

University of East London Institutional Repository: <http://roar.uel.ac.uk>

This paper is made available online in accordance with publisher policies. Please scroll down to view the document itself. Please refer to the repository record for this item and our policy information available from the repository home page for further information.

**Editors:** Coates, Paul; Thum, Robert

**Title:** Generative modelling

**Alternative title:** Generative modelling Workbook

**Year of publication:** 1995

**Citation:** Coates, P. and Thum, R. (eds.) (1995) 'Generative modelling.' London: University of East London

generative modelling

---

This Generative modelling Workbook was designed compiled and edited by Robert Thum, UEL School of Architecture, under the direction of Paul Coates, course director MSc Architecture:Computing & Design.

It was prepared with the assistance of a bursary from the educational development initiative, UEL, as part of the MSc Architecture:Computing & Design course material for part time students.

### Acknowledgements

Introduction	Paul Coates Robert Thum
Autolisp	Robert Thum (stair,fern,Gasket) Paul Coates (Koch,Wander) Miles Walker (blocks Grammar, Latham)
Mini-Pascal	Andrew Howe(door, gasket) Paul Coates (tree,pythagoras,wander)
GDL	Phil Canon (table,tree,blocks) Barry Docker(blocks grammar)
Lingo	Janet Insull (navigation) Paul Coates (tree,koch)

Prepared using AutoCad™ 12,Archicad™4.5,MiniCad +™ 4.03,Macromind Director™4.0, Quark Express™,Photoshop™,BBedit.Published by UEL 1995 printed by UEL printing unit. All software running on Apple Macintosh™ computers.

## Warning to Children

---

Children, if you dare to think  
Of the greatness, rareness, muchness,  
Fewness of this precious only  
Endless world in which you say  
You live, you think of things like this:  
Blocks of slate enclosing dappled  
Red and green, enclosing tawny  
Yellow nets, enclosing white  
And black acres of dominoes,  
Where a neat brown paper parcel  
Tempt you to untie the string.  
In the parcel a small island,  
On the island a large tree,  
On the tree a husky fruit.  
Strip the husk and pare the rind off:  
In the kernel you will see  
Blocks of slate enclosed by dappled  
Red and green, enclosed by tawny  
Yellow nets, enclosed by white  
And black acres of dominoes,  
Where the same brown paper parcel -  
Children, leave the string untied!  
For who dares undo the parcel  
Finds himself at once inside it,  
On the island, in the fruit,  
Blocks of slate about his head,  
Finds himself enclosed by dappled  
Green and red, enclosed by yellow  
Tawny nets, endorsed by black  
And white acres of dominoes.

Robert Graves





## TREMOLETO (Tuscany)

I went to Tremoleto in the summer of 64. It was a sort of holiday and an assignment. The hired villa was actually a small entrance lodge that sat next to a large dusty building full of decaying eighteenth century furniture. I dutifully measured the large house, but the village was more interesting.

I took a series of black and white photographs all down the one and only street. The buildings were placed in a jumbled necklace of stone cubes, and each was different from the others. A photograph taken from one place would be entirely different from a photo taken just a few yards away.

I must say, I wasn't only interested in architecture, and it was on the last day that I quickly walked down the main street, taking photographs at random. Those grey washed out images were nevertheless a haunting reminder for years of an environment that seemed to have emerged from a well considered balance between the people and their building techniques. Not just there, but in Sienna, Volterra, and once you start looking, practically everywhere that was untouched by the 'designers'!

Vernacular architecture was better than the real kind, when it came to everyday stuff.

### A Bottom up Approach to Life

The idea that human settlements should have evolved suitable forms over many centuries was curiously absent from the textbooks available to a young architectural student in 1963. All the examples of good design were of plans attributed to some one or other, never 'anon'. This was not because the authors had never seen or enjoyed the traditional urban fabric of Europe (there was more of it around in those days) but because they didn't think it mattered. It seemed to me that it did. But nobody had any good ideas about how you made it. It looked as though nobody wanted to make it anymore anyway.

### Form as Process

Describing the actual form of something in terms of the processes that lead to its formation sounds like a simple enough idea, but it was not prevalent in the 1960's. There were many attempts to



2

control would have seemed lax to the mid 20th C developer. The failure of the conventional top down approach to control was that it didn't seem to correspond to the actual social organisation of the people it was supposed to be housing. What had happened was that ordinary people had surrendered control over more and more aspects of the design and layout of their houses and land, replacing the original loose control for more and more rigid hierarchy, until they became council tenants.

So the project was to try to demonstrate that this control could be transferred back again, to allow more room for individual decisions, and especially to try to define the control in such a way as not to predefine the actual geometry that might emerge.

My first attempt was a game that would try to eke out the forms of such buildings. It depended on a toy roulette wheel (subsequently substituted by a book of random numbers) a 16mm camera and fluorescent self adhesive green and red dots. (This was 1969 after all).

#### My Algorithm was

- 1• get a random number from the roulette wheel and say its the x or horizontal grid coordinate
- 2• get another random number from the roulette wheel and say its the y or vertical grid coordinate.
- 3• If there is room, stick down a luminous sticker at ( x y), unless doing so would completely block another blob, in which case stick down a piece of street.
- 4• have another go.

The resultant agglomerations were not very convincing, and anyway it took a long time to get anywhere near the density needed to start 'firing' the rules about overlapping and so on - rather like having to wait until everyone gets hotels in Monopoly.

This was time consuming, and seemed in need of automation.

#### The Idea of Computation

Is that we can define a series of steps which will guarantee the transformation of the subject from its current state to a defined new one after the computation. Computers were originally designed



not to be specific problem solving machines (like pocket calculators) but general purpose step followers. With most normal computations it is the end result we are interested in , but with the sort of computations that are possible with computers it is now possible to watch and record the process itself.

The process of computation , once observed sideways as it were, shows up all sorts of bumps and nonlinearities, and different states along the way . Conway's life game, and Mandelbrot's sets are both good examples of a view from the side.

So one might posit an ideal algorithm for a cluster of buildings and just watch the actual examples from the side as it were, as some of the many possibilities of actual arrangements of solid and space, back yards and porches, shops, workshops and houses.

Computers as anarchy makers. Discuss!

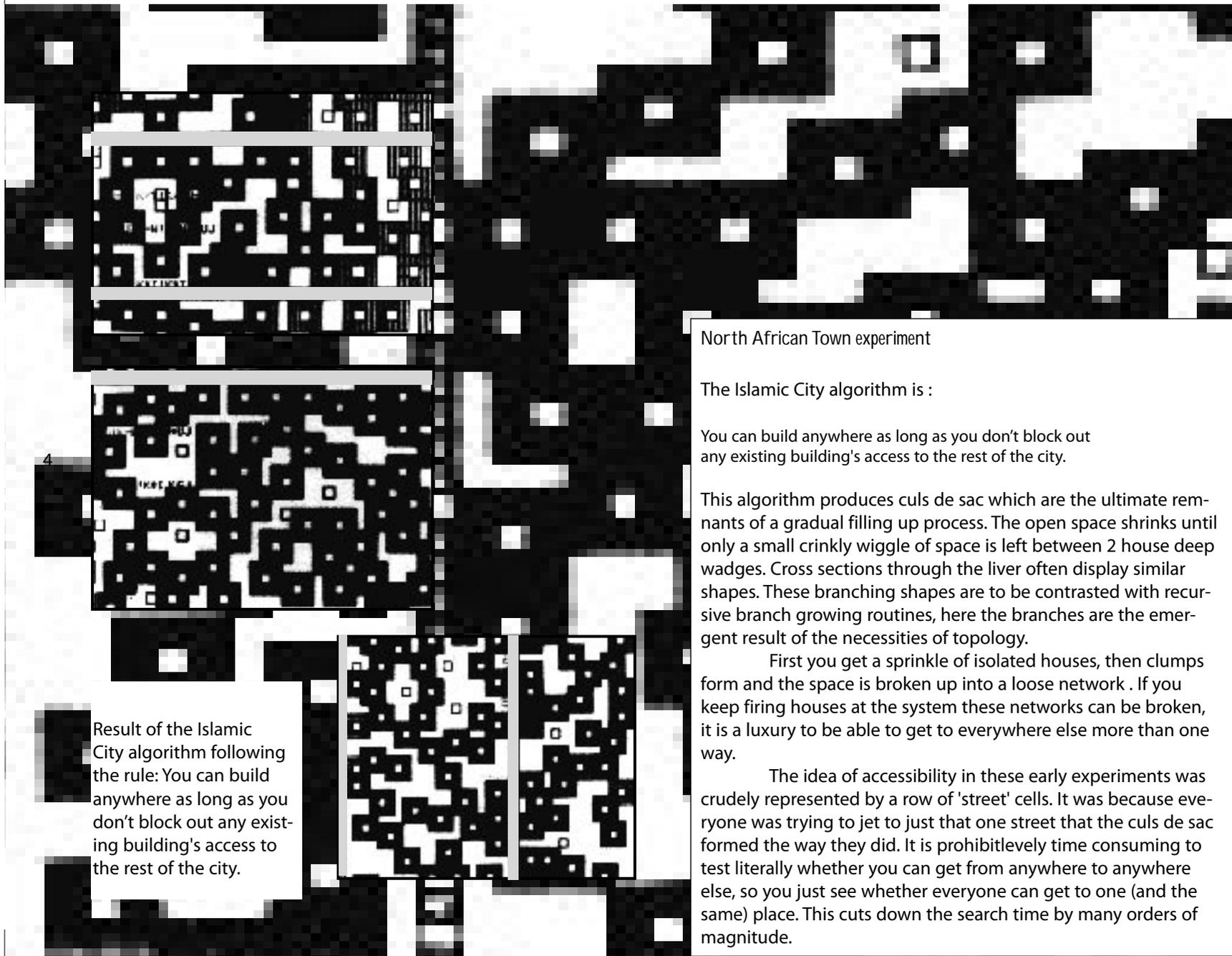
The Reductionist Versus The Expansionist Paradigms.

That is , it was proposed as a statement about one of the fundamental issues in mathematics, that of computability, and not as an amusing way to create patterns. Classical mathematics evolved in the days before computers, when the trick was to reduce the amount of arithmetic to the minimum by a process of logical compression. A mathematical statement is a very concise symbolic representation of a whole series of relationships.

This reductionist approach has been very successful, but there is a class of problems (to do with infinite series and other 'irreducible' sets)that simply can't be addressed. It seemed to Conway and others (Neumann & Ulam) that there was a lot of interesting stuff in there that classical reductionist maths was overlooking.

The key thing about their innovation was to explore the effects of applying some very simple transformation again and again along the potentially infinite series that arises from some initial set. It is no accident that ideas about cellular automata arose at the same time as the development of fast reliable logic machines. In fact the developments in computing theory that lead to the first practical general purpose symbol manipulating machines were reliant on the idea of the cellular automaton itself. Von Neumann is credited with inventing both the idea of the computer as we know it and the idea of cellular automata as an interesting area of study.

The rules for the transformation from state to state are necessarily local; no global rules can be invoked since there is no way



4

Result of the Islamic City algorithm following the rule: You can build anywhere as long as you don't block out any existing building's access to the rest of the city.

### North African Town experiment

The Islamic City algorithm is :

You can build anywhere as long as you don't block out any existing building's access to the rest of the city.

This algorithm produces culs de sac which are the ultimate remnants of a gradual filling up process. The open space shrinks until only a small crinkly wiggle of space is left between 2 house deep wedges. Cross sections through the liver often display similar shapes. These branching shapes are to be contrasted with recursive branch growing routines, here the branches are the emergent result of the necessities of topology.

First you get a sprinkle of isolated houses, then clumps form and the space is broken up into a loose network . If you keep firing houses at the system these networks can be broken, it is a luxury to be able to get to everywhere else more than one way.

The idea of accessibility in these early experiments was crudely represented by a row of 'street' cells. It was because everyone was trying to jet to just that one street that the culs de sac formed the way they did. It is prohibitively time consuming to test literally whether you can get from anywhere to anywhere else, so you just see whether everyone can get to one (and the same) place. This cuts down the search time by many orders of magnitude.

of defining the 'end product'(if there were, then the reductionist approach would have worked anyway). The only global rules are artificial restrictions on the size of the series, based on the limitations of the finite machine the expansion is being calculated on. The 2D life game is a development of the one dimensional example, where the initial seed is a row of ons and offs - like a binary number. The next number is created by a rule which substitutes 0 and 1 based on the immediate neighbours of the row above. In the life game the initial 'number' is a two dimensional array of cells, either alive or dead ( 1 or 0). The rules are:

If a dead cell has two alive neighbours then it becomes alive;

The life game was invented by a mathematician,  
not a textile designer.

If an alive cell has three or more neighbours it dies

If an alive cell has less than 2 neighbours it dies

Once you run the life game on a computer you can explore a long way along the series. Some surprising 'end results' were:

Stable states      Binary or n cycle stable states

Gliders              Eaters  
Guns                 Puffers

The key fact here is that simple local rules can create complex global outcomes. Randomly seeding the universe with cells usually creates (perhaps fleetingly) most of these examples of emergent behaviour.

In the classical world view you take a disparate and complex

situation and attempt to reduce it to a set of expressions, as terse as possible, thus cutting out tedious and error prone iterative calculations. This holds good for mathematics, Architecture, music.

In this new world you take a simple function and expand it into a disparate and complex situation - exactly the opposite. Mathematicians have been using this paradigm since the 1950's, musicians are able to dispense with notation, but architecture is still stuck in the classical reductionist mould (see any constructive geometry based modeller).

The expansionist paradigm has provided us with a way of looking at form not as a 'given' but as emergent. Computers provide us with the tools to carry out endless experiments.

Paul Coates

The purpose of this book is to introduce the idea of computer generative modelling as a means of exploring key ideas in the design of the built environment, by way of a series of worked exercises in 3 languages (GDL, MiniPascal and AutoLisp) which go beyond the standard customising issues to allow experimentation with the automatic generation of form, or generative modelling.

These examples are intended to do two things:

### Birth

1) Explain the basic technical issues, assuming some use of introductory material provided by the software supplier.

3) Set out some useful concepts for exploring these ideas, such as recursion, random numbers, shape grammars, cellular automata.

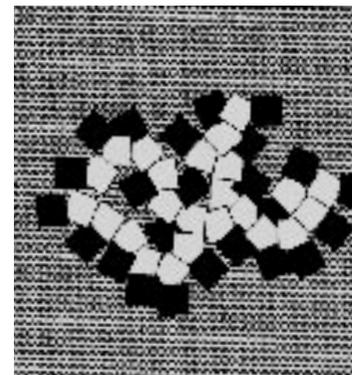
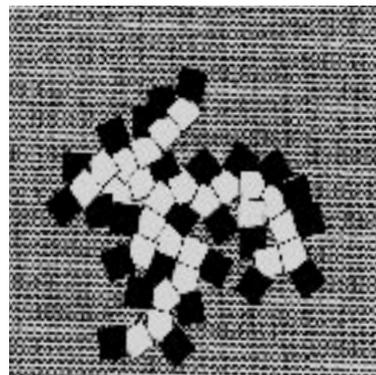
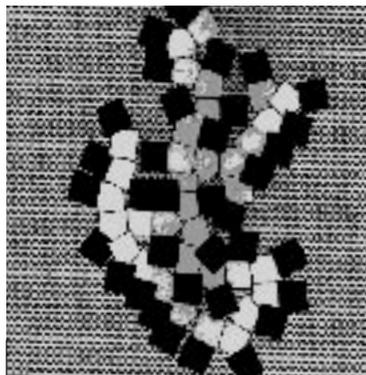
Essential to the theme of the book is the idea of an algorithm, and ways of implementing algorithms on a Cad system. The exigencies of computers today means that to get something done we have to make use of a computer language and do some coding, but it should be remembered that the code is just a means to an end.



The Alpha syntax village generator

This is based on Bill Hillier's original idea of a three state automaton which attempts to capture the essence of 'unplanned' organic village growth in Europe.

There are three types of cell, X,Y, and 'nothing'. (unlike the life game where there are two types - 'alive' or 'dead'). The idea is that there are two types of space, closed private 'inside' space and open public 'outside' space. They are related such that each bit of private space is related to a bit of public space



This is based on the natural assumption that you need to be able to get to your front door.

The next rule is that each bit of public space is connected to all the others.



The result of running the system is the gradual development of a network of open space elements, with 'houses' attached. The original version (illustrated in 'The Social Logic of Space') was cellular, