

To render the surface of an object, we project pixel areas on to surface and then reflect the projected pixel area on to the environment map to pick up the surface shading attributes for each pixel. If the object is transparent, we can also refract the projected pixel are also the environment map. The environment mapping process for reflection of a projected pixel area is shown in figure. Pixel intensity is determined by averaging the intensity values within the intersected region of the environment map.

A simple method for adding surface detail is the model structure and patterns with polygon facets. For large scale detail, polygon modeling can give good results. Also we could model an irregular surface with small, randomly oriented polygon facets, provided the facets were not too small.

Surface pattern polygons are generally overlaid on a larger surface polygon and are processed with the parent's surface. Only the parent polygon is processed by the visible surface algorithms, but the illumination parameters for the surfac3e detail polygons take precedence over the parent polygon. When fine surface detail is to be modeled, polygon are not practical.

5.8.1 Texture Mapping

A method for adding surface detail is to map texture patterns onto the surfaces of objects. The texture pattern may either be defined in a rectangular array or as a procedure that modifies surface intensity values. This approach is referred to as texture mapping or pattern mapping.

The texture pattern is defined with a rectangular grid of intensity values in a texture space referenced with (*s,t*) coordinate values. Surface positions in the scene are referenced with UV object space coordinates and pixel positions on the projection plane are referenced in *xy* Cartesian coordinates.

Texture mapping can be accomplished in one of two ways. Either we can map the texture pattern to object surfaces, then to the projection plane, or we can map pixel areas onto object surfaces then to texture space. Mapping a texture pattern to pixel coordinates is sometime called texture scanning, while the mapping from pixel coordinates to texture space is referred to as **pixel order scanning** or **inverse scanning** or **image order scanning**.

To simplify calculations, the mapping from texture space to object space is often specified with parametric linear functions

$U=fu(s,t)=au\ s+\ but\ +\ cu$

$V=fv(s,t)=av\ s+\ bvt\ +\ cv$

CS2401 COMPUTER GRAPHICS UNIT V

22

The object to image space mapping is accomplished with the concatenation of the viewing and projection transformations.

A disadvantage of mapping from texture space to pixel space is that a selected texture patch usually does not match up with the pixel boundaries, thus requiring calculation of the fractional area of pixel coverage. Therefore, mapping from pixel space to texture space is the most commonly used texture mapping method. This avoids pixel subdivision calculations, and allows anti aliasing procedures to be easily applied.

The mapping from image space to texture space does require calculation of the inverse viewing projection transformation mVP -1 and the inverse texture map transformation mT -1 .

5.8.2 Procedural Texturing Methods

Next method for adding surface texture is to use procedural definitions of the color variations that are to be applied to the objects in a scene. This approach avoids the transformation calculations involved transferring two dimensional texture patterns to object surfaces.

When values are assigned throughout a region of three dimensional space, the object color variations are referred to as solid textures. Values from texture space are transferred to object surfaces using procedural methods, since it is usually impossible to store texture values for all points throughout a region of space (*e.g*) Wood Grains or Marble patterns Bump Mapping.

Although texture mapping can be used to add fine surface detail, it is not a good method for modeling the surface roughness that appears on objects such as oranges, strawberries and raisins. The illumination detail in the texture pattern usually does not correspond to the illumination direction in the scene.

A better method for creating surfaces **bumpiness** is to apply a perturbation function to the surface normal and then use the perturbed normal in the illumination model calculations. This technique is called **bump mapping.**

If *P(u,v)* represents a position on a parameter surface, we can obtain the surface normal at that point with the calculation

$N = Pu \times Pv$

Where *Pu* and *Pv* are the partial derivatives of *P* with respect to parameters u and v.

To obtain a perturbed normal, we modify the surface position vector by adding a small perturbation function called a **bump function**.

$P'(u,v) = P(u,v) + b(u,v)\, n.$

This adds bumps to the surface in the direction of the unit surface normal n=N/|N|. The perturbed surface normal is then obtained as

N'=Pu' + Pv'

We calculate the partial derivative with respect to u of the perturbed position vector as

Pu' = _∂_(P + bn)

∂u

= Pu + bu n + bnu

Assuming the bump function b is small, we can neglect the last term and write

p u' ≈ pu + bun

Similarly p v'= p v + b v n.

and the perturbed surface normal is

N' = Pu + Pv + b v (Pu x n) + bu (n x Pv) + bu bv (n x n).

CS2401 COMPUTER GRAPHICS UNIT V

23

But n x n =0, so that

N' = N + bv (Pu x n) + bu (n x Pv)

The final step is to normalize N' for use in the illumination model calculations.

5.8.3 Frame Mapping

Extension of bump mapping is frame mapping.

In frame mapping, we perturb both the surface normal N and a local coordinate system attached to N. The local coordinates are defined with a surface tangent vector T and a binormal vector B x T x N.

