

4 Scripts, algorithms, and other predicaments

An algorithm is a computational procedure for addressing a problem in a finite number of steps. It involves deduction, induction, abstraction, generalization, and structured logic. It is the systematic extraction of logical principles and the development of a generic solution plan. Algorithmic strategies utilize the search for repetitive patterns, universal principles, interchangeable modules, and inductive links. The intellectual power of an algorithm lies in its ability to infer new knowledge and to extend certain limits of the human intellect. An algorithm may be compared to the steps in a recipe; the steps of gathering the ingredients, preparing them, combining them, cooking, and serving are algorithmic steps in the preparation of food. Obviously, the number, size, and quality of ingredients, the sequence and timing of events, as well as the serving and presentation of the final product are key factors to a recipe. Theoretically, an algorithm is the abstraction of a process and serves as a sequential pattern that leads towards the accomplishment of a desired task. For instance, the algorithm for cooking potatoes may be composed of the following steps:

1. Peel
2. Boil
3. Cut
4. Serve

If the steps are reversed or one more step is added or deleted, alternative recipes may be created that produce different results. These results may be better, the same, or worse than the original intention. However, as in cooking, alterations, randomness, or accidents in the process

may lead to new solutions, none of which was known in advance and whose newly emerged identity often differs significantly from the originally intended target. In these cases, the algorithm serves as a pattern of thought that helps in understanding the problem, addresses its possible solutions, and/or is a vehicle for defining new problems.

The common definition of the term algorithm involves the word *finite* as it relates to a number of distinguishable, countable, well-defined and therefore limited, bounded, or determinable series of steps. However, while such an assumption ensures that the description of a solution to the problem, i.e. an algorithm, is composed of finite steps this does not mean that the problem itself has to be finite, bounded, or deterministic. For instance, a common practice in the world of algorithms is something referred to as an "infinite loop" – such a situation is regarded as a misfortune and often results in a termination. While the steps that describe an infinite loop may be finite and specific, the resulting situation is indeterminate and infinite. For instance, the simple repetitive pattern defined through the following statements:

```
A = false;

start:

    if A is false then A = true;
    if A is true then A = false;

go to start;
```

leads to an infinite cyclical argument where A changes between true and false with no end. Yet, the series of statements are indeed finite, well defined, and accurate. Consider now the following simple algorithm:

```
start:

    A = random number between 0 and 10;

    If A is greater than 5 then exit

    Else go to start;
```

In this case, there is a temporary uncertainty about the generation and occurrence of a number greater than 5 to terminate the loop. While eventually such a possibility is

almost certain, its time of occurrence is not necessarily so. The series of statements are finite yet lead to indeterminate, uncertain, and unpredictable behavior.

In the following sections of this chapter basic structures and processes of MEL scripting¹ will be introduced in order to understand, clarify, and illustrate some of the mechanisms, relationships, and connections behind the forms generated. This is not intended to be an exhaustive introduction to scripting but rather an indication of the potential and a point of reference for assessing the value of algorithms.

Variables

A variable is a symbolic representation of a changing data value. Variables can be created using letters preceded by a dollar sign. As long as there is no empty space a variable name should be valid. For example, \$a, \$sum, \$value, \$randomNumber are all valid names. Variables, once defined, are case sensitive, i.e. \$temp is not the same as \$Temp. Case sensitivity applies also to commands, i.e. move is not the same as Move, or polyCube is not the same as polycube.

To assign a value to a variable simply use the = sign and place the data value to the right side. For example, \$a = 5 will assign 5 to variable \$a, or \$x = 3.5 will assign 3.5 to variable \$x (until a new value is assigned). We distinguish two types of arithmetic data: integer numbers (whole) and float numbers (fractional).

A variable can also accept data from a command whose output may be unknown. For example, \$r = rand(0,10) is a case where the output of a random process that creates random numbers between 0 and 10 will be assigned to \$r. In this case, we will not know what the value of \$r is until we print it out using the print command. For example,

```
$r = rand(0,10);  
print $r;
```

These two sentences separated by a semi-colon (;) will assign a random number and print it on the screen.

Variable data types

We use variables to hold data. Variables can be of different types: if we hold whole numbers we called them integer variables and use the symbolic name **int**, if hold characters we call them strings and use the name **string**. There are four variable data types

Table 4.1 Variable data types

Type	Description	Example
int	Integer numbers	-1, 0, 4, 100
float	Fractional numbers	2.3, -0.1, 23.45
string	Characters and words	"a," "apple," "12"
vector	3D coordinates	<<1.1,2.2,4.1>>, <<0,0,0>>

Operations

We have two types of operation: arithmetic and logical. Arithmetic operations are addition (+), subtraction (-), multiplication (*), and division (/). There is also an operation called remainder (%) and returns the remainder of the division between two numbers. For example,

```
$a = 5;
$b = 2;
$c = $a + $b;
```

In this case \$c will become 7. In the following example:

```
$c = $a % $b;
```

\$c will become be 1, because 5 divided by 2 is 2 and the remainder is 1. In brief:

Table 4.2 Arithmetic operations

Operator	Use	Description
+	op1 + op2	Adds op1 and op2
-	op1 - op2	Subtracts op2 from op1
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Computes the remainder of dividing op1 by op2

Logical operations are those that involve the truth or falsity of the comparison between two variables. Those operations are greater (>), greater or equal (>=), equal (==), not equal (!=), less (<), and less or equal (<=). The logical operation is performed using the command **if** followed by parentheses containing the logical operation. As an alternative condition we use the **else** command. For example, the statement:

```
$a=3;
$b=4;
if($a > $b) {
    print $a;
}
else {
    print $b;
}
```

The “if” statement tests the truthfulness of the operation and if it is true executes the following commands enclosed between the curly brackets { and }. In general, curly brackets group statements that need to be executed sequentially as a group. In the following example we will determine the truthfulness of a logical statement whose operand data is unknown.

```
$r1 = rand(0,10);
$r2 = rand(0,10);
if($r1 == $r2){
    print $r1;
}
```

The “if” statement here is used to determine whether the random number \$r1 is equal to the random number \$r2. Notice that the equality test uses the symbol == instead of the data assignment symbol =.

Table 4.3 Logical operations

Operator	Use	Returns true if
>	op1 > op2	op1 is greater than op2
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 <= op2	op1 is less than or equal to op2
==	op1 == op2	op1 and op2 are equal
!=	op1 != op2	op1 and op2 are not equal

Repetition

A repetition statement is referred to as a loop. It consists of the command **for** followed by parentheses containing three parts: an initial condition; a termination condition; and a pace of repetition. For example,

```
for($i=0; $i<10; $i=$i+1){
    print $i;
}
```

is a loop that will initiate a variable `$i` to 0, terminate when the variable is 10, and increment in steps of 1. At each loop, it will print out the value of `$i`. The result would be

```
0123456789
```

Notice that the loop starts at 0 and exits at 10, so 9 is the last printed element. However, as a total, the printed elements are 10.

The statement `$i = $i + 1` can be also expressed as `$i++`. The `++` symbol means “add 1 to” In contrast the symbol `--` means “subtract 1 from” So, below are various loops:

```
for($i=0; $i<20; $i++)
for($i=10; $i>0; $i--)
for($x=-10; $x<10; $x=$x+2)
```

The first loop will start at 0, stop at 20 and increment by 1. The second loop will start at 10, stop at 0, and decrement by 1. The third loop will start at -10, stop at 10, and increment by 2.

There are two commands associated with loops: **continue** and **break**. “Continue” will skip one loop and “break” will exit the whole loop. For example, the following loop:

```
for($i=0; $i<20; $i++){
    if($i==4)continue;
    if($i>7) break;
    print $i;
}
```

will produce the following printout: 0123567. 4 is skipped and after 7 the loop is abandoned.

By using simple arithmetic operations one can produce various number patterns. For instance,

```
for($i=0; $i<20; $i++){
    $x = $i/2;
    print $x;
};
```

will produce the following number pattern (notice that \$i is integer so fractional values will be omitted)

00112233445566778899...

Similarly, the following formulas will result in the following number patterns:

Table 4.4 Repetition patterns

Formula	Result
$x = i/3;$	00011122233344455566
$x = i/4;$	00001111222233334444
$x = (i+1)/2;$	011223344556677889910
$x = (i+2)/2;$	1122334455667788991010
$x = i\%2;$	01010101010101010101
$x = i\%3;$	01201201201201201201
$x = i\%4;$	01230123012301230123
$x = (i+1)\%4;$	12301230123012301230
$x = (i+2)\%4;$	23012301230123012301
$x = (i/2)\%2;$	00110011001100110011
$x = (i/3)\%2;$	00011100011100011100
$x = (i/4)\%2;$	00112233001122330011

Arrays

An array is an ordered set of data. We can have arrays of integers, floats, strings etc. We define an array by using the [] symbol. For example:

```
int $sevenNumbers[];
string $listOfNames[5];
```

The above arrays define 0 and 5 elements respectively. An array can be initialized with data values using the {} operation. For example:

```
int $numbers[4] = {3, 5, 2, 1};
float $temperatures[6] = {103.4, 101.0, 99.3, 98.2,
                          97.3, 98.1};
string $colors[3] = {"brown", "red", "green"};
```

To extract the value of an array member we use an index starting from 0. So, in the following example:

```
int $numbers[4] = {3, 5, 2, 1};
print($numbers[0]);           //should be 3
print($numbers[3]);           //should be 1
```

Once we create an array we can fill it with data and then access them. For example:

```
int $sevenNumbers[];
for($i=0; $i<50; $i++){
    $sevenNumbers[$i] = $i * 2;
}
print($sevenNumbers[31]);     //should be 62
```

The command `size()` returns the size of an array as an integer number. The commands `clear()` and `sort()` will clear and sort arrays respectively. For example:

```
int $x[30];
print(size($x));              //will be 30
clear($x);
print(size($x));              //will be 0
```


Geometrical objects and transformations

Maya has geometrical objects such as curves, surfaces or solids that have names in MEL. For example, a curve is called "curve," a NURBS surface is called "nurbsPlane," a sphere is called "sphere," or a polygonal cube is called "polyCube." For example, to create a sphere, the command "sphere," typed in the editor, will generate a default sphere. These geometrical commands are listed in the **Help->MEL Command Reference**. To modify the parameters of any geometrical object we use modifiers called flags that are also listed in the MEL command reference. For example, the command:

```
sphere -r 5;
```

or

```
polyCube -w 1 -d 0.5 -h 3;
```

will create a sphere of radius 5 and a polygonal cube of width 1, depth 0.5, and height 3. Each flag is defined by a minus sign, a name (or initial) and a data value, i.e. "-w 1" means width of 1 unit.

By default all objects are situated at the 0,0,0 origin at the object's center. To change the location, orientation, or size we use the **move**, **rotate** and **scale** commands. Each command takes three numbers that represent the x, y, and z value of the transformation. For example,

```
sphere -r 1
move 10 3 5.5
rotate 30 0 45
scale 1 2 1
```

will create a sphere of radius 1 and then move it at location 10, 3, 5.5, then rotate it by 30 degrees in the x direction, 0 degrees in the y direction and 45 degrees in the z direction and finally scale the object twice in the y direction but leave it intact in the other directions (1 is the identity operator for scaling). Each transformation also has modifier flags associated with the transformation. The most important flag is -r (-relative) which moves, rotates,

or translates an object relative to its previous location. So, for example, in the previous example:

```
move -r 10 3 5.5
```

will move the sphere 10, 3, 5.5 units away from its previous position to location 20 6 11 (as opposed to the absolute location of 10,3,5.5 which is the default state when no flag is used).

Attributes

A geometrical object may be composed of subelements, i.e. a polygon-based (polyCube) is composed of faces, edges, and vertices and a NURBS-based (cube) surface composed of isoparms and control points. The location of these elements can be addressed using the . (=dot) separator in the form

Object.element

For example, for a polyCube named MyCube, we can extract its first vertex in the following manner:

```
$p = pointPosition MyCube.vtx[0];
```

The pointPosition command returns the actual coordinates of the point in Cartesian space.

To get the attributes associated with an object we use the getAttr command. So

```
$r = getAttr MyCube.rotate
```

will return the xyz rotational angles of MyCube in the variable \$r. These values, in turn, can be extracted as

```
float $rx = $r[0];
```

```
float $ry = $r[1];
```

```
float $rz = $r[2];
```

In this case \$r is an array with three elements that correspond to the values of xyz in the order 012.

Similarly, the xyz translational coordinates of MyCube can be obtained using the command

```
$t = getAttr MyCube.translate
```

This array \$t holds the xyz coordinates of the center of the object. To get the actual location of each vertex we use the pointPosition command (above).

Sometimes the name of an object may be a variable, i.e. \$object_name. If we use the getAttr command the system will return an error because the following

```
$t = getAttr $object_name.translate
```

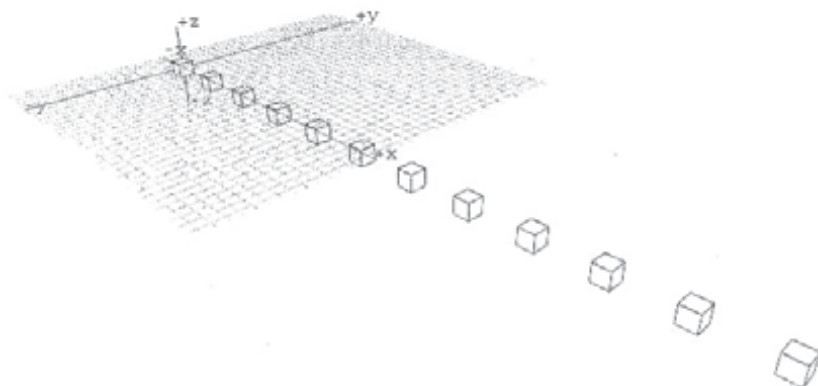
is confusing. So, instead we create the actual command as a string and then execute it using the eval command:

```
$t = eval("getAttr" + $object_name + ".translate");
```

Sequential transformation

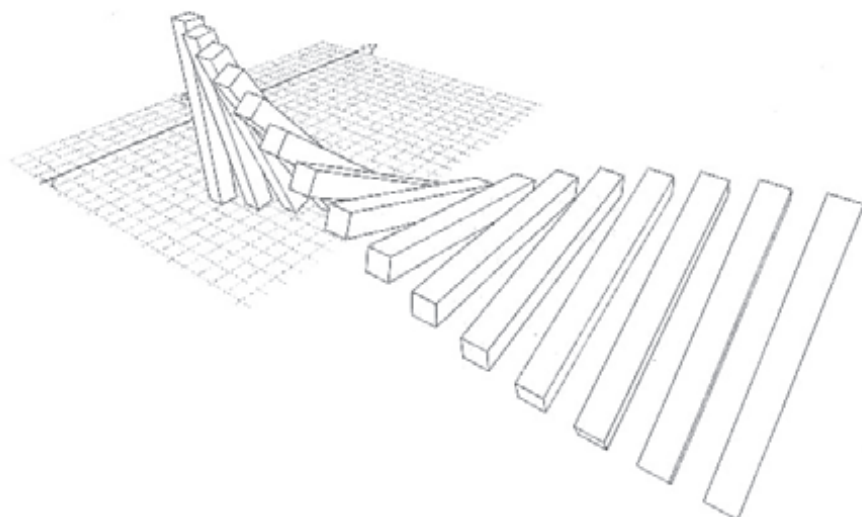
Using loops, geometrical objects, and transformations we can place objects sequentially creating rhythm, repetition, or progression. For example, the following code will generate 12 cubes of half-unit and place them in equal distance of 2 units in the x direction:

```
for($x=0; $x<12; $x++){
    polyCube -w 0.5 -d 0.5 -h 0.5;
    move ($x*2) 0 0;
}
```



Similarly, the following code will generate a series of 15 cubes, scale them along the z axis, move them 1 unit apart, and progressively rotate them by 10 degrees in the x direction:

```
for($x=0; $x<15; $x++){
    polyCube -w 1 -d 1 -h 1;
    move ($x) 0 0;
    scale 0.5 0.5 5;
    rotate ($x*10) 0 0;
}
```

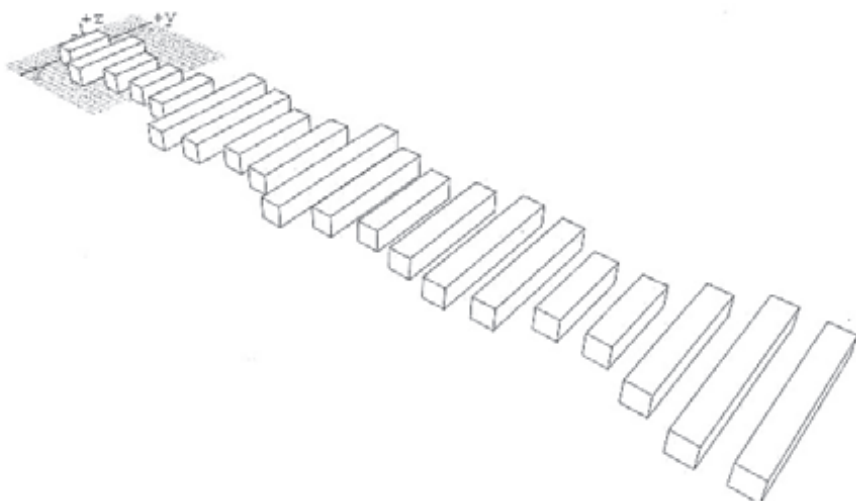


4.2

Rotational progression

We can also introduce randomness in the process by adding (or subtracting) a randomly generated value. For example, the following code will create 20 sequential cubes whose height will be random:

```
for($x=0; $x<20; $x++){
    $r = rand(-3, 3);
    polyCube -w 1 -d 1 -h (5+$r);
    move ($x*2) 0 0;
}
```

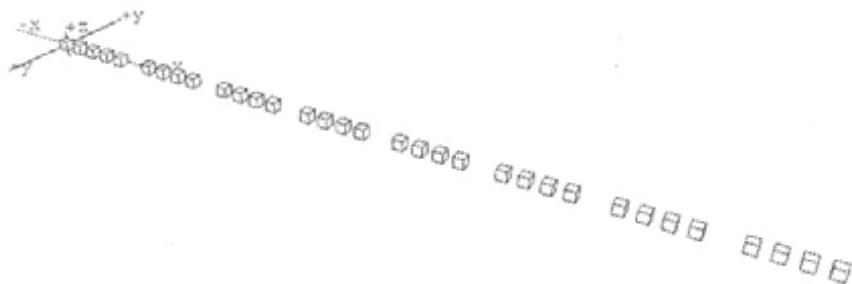


4.3

Scaling variations

The remainder operation (%) can also be used to create repetition at variable distances. Because the remainder of the division of any number by a divisor will always be 0 if the divisor is divided exactly we can use this property to create rhythm. For example:

```
for($x=0; $x<40; $x++){
    if($x%5==0) continue;
    polyCube -w 1 -d 1 -h 1;
    move ($x*2) 0 0;
}
```

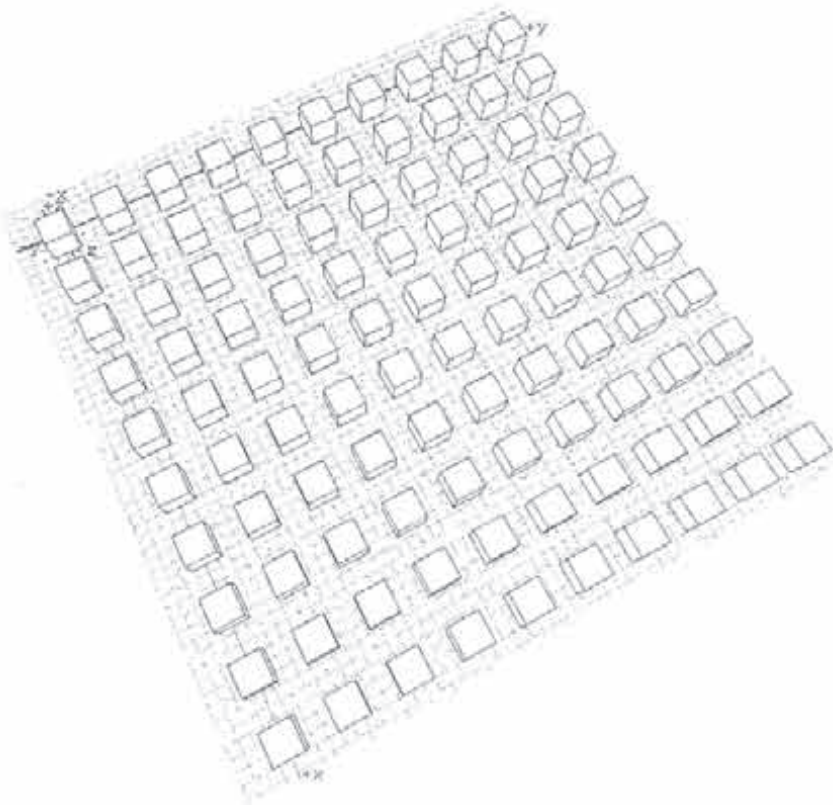


will skip one every five cubes because the remainder of the division of $\$x$ with 5 is 0 every time $\$x$ is a multiple of 5.

Loops can occur in one direction, but also in two or three directions. For example, the following code:

```
for($y=0; $y<10; $y++){
  for($x=0; $x<10; $x++){
    polyCube -w 1 -d 1 -h 1;
    move ($x*2) ($y*2) 0;
  }
}
```

will produce a series of rows of cubes in the x direction (as seen in the inner x loop) but then will produce sequences of rows in the y direction (as seen in the outer y loop).



4.5

A two-dimensional grid

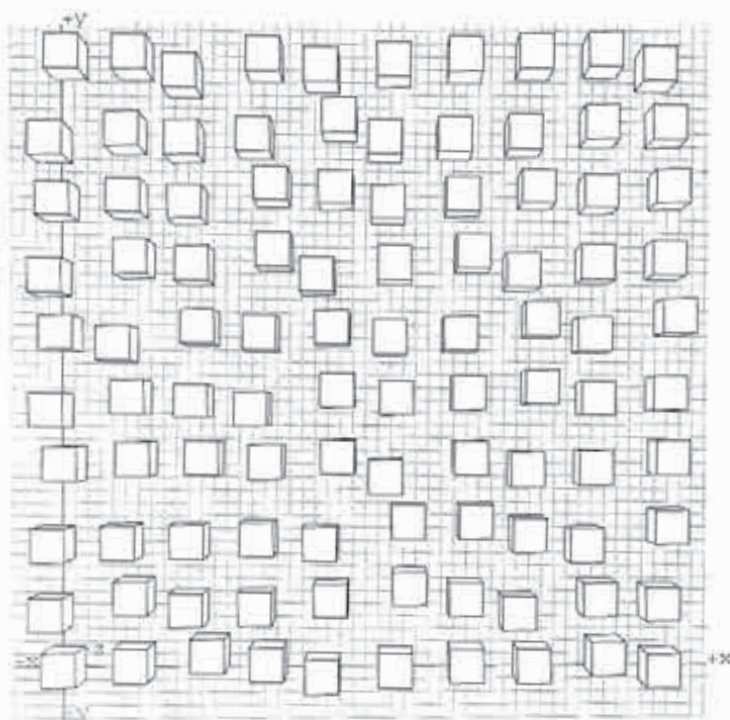
The command **rand**(\$min, \$max) will create a random number between \$min and \$max. For example:

```
$r = rand(-5, 5)
```

will create a random fractional number (i.e. 2.1 or -4.1) between -5 and 5.

Randomness can always be introduced as a deviation from an orderly placement. For example, adding a random number to an existing location will produce the following code and effect:

```
for($y=0; $y<10; $y++){
  for($x=0; $x<10; $x++){
    polyCube -w 1 -d 1 -h 1;
    $r = rand(-0.2, 0.2);
    move (($x+$r)*2) (($y+$r)*2) 0;
  }
}
```



The amount of randomness added to an orderly pattern can be subtle enough to create a balanced composition of order and disorder.

Multi-Booleans

Geometric Boolean operations are used to articulate the presence or absence of material substances. Theoretically, there are three boolean operations, OR, AND, and NOT, that correspond to logical operations. However, in the context of solid objects, these operations correspond to union, intersection, and difference. While the minimum operands



4.7

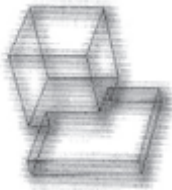
Order and randomness (class project by M. Snyder for course GSD2311 taught by Kostas Terzidis in Fall 2005 at Harvard University)



4.8

Disturbed landscape (class project by C. Shusta for course GSD2311 taught by Kostas Terzidis in Fall 2005 at Harvard University)

for these operations are always two, combination or repetitive application of the operations can result in complicated forms.



A union B



A intersection B



A difference B



B difference A

4.9

Boolean operations

Traditionally, architects have maintained a logic of accumulative progression during the design process. Because of the artificial nature of design, architects traditionally follow a bottom-up approach where elements are composed into objects, and objects into groups to form structures, systems, and buildings. Iakov Chernikov and other constructivists, for example, elaborated on the combination of forms, using the basic concepts of "constructive combinations," such as combination, assemblage, penetration, mounting, integrating, coupling, interlacing, and so on, both statically and dynamically, using hard or soft materials. These concepts constitute a Boolean design language, one having the advantage of combining forms in a manner familiar to architects.

The following code shows a simple method to union multiple objects and to difference one object from multiple objects creating holes or niches:

```

1  seed(5.); //optional, in order to repeat randomness
2
3  for($i=0; $i<20; $i++){
4      $name = "MyObject" + $i;
5      polyCube -name $name;
6      move (rand(-.5,.5)) (rand(-.5,.5)) (rand(-.5,.5));
7      scale (rand(2.5,3.5)) (rand(2.5,3.5)) (rand(2.5,3.5));

```

```

8 } //for i
9
10
11 polyBoolOp -op 1 -name MyResult1 MyObject0 MyObject1; //union the first
    two objects
12
13 for($next=2; $next <20; $next++){ // go for the rest
14     $obj_next = "MyObject" + $next; //define the next object
15     $previous = $next - 1;
16     $obj_previous = "MyResult" + $previous;
17     $result = "MyResult" + $next; //define the previous object
18     polyBoolOp -op 1 -name $result $obj_previous $obj_next; //union
        the previous with the next
19 } // for next
20
21 rename MyResult19 MyResult0; //rename the object to start the difference
    process
22
23 for ($i=0; $i<20; $i++) { //for 30 objects to be differences (i.e. holes)
24     //create a cylinder to be used for subtraction (difference)
25     polyCylinder -n ("MyCylinder" + $i) -h (rand(8,9)) -r (rand (.05,.1));
26     move (rand(-1,1)) (rand(-1,1)) (rand(-1,1)); //move anywhere within
        the target body
27     //subtract (difference) the previous with the next
28     eval ("polyBoolOp -op 2 -n MyResult" + ($i + 1) + "MyResult" +
        $i + "MyCylinder" + $i);
29 }; //for i

```

The first line of the code uses the command `seed` that initializes a sequence of random numbers. The statement is optional and should be used to produce the exact same effect of random sequences. Lines 3 to 8 are a description of creating 20 cubes of various shapes that overlap one another. At line 11 we union the first two objects (i.e. `MyObject0` and `MyObject1`) and create

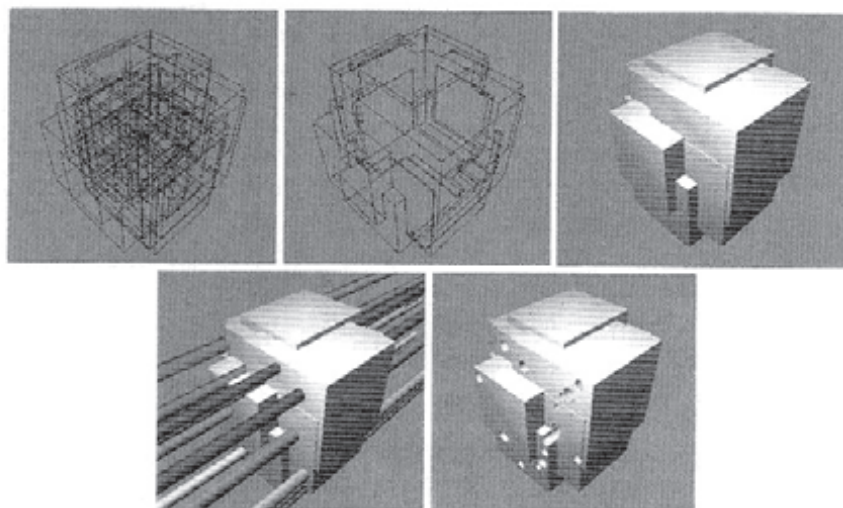
a combined object called `MyResult1`. We then loop through the rest of the objects (i.e. 2 to 20) to union every object with the next one. At each step we create a resulting object called `MyResult` which is used as the previous object.

At line 21 we are renaming the last and only object in the scene which is the result of the combined union. We call it `MyResult0`. This object will be used as a target for difference operations to follow.

We loop through 20 times and create long cylinders that are dispersed so that they can penetrate throughout the target object `MyResult0`. We then subtract each cylinder from the target object and then use the new object created to subtract from it again.

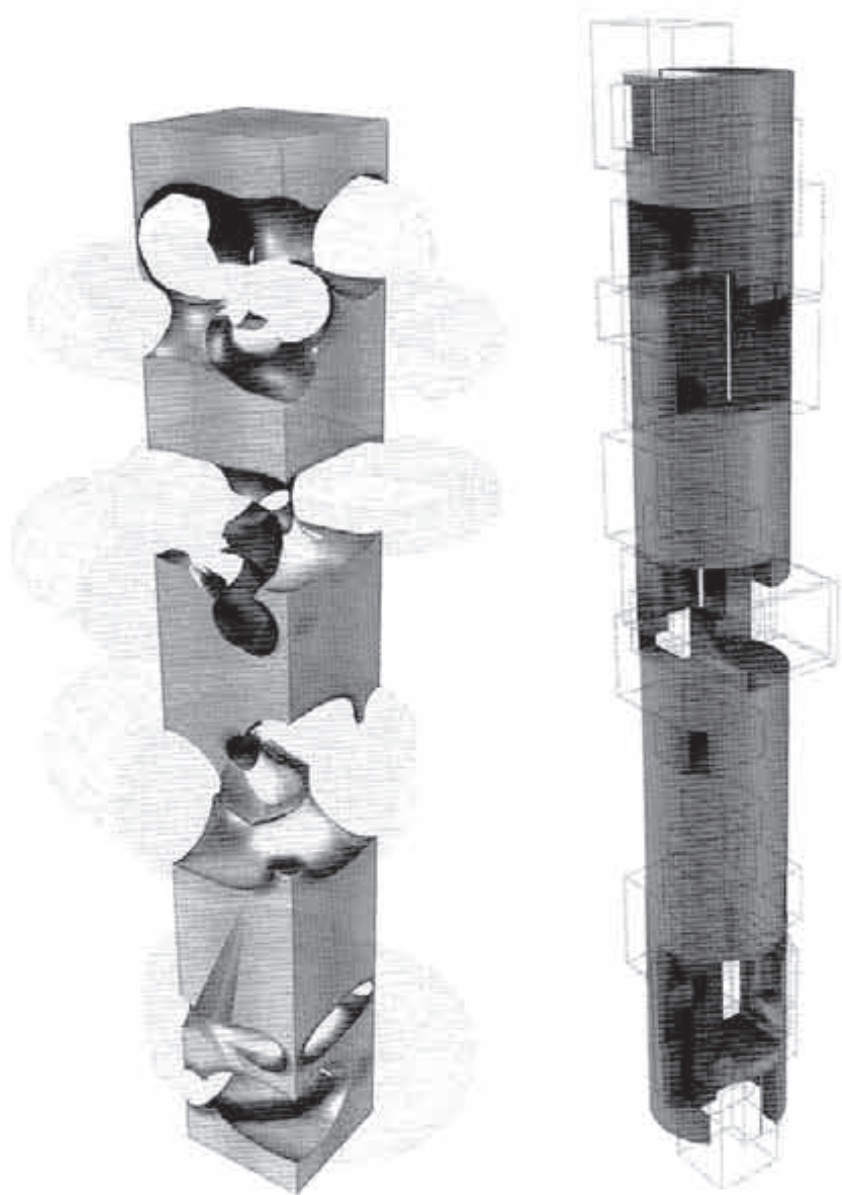
The results of these operations are shown in Figures 4.10 and 4.11.

Similarly, a series of spheres can be subtracted from the cube or a series of cubes can be subtracted from a cylinder. While the original objects are subtracted and therefore deleted, their presence is still visible through the mold that reflects their absence. Absence becomes the state of not being present.



4.10

Union of multiple cubes (top) and subtraction of multiple cylinders (bottom)



4.11

Subtracted objects are still present through their absence (class project by C. Shusta for course GSD2311 taught by Kostas Terzidis in Fall 2005 at Harvard University)

Stochastic search

A stochastic search is defined here as a random search in space until a given condition is met. For instance, the placement of toys in a playpen so that each toy does not

overlap another and they all fit within the limits of the playpen can be addressed with a stochastic search. The algorithm will work as follows:

```
while(no more toys left to place){
    choose randomly a position (rx, ry) within the playpen
    compare it with all previous toy locations
    is there an overlap? (if no then place the
    toy at (rx, ry))
}
```

This algorithm can use used to place objects within a site so that there is no overlap (or some other criterion is satisfied). In the following code, a series of 100 cubes is placed within an area of 10×10 .

```

1  for($i=0; $i<100; $i++){ //for all objects
2
3      $name = "MyCube" + $i; //create a series name
4      polyCube -w 1 -d 1 -h 1 -name $name; //make a cube
5
6      $range = 10; //define a range
7      $sv=0; //set a safety valve to avoid infinite loops
8      while(true){ //search forever
9          $rx = rand(-$range,$range); //get a random x position (dart)
10         $ry = rand(-$range,$range); //get a random x position
11         $overlap = false; //set a flag a false (until proof of the opposite)
12         for($j=0; $j<$i; $j++){ //loop through all the previous
13             $name = "MyCube" + $j; //get a previous name
14             $px = eval("getAttr "+ $name + ".translateX");
15             //get the x location of the previous
16             $py = eval("getAttr "+ $name + ".translateY");
17             //get the y location of the previous

```

fi w fj

```

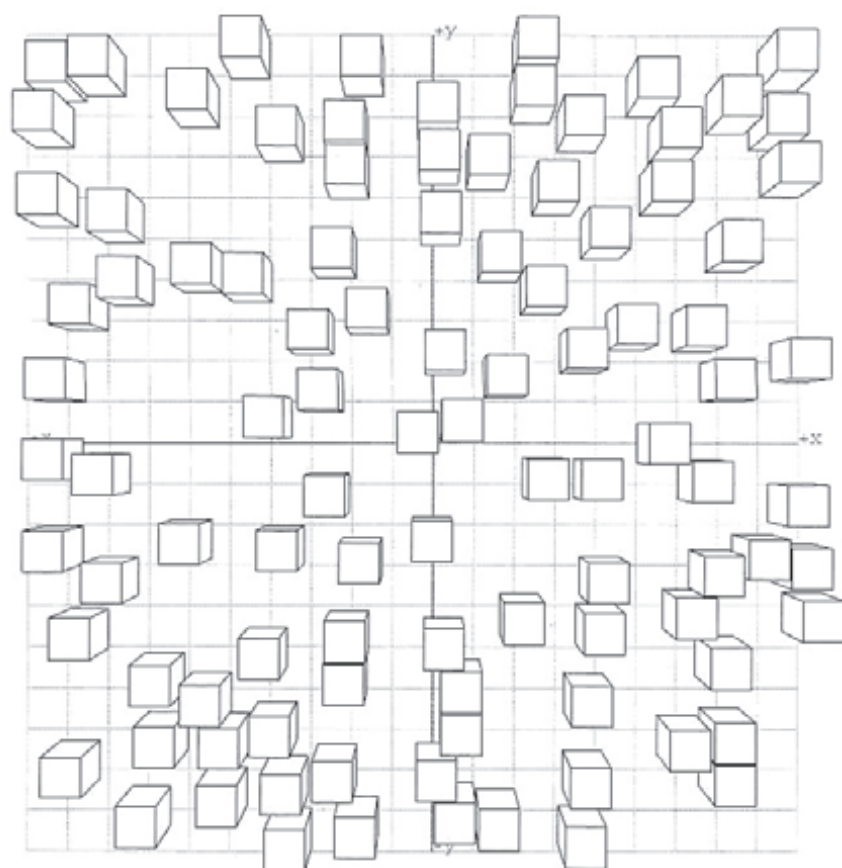
16      $diffx = abs((float)($rx-$px)); //get the x distance
      between previous and candidate
17      $diffy = abs((float)($ry-$py)); //get the y distance
      between previous and candidate
18      if($diffx < 1. && $diffy < 1. ) // If x and y distance is less
      than tolerance (i.e. 1)
19          $overlap = true; //set the flag
20      - } //for j again
21      if($overlap==false){ //if there is no overlap
22          move $rx $ry 0; // move the cube in the location (the position
      satisfies the criterion)
23          textCurves -f "Courier" -t $i; // make a label with its serial
      number to identify it
24          move $rx $ry 0; // move the label to the middle of the new cube
25          scale -r 0.1 0.1 0.1; // scale the label
26          break; //exit the while loop
27      }
28      $sv++;if($sv>1000){ print("not found\n");break;} //if 1000
      unsuccessful attempts are made exit
29      - }//while (again)
30
31  }; //for $i

```

The algorithm starts with an outer loop where a hundred cubes are to be placed. First, each cube is named sequentially as MyCube0, MyCube1, MyCube2, etc. and then created (lines 3 and 4). A range is defined as 10 and a counter \$sv is initialized to be used later in order to avoid infinite loops (to be called here a safety valve). Then, in lines 9 and 10 we create random locations and x and y between negative and positive ranges. If flag is being set to false that will be used later to determine whether there's an overlap or not. Now, we need to loop back through all the objects that have been created already (line 12). So, we get the name of each previous cube and use it to extract the translational value in both x and y, which we use to determine the distance between the previous and the candidate. In other words, given a random location we need to determine whether positioning a cube will overlap any of the previous cubes. In line 18

we test to see whether the difference is enough to allow a cube to be placed in between. If the distance is not enough then set the flag to be true. Else we set the flag to false. We continue looping for all the objects and if the flag continues to be false (i.e. there is no overlap) we move the cube at the candidate random location (line 22). The next three lines of code create 3D text with the number of the cube (for identification purposes) and then break out of the while loop. Line 28 is referred to as a "safety valve" and its purpose is to force an exit to the loop if an infinite loop situation occurs. The counter \$sv increases by one at each attempt, so if the number of unsuccessful attempts exceeds 1000 (or any sufficiently large number) then the system is forced to exit the loop.

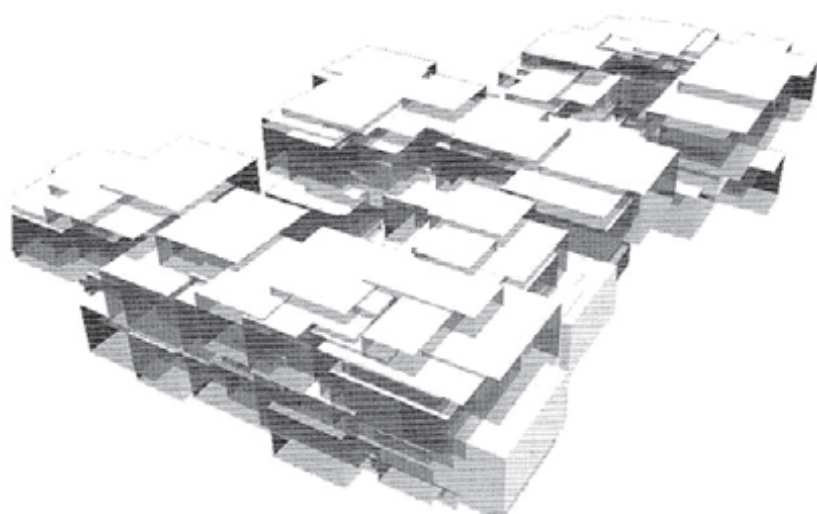
Figure 4.12 shows the placement of a hundred troops in an area of 10×10 without overlaps.



4.12

Distribution of 100 cubes within a 10×10 site with no overlap

Of course, this algorithm can be modified to allow only overlaps by reversing the logic, or the conditions for placement can be so complex that they can satisfy various architectural conditions (i.e. public space, sun exposure, zoning envelope, etc.).



4.13

A housing development that uses stochastic search to determine the location of its units

Fractals

A fractal is a geometric object generated by a repeating pattern, in a typically recursive or iterative process. Usually, the outcome shape can be divided into parts, each of which is similar to the original shape. Fractals are said to possess infinite detail, and some of them have a self-similar structure that occurs at different levels of magnification. The term fractal was coined in 1975 by Benoît Mandelbrot, from the Latin word *fractus* or *broken*.

In a fractal process, there are at least two shapes: a base and a generator. In each iteration, the generator replaces each segment of the base shape. Theoretically this process continues infinitely. The algorithm to create fractals consists of a basic procedure that fits a shape between two points. The process of fitting involves scaling, rotation, and translation of the generator to fit between two points

of a segment of the base. The following code shows the procedure:

```

1 proc fit (vector $start, vector $end, vector $shape[]){
2
3     // make a curve out of the array points in shape[]
4     string $points = ""; //create an empty string
5     $point = $shape[0]; //get the first point
6
7     for($i=0; $i<size($shape); $i++){ //for all points of the shape
8         $point = $shape[$i]; //extract a point
9         $points += " -p " + $point; //add it to the string
10    }
11    eval("curve -d 1 " + $points); //create a curve out of the input points
12    in $shape[]
13
14    //scale to fit
15    $temp_end = $end - $start; // move the points to the origin
16    float $mag = mag($temp_end); // get the magnitude of their length
17    float $amag = mag($point); // get the magnitude of the length of the
18    base segment
19
20    float $scale_factor = $mag/$amag; //get the scaling factor
21    scale $scale_factor $scale_factor $scale_factor; // scale
22
23    //rotate to fit
24    float $angle = atan2d($temp_end.y,$temp_end.x); //atan2d gives the
25    angle
26    rotate 0 0 $angle; // rotate from the origin to a point
27
28    //move to fit
29    move ($start.x) ($start.y) ($start.z); // move back to the original location
30
31    refresh; //refresh the screen to see the replacement as it happens
32 } //procedure end

```


The procedure called `fit` takes as input a point called `$start`, a point called `$end`, and an array of points that define the shape to be fitted. Each point is defined here as a vector. A vector is a data type that contains three float numbers for `x`, `y`, and `z`, which can be extracted using the dot operator, i.e. for a vector called `$start` we have `$x = $start.x`; `$y = $start.y`; and `$z = $start.z`.

First, we create an empty string to be filled with points that will later on define a curve. In line 5 we extract the first point of the input shape (to be used later in line 16). Then we loop through all the points of the input shape, extract each point and put them into the variable `$point`, and concatenate each point to the string `$points` (lines 8 and 9). The command `size()` is used here to extract the size of the array `$shape`. After the loop is done, we create a curve of dimension one (a polyline). This is the generator shape that will be fitted between points `$start` and `$end`.

Next we scale the curve to fit between the two points. In line 14 we subtract start from end thus bringing the gap to be fitted to the origin (0,0,0). We then get the magnitude of the gap as well as the magnitude of the length of the curve to be fitted. We divide the two magnitudes attending a factor for scale. We use that factor to scale the curve.

Then we rotate the curve by an angle. This angle points at the direction of the end point. The command `atan2d` returns the angle from the origin to a point. In this case, the point is the offset end point `$temp_end`. Finally, we move the point back to its original location. (The `refresh` command is used here as an effect so that we can see the replacement as it happens.)

The next procedure is the main one where the fractal operation is called (the identifier **global** allows it to be called from everywhere and at anytime):

```

1  global proc fractal(){
2
3      vector $generator[]; //make an array for the generator
4      $generator = getPoints("curve1"); //populate the array with the curve1
        (generator) points
5      vector $base[]; //make an array for the base
```



```

6
7  string $list[];
8  $list = eval("ls -transforms \"curve*\""); //returns an array with the names of
9                                           //all objects in the scene that match the
                                           word "curve"
10
11 for($j = 1; $j<size($list); $j++){ //we start at 1 because 0 is the generator
12   $base_name = $list[$j];
13
14   $base = getPoints($base_name); //get the points
15   //once we have the points we erase the shape since we will replace it
16   eval("delete " + $base_name);
17
18   for($i=0; $i<(size($base)-1); $i++)
19     //replace the current (i) and next (i+1) points with the
     generator's array
20     fit($base[$i], $base[$i+1], $generator);
21   } //for j
22 } // procedure end

```

The fractal procedure starts by defining two main arrays: generator and base where the points of the generator and base shape will be stored respectively. In line 4 a procedure called `getPoints` is called that will extract the points from a shape and put them in an array (this procedure will be defined later). At this point we assume



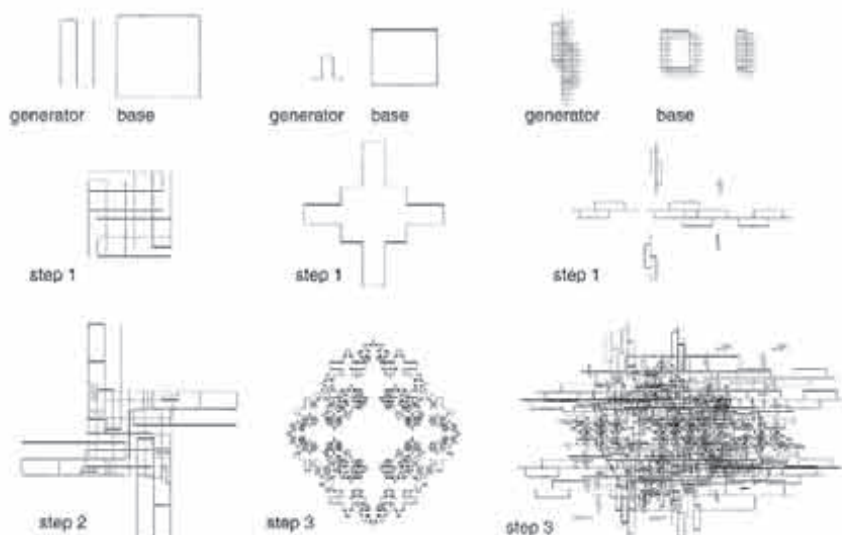
that we have created two curves (of dimension 1) that are called by default names *curve1* (generator) and *curve2* (base). It is important that the generator starts at the origin and that the y-axis is upwards.

In line 8 we use this command to extract all the objects in the scene that match the word *curve*. These curves will be replaced by the generator. So, in line 11 we loop through all the curves except the first one because that is the generator. We extract their name and then extract their points and put them in an array called *\$base*. Once we are done we delete the base shape because it is not needed anymore. We then loop through all the points of each segment of the base shape and replace it with the generator (line 20).

The next procedure extracts the points of a given curve and puts them (populates) into an array which it then returns as an array of vectors:

```

1  proc vector[] getPoints(string $curve_name){
2
3      //get the number of spans
4      $numSpans = eval("getAttr " + $curve_name + ".spans");
5      vector $points[]; //make a vector array to collect the points
6
7      for($i=0; $i<($numSpans+1); $i++){
8          $point = eval("pointPosition " + $curve_name + ".cv[" +
              $i + "]");
              //get the cvs
9          float $x = $point[0];
10         float $y = $point[1];
11         float $z = $point[2];
12         vector $v = <<$x, $y, $z>>;
13         $points[$i] = $v; //store the values in points[]
14     } //for i
15     return $points; //return the array
16 } //procedure end
```

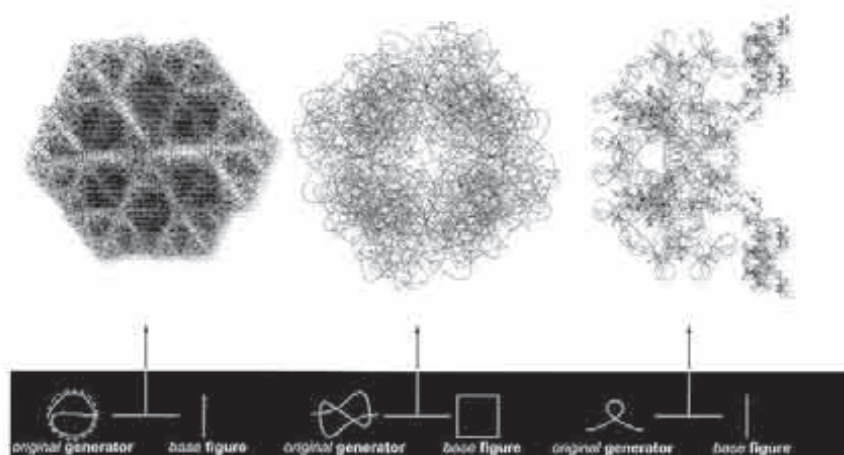


4.15

Various fractals (class project by K. Hopkins for course GSD2311 taught by Kostas Terzidis in Fall 2005 at Harvard University)

First, the number of points (i.e. spans) need to be extracted. We use the `geAttr` command to extract the number of spans (line 4), then we loop through all the spans to extract the position of each control point. We put these coordinates into a vector array and return it.

Below are some examples using fractals:

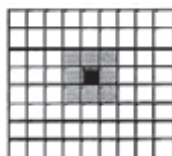


4.16

Fractals with curves (class project by K. Takeuchi for course GSD2311 taught by Kostas Terzidis in Fall 2005 at Harvard University)

Cellular automata

A cellular automaton (plural: cellular automata) is a discrete model that consists of a finite, regular grid of cells, each in one of a finite number of states. Time is also discrete, and the state of a cell at a time slice is a function of the state of a finite number of cells called the neighborhood at the previous time slice. Every cell exhibits a local behavior based on a rule(s) applied which in turn is based on values in its neighborhood. Each time the rules are applied to the whole grid a new generation is produced.



4.17

An 8-neighborhood

While cellular automata (CA) were developed originally to describe organic self-replicating systems, their structure and behavior were also useful in addressing architectural, landscape, and urban design problems. From vernacular settlements and social interaction to material behavior and air circulation, CA may provide interesting interpretations of urban and architectural phenomena. The basic idea behind CA is not to describe a complex system with complex equations, but to let the complexity emerge by interaction of simple individuals following simple rules. Typical features of CA include: absence of external control (autonomy), symmetry breaking (loss of freedom/heterogeneity), global order (emergence from local interactions), self-maintenance (repair/reproduction metabolisms), adaptation (functionality/tracking of external variations), complexity (multiple concurrent values or objectives), and hierarchy (multiple nested self-organized levels).

The following algorithm was developed as a kernel to implement cellular automata for architectural purposes:

```

1      int $xmax = 40;
2      int $ymax = 40;
3
4  //create the lattice of cells
```

```

5  for($x=0; $x<$xmax; $x++) // loop in x-direction
6      for($y=0; $y<$ymax; $y++){ // loop in y-direction
7          polyPlane -ax 0 0 1 -w 1 -h 1 -sx 1 -sy 1 -name ("MyPlane" +
            $x + "x" + $y);
8          move $x $y 0;
9      }
10 //disurb it
11 for($x=0; $x<$xmax; $x++) // loop in x-direction
12     for($y=0; $y<$ymax; $y++){ // loop in y-direction
13         $name = ("MyPlane" + $x + "x" + $y);
14         if(rand(2.) > 1.) eval("setAttr "+$name+".visibility 0");
            //show or hide
15     }
16
17 int $status[]; //keep a memory of the current state of each cell
18
19 for($gen=0; $gen<10; $gen++) { //the number of trials (generations)
20
21     //first pass: collect information from the neighbors
22     int $idx = 0; //initialize a counter
23     for($x=1; $x<$xmax-1; $x++) // loop in x-direction
24         for($y=1; $y<$ymax-1; $y++){ // loop in y-direction
25             $name = ("MyPlane" + $x + "x" + $y);
26             $visible = 0.;
27             for($i=-1; $i<=1; $i++) // loop by three positions
28                 for($j=-1; $j<=1; $j++){ // loop by
                    three positions
29                     if($i==0 && $j==0)continue;
                        // exclude the cell itself
30                     $nameNeighbor = ("MyPlane"
                        + ($x+$i) + "x" + ($y+$j));
31                     $v = eval("getAttr
                        "+$nameNeighbor+".visibility");
                        //get value

```



```

32                                     $visible += $v; //add them
33                                     }
34                                     $status[($idx)] = $visible; //remember each cell's
                                     neighborhood condition
35                                     $idx++; //increment the counter
36                               } //for y
37
38                               //Second pass: apply a simple rule that may amount into a complex
                               pattern
39                               $idx = 0; // initialize the counter
40                               for($x=1; $x<$xmax-1; $x++) // loop in x-direction
41                                   for($y=1; $y<$ymax-1; $y++){ // loop in y-direction
42                                       $name = ("MyPlane" + $x + "x" + $y);
43                                       if($status[($idx)] == 3) //rule 1
44                                           eval("setAttr "+$name+".visibility 1");
                                           //show
45                                       else if($status[($idx)] >= 6) //rule 2
46                                           eval("setAttr "+$name+".visibility 0");
                                           //hide
47                                       $idx++;
48                                   } //for y
49
50                               refresh; //show the pattern as it evolves
51
52 } // next generation

```

The first two lines define the size of the grid (40×40 cells). The first loop starting at line 5 creates a grid of polygonal planes with a unique name that corresponds to their x, y location, and the second loop (at line 11) creates random disturbance of visibility: each plane (cell) has a 50% chance of being opaque or transparent. Next, we create an array of integers to store the current state of each cell (line 17). Now, a loop of 10 iterations corresponding to 10 generations (or number of trials) is initiated within which there will be two passes; we can now apply the rules to each cell individually.

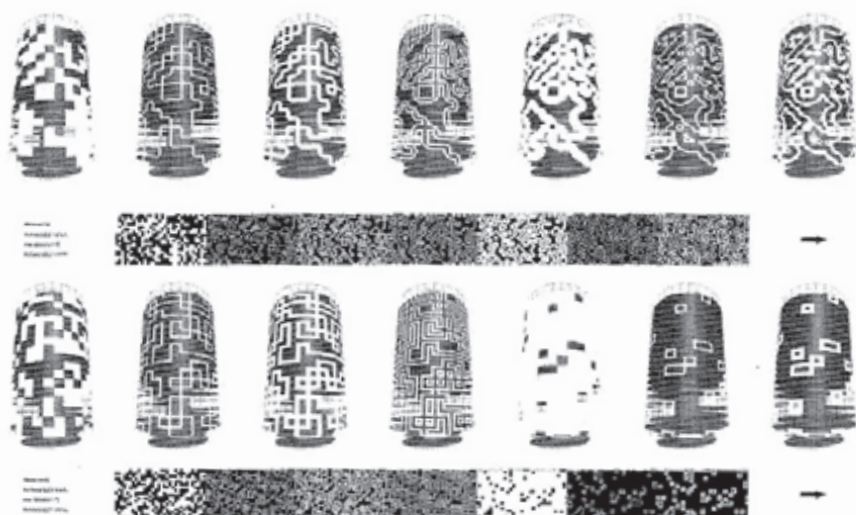
The first pass (line 23) goes for all the cells of the grid except the ones that lie on the borders, because those cells have no neighbors on at least one of their sides (in other implementations the border cells can wrap around to corresponding neighbors on the opposite side of the grid). As we reach each cell we recall its name (line 25) and set a counter called \$visible to zero; this counter will be used to count the number of visible cells in a neighborhood of 3×3 . So, we loop twice in the x and y directions from -1 to 1 excluding the case where both counters are 0 and extract the visibility attribute of each neighbor. If the visibility is one we increase the counter by one. This way we are able to detect the number of visible neighbors on a three by three neighborhood and place that number in an array (\$status).

In the second pass we apply the rules to each cell given its neighborhood values: we loop in the x and y directions again, extract the name of the cell (line 42), and then apply the rules. In this particular case, there are two rules: (a) if the number of visible neighbors is equal to three then the cell should be visible and (b) if the number of visible neighbors is greater than or equal to six then the cells should be invisible (lines 43 to 46). The last line of code is a command called refresh and is used here to show the pattern as it evolves.

Figures 4.18 and 4.19 show examples of the use of CA in an architectural context.

Hybridization

Hybridization (a.k.a. morphing) is a procedure in which an object changes its form gradually in order to obtain another form. Morphing is a gradual transition that results in a marked change in the form's appearance, character, condition, or function. The operation of morphing consists basically of the selection of two objects and the assignment of a number of in-between transitional steps. The first object then transforms into the second in steps. The essence of such a transformation is not so much in the destination form but rather in the intermediate phases these transformations pass through, as well as in the extrapolations, which go beyond the final form. It is the transitional continuity of a form that progresses through a series of evolutionary stages.



4.18

Cellular automata as an LCD display wrapped around a building (class project by N. Anderson for course GSD2311 taught by Kostas Terzidis in Fall 2005 at Harvard University)

Architectural morphing preserves the structural integrity of the objects involved, that is, an object changes into another object as a single entity. A cube, for instance, may be gradually transformed into a pyramid. From the viewer's point of view, there are always two objects: *the original (or source)*, to which transformation is applied, and the *destination object (or target)*, which is the object one will get at the final step of the transformation. However, theoretically, there is only one object, which is transformed from one state (original) into another (destination). This object combines characteristics of both



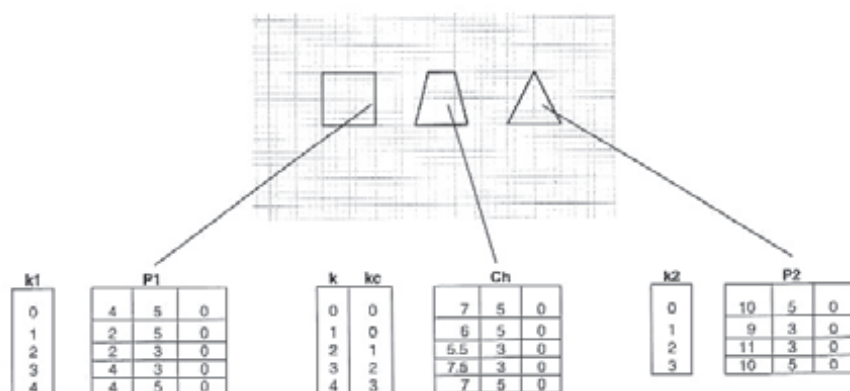
4.19

Cellular automata maze (class project by M. Snyder for course GSD2311 taught by Kostas Terzidis in Fall 2005 at Harvard University)

parent objects, which are involved in the transformation and is called a *hybrid object*. This object is actually composed of the topology of one object and the geometry of the other. It is an object in disguise. Although it is topologically identical to the one parent it resembles the geometry of the other parent.

Interpolation is a method for estimating values that lie between two known values². The hybrid object derives its structure from its parents through formal interpolations. While it is easy to derive hybrid children from isomorphic parents, a challenge arises for heteromorphic parents. In an isomorphic transformation, a one-to-one correspondence applies between the elements of the two parent sets, such that the result of an operation on elements of one set corresponds to the result of the analogous operation on their images in the other set. In the case of heteromorphism, the lack of homogeneity among the parents leads necessarily to a selective process of omissions and inclusion of elements between the two sets. The guiding principle in this mapping process is the preservation of the topological and geometrical properties of the hybrid object. For instance, in the case of a square mapped to a triangle, the addition of a fourth point to the triangle preserves the topology of the square and yet its disguised location preserves the geometrical appearance of the triangle.

In the example in Figure 4.20, a square is mapped to a triangle: the hybrid child is a four-sided polygon in which two



4.20

The coordinates of the parents and the hybrid child

of the vertices overlap and are ordered to form a triangle. The problem here is to map two counters so that when the one is counting points from one object to another the counter *kc* should skip points from the other object. For example, if the one counter *k* increments as 01234 the counter *kc* should increment as 00123 (or 01123 or 01223 or 01233). To obtain such behavior, we use the function $kc = k/(p1/p2)$ or $kc = k/(p2/p1)$.

```

1  global proc hybrid(string $parent1, string $parent2, float $ratio){
2
3      int $p1pnts = eval("getAttr "+ $parent1 + ".spans"); //number of
        points of parent 1
4      int $p2pnts = eval("getAttr "+ $parent2 + ".spans"); //number of points of
        parent 2
5      int $degree = eval("getAttr "+ $parent2 + ".degree");
6      int $numpoints = max($p1pnts, $p2pnts); //child has the number of
        points of the biggest parent
7      int $k1 = 0; // counter 1
8      int $k2 = 0; // counter 2
9
10     string $spoints = ""; //string to hold curve values
11     float $point[];
12
13     for($k=0; $k<($numpoints+1); $k++){
14         if($p1pnts>=$p2pnts){ //if p1 is greater than p2
15             $k1 = $k; //counter 1 remains
                as is
16             $k2 = $k/((($p1pnts*1.)/($p2pnts*1.))); //counter 2
                must be adjusted
17         else { //if p2 is greater than p1
18             $k1 = $k/((($p2pnts*1.)/($p1pnts*1.))); //counter 1
                must be adjusted
19             $k2 = $k; //counter 2 remains as is

```



```

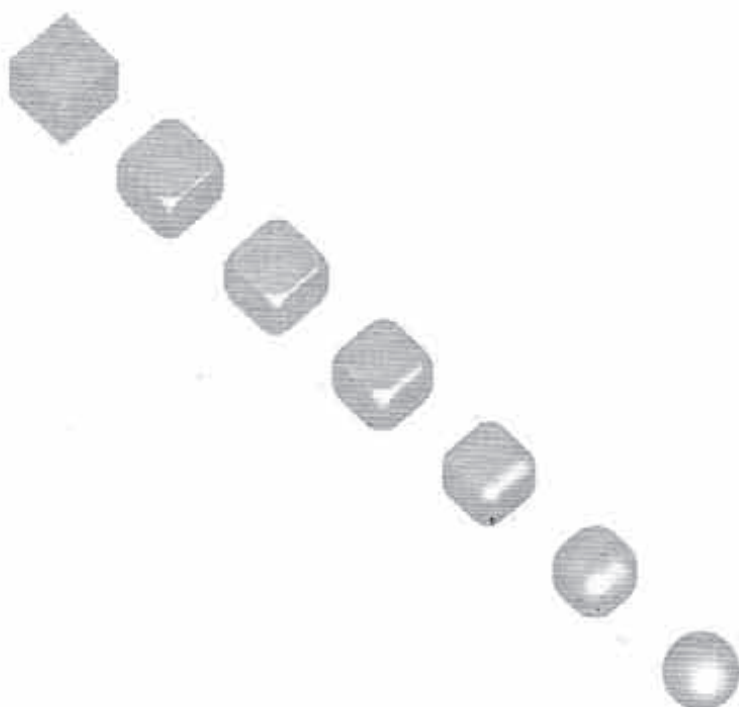
20      // interpolate the values of the child relative to the parents
21      $p1 = eval("pointPosition "+ $parent1 + ".cv["+ $k1 + " ]"); //get
      parent 1's points
22      $p2 = eval("pointPosition "+ $parent2 + ".cv["+ $k2 + " ]"); //get
      parent 2's points
23      for($j=0; $j<3; $j++)
24          $point[$j] = $p1[$j] + $ratio * ($p2[$j] - $p1[$j]);
25          $spoints += " -p " + $point[0] + " " + $point[1] + "
      " + $point[2]; //assign child pnts
26 }
27
28 eval("curve -d " + $degree + " " + $spoints ); //the child
      curve
29
30 }

```

The procedure *hybrid* takes as input the two parents' names and the ratio of interpolated steps. First, we extract the number of points of the two parents and their degree (and this case we're looking at curves). The number of points of the child object will be equal to the number of points of the largest parent. We then initialized two counters *\$k1* and *\$k2*, a string *\$spoints* to hold the curve values, and an array of floats called *\$point* to hold the points of the child object.

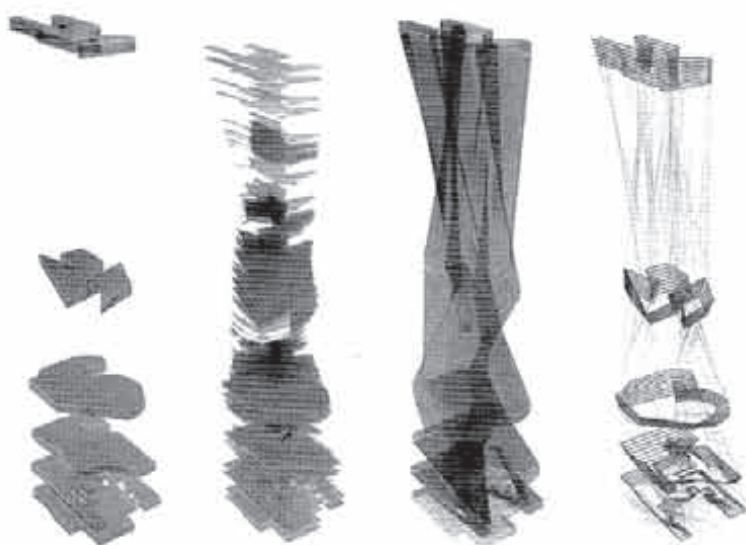
Next, we loop for all the points of the child object and determine the two counters: depending on which parent is greater we adjust one of the two counters. Since all counters are integers any division between them will be cast to the closest integer. Then, in line 21 we extract the points of its parent throughout the counters and then multiply by the ratio in order to obtain the points of the child object.

The examples in Figures 4.21 and 4.22 illustrate the pursuit for in-between hybrid objects.



4.21

The mid-hybrid object may be defined as either a *spherical cube* or a *cubical sphere*



4.22

Vertical transitions (class project by J. Paek and C. Santos for course GSD2311 taught by Kostas Terzidis in Fall 2005 at Harvard University)

Endnotes

⁴MEL stands for Maya Embedded Language. It is a way of instructing Maya to execute a series of actions through commands typed in an editor. It is also referred to as scripting. The difference between scripting and manual design is in the complexity and unpredictability of the actions. The human designer may be constrained by quantitative complexity and may be unable to construct unpredictability since that would negate a designer's intellectual control.

In Maya the script editor can be invoked by selecting **Window->General Editors->Script Editor....** A window with a divider should appear. The lower part (white) is where you type in the scripts and the upper part (gray) is the part where Maya responds to the scripts. In the menu bar at the top you can execute the scripts by selecting **Script->Execute** (or simply Ctrl-Enter). Help for each command can be found at the menu bar under Help.

A script is composed of variables, operations, and commands spelled and placed in a specific syntax. If the syntax or spelling is wrong, Maya will respond with a complaint (in the gray area of the editor). A free educational version of Maya is available at www.alias.com/maya

²The word *interesting* is derived from the Latin word *interesse* which means to be between, make a difference, concern, from *inter-* + *esse* (=to be). Interestingly, the in-between is literally interesting.