Frame mapping is used to model anisotrophic surfaces. We orient T along the grain of the surface and apply directional perturbations in addition to bump perturbations in the direction of N. In this way, we can model wood grain patterns, cross thread patterns in cloth and streaks in marble or similar materials. Both bump and directional perturbations can be obtained with table look-ups.

To incorporate texturing into a ray tracer, two principal kinds of textures are used. With image textures, 2D image is pasted onto each surface of the object. With solid texture, the object is considered to be carved out of a block of some material that itself has texturing. The ray tracer reveals the color of the texture at each point on the surface of the object.

5.8.4 Solid Texture

Solid texture is sometimes called as **3D texture**. We view an object as being carved out of some texture material such as marble or wood. A texture is represented by a function texture (x, y, z) that produces an (r, g, h) color value at every point in space. Think of this texture as a color or inkiness that varies with position, if u look at different points (x, y, z) you see different colors. When an object of some shape is defined in this space, and all the material outside the shape is chipped away to reveal the object's surface the point (x, y, z) on the surface is revealed and has the specified texture.

5.8.5 Wood grain texture

The grain in a log of wood is due to concentric cylinders of varying color, corresponding to the rings seen when a log is cut. As the distance of the points from some axis varies, the function jumps back and forth between two values. This effect can be simulated with the modulo function.

Rings(r) = ((int) r)%2

where for rings about z-axis, the radius r = √x2+y2 .The value of the function rings () jumps between zero and unity as r increases from zero.

5.8.6 3D Noise and Marble Texture

The grain in materials such as marble is quite chaotic. Turbulent riverlets of dark material course through the stone with random whirls and blotches as if the stone was formed out of some violently stirred molten material. We can simulate turbulence by building a noise function that produces an apparently random value at each point (x,y,z) in space. This noise field is the stirred up in a well-controlled way to give appearance of turbulence.

5.8.7 Turbulence

CS2401 COMPUTER GRAPHICS UNIT V

24

A method for generating more interesting noise. The idea is to mix together several noise components: One that fluctuates slowly as you move slightly through space, one that fluctuates twice as rapidly, one that fluctuates four times rapidly, etc. The more rapidly varying components are given progressively smaller strengths

turb (s, x, y, z) = 1/2noise(s ,x, y, z) + 1/4noise(2s,x,y,z) +1/8 noise (4s,x,y,z).

The function adds three such components, each behalf as strong and varying twice as rapidly as its predecessor.

Common term of a turb () is a

turb (s, x, y, z) = 1/2 1/2Knoise(2ks, x, y, z).

5.8.8 Marble Texture

Marble shows veins of dark and light material that have some regularity ,but that also exhibit strongly chaotic irregularities. We can build up a marble like 3D texture by giving the veins a smoothly fluctuating behavior in the z-direction and then perturbing it chantically using turb(). We start with a texture that is constant in x and y and smoothly varying in z.

Marble(x,y,z)=undulate(sin(2)).

Here undulate() is the spline shaped function that varies between some dark and some light value as its argument varies from -1 to 1.

5.9 REFLECTIONS AND TRANSPERENCY

The great strengths of the ray tracing method is the ease with which it can handle both reflection and refraction of light. This allows one to build scenes of exquisite realism, containing mirrors, fishbowls, lenses and the like. There can be multiple reflections in which light bounces off several shiny surfaces before reaching the eye or elaborate combinations of refraction and reflection. Each of these processes requires the spawnins and tracing of additional rays.

The figure 5.15 shows a ray emanating, from the eye in the direction dir and hitting a surface at the point Ph. when the surface is mirror like or transparent, the light I that reaches the eye may have 5 components

I=Iamb + Idiff + Ispec + Irefl + Itran

The first three are the fan=miler ambient, diffuse and specular contributions. The diffuse and specular part arise from light sources in the environment that are visible at Pn. Iraft is the reflected light component ,arising from the light , Ik that is incident at Pn along the direction –r. This direction is such that the angles of incidence and reflection are equal,so

R=dir-2(dir.m)m

Where we assume that the normal vector m at Ph has been normalized.

Similarly Itran is the transmitted light components arising from the light IT that is transmitted thorough the transparent material to Ph along the direction –t. A portion of this light passes through the surface and in so doing is bent, continuing its travel along –dir. The refraction direction + depends on several factors.

I is a sum of various light contributions, IR and IT each arise from their own fine components – ambient, diffuse and so on. IR is the light that would be seen by an eye at Ph along a ray from P' to Pn. To determine IR, we do in fact spawn a secondary ray from Pn in the direction r, find the first object it hits and then repeat the same computation of

CS2401 COMPUTER GRAPHICS UNIT V

25

light component. Similarly IT is found by casting a ray in the direction t and seeing what surface is hit first, then computing the light contributions.

5.9.1 The Refraction of Light

When a ray of light strikes a transparent object, apportion of the ray penetrates the object. The ray will change direction from dir to + if the speed of light is different in medium 1 than in medium 2. If the angle of incidence of the ray is θ1, Snell's law states that the angle of refraction will be

sin(θ2) = sin(θ1)

C2 C1

where C1 is the spped of light in medium 1 and C2 is the speed of light in medium 2. Only the ratio C2/C1 is important. It is often called the index of refraction of medium 2 with respect to medium 1. Note that if θ1 ,equals zero so does θ2 .Light hitting an interface at right angles is not bent.

In ray traving scenes that include transparent objects, we must keep track of the medium through which a ray is passing so that we can determine the value C2/C1 at the next intersection where the ray either exists from the current object or enters another one. This tracking is most easily accomplished by adding a field to the ray that holds a pointer to the object within which the ray is travelling.

Several design polices are used,

1) Design Policy 1: No two transparent object may interpenetrate.

2) Design Policy 2: Transparent object may interpenetrate.

5.10 COMPOUND OBJECTS: BOOLEAN OPERATIONS ON OBJECTS

A ray tracing method to combine simple shapes to more complex ones is known as constructive Solid Geometry(CSG). Arbitrarily complex shapes are defined by set operations on simpler shapes in a CSG. Objects such as lenses and hollow fish bowls, as well as objects with holes are easily formed by combining the generic shapes. Such objects are called compound, Boolean or CSG objects.

The Boolean operators: union, intersection and difference are shown in the figure 5.17.

Two compound objects build from spheres. The intersection of two spheres is shown as a lens shape. That is a point in the lens if and only if it is in both spheres. L is the intersection of the S1 and S2 is written as

L=S1∩S2

The difference operation is shown as a bowl.A point is in the difference of sets A and B, denoted A-B,if it is in A and not in B.Applying the difference operation is analogous to removing material to cutting or carrying.The bowl is specified by

B=(S1-S2)-C.

CS2401 COMPUTER GRAPHICS UNIT V

26

The solid globe, S1 is hollowed out by removing all the points of the inner sphere, S2,forming a hollow spherical shell. The top is then opened by removing all points in the cone C.

A point is in the union of two sets A and B, denoted AUB, if it is in A or in B or in both. Forming the union of two objects is analogous to gluing them together.

The union of two cones and two cylinders is shown as a rocket.

R=C1 U C2 U C3 U C4.

Cone C1 resets on cylinder C2.Cone C3 is partially embedded in C2 and resets on the fatter cylinder C4.

5.10.1 Ray Tracing CSC objects

Ray trace objects that are Boolean combinations of simpler objects. The ray inside lens L from t3 to t2 and the hit time is t3.If the lens is opaque, the familiar shading rules will be applied to find what color the lens is at the hit spot. If the lens is mirror like or transparent spawned rays are generated with the proper directions and are traced as shown in figure 5.18.

Ray,first strikes the bowl at t1,the smallest of the times for which it is in S1 but not in either S2 or C. Ray 2 on the other hand,first hits the bowl at t5. Again this is the smallest time for which the ray is in S1,but in neither the other sphere nor the cone.The hits at earlier times are hits with components parts of the bowl,but not with the bowl itself.

5.10.2 Data Structure for Boolean objects

Since a compound object is always the combination of two other objects say obj1 OP Obj2, or binary tree structure provides a natural description.

5.10.3 Intersecting Rays with Boolean Objects

We need to be develop a hit() method to work each type of Boolean object.The method must form inside set for the ray with the left subtree,the inside set for the ray with the right subtree,and then combine the two sets appropriately.

```
bool Intersection Bool::hit(ray in Intersection & inter)
{
Intersection lftinter,rtinter;
if (ray misses the extends)return false;
if (C) left –>hit(r,lftinter)||((right–>hit(r,rtinter)))
return false;
return (inter.numHits > 0);
}
```

CS2401 COMPUTER GRAPHICS UNIT V

27

Extent tests are first made to see if there is an early out.Then the proper hit() routing is called for the left subtree and unless the ray misses this subtree,the hit list rinter is formed.If there is a miss,hit() returns the value false immediately because the ray must hit dot subtrees in order to hit their intersection.Then the hit list rtInter is formed.

The code is similar for the union Bool and DifferenceBool classes. For UnionBool::hit(),the two hits are formed using

```
if((!left-)hit(r,lftInter))**(|right-)hit(r,rtinter)))
return false;
```

which provides an early out only if both hit lists are empty.

For differenceBool::hit(),we use the code

```
if((!left–>hit(r,lftInter)) return false;
if(!right–>hit(r,rtInter))
{
inter=lftInter;
return true;
```

}

which gives an early out if the ray misses the left subtree,since it must then miss the whole object.

5.10.4 Building and using Extents for CSG object

The creation of projection,sphere and box extend for CSG object. During a preprocessing step,the true for the CSG object is scanned and extents are built for each node and stored within the node itself. During raytracing,the ray can be tested against each extent encounted,with the potential benefit of an early out in the intersection process if it becomes clear that the ray cannot hit the object.

CS2401 COMPUTER GRAPHICS UNIT V

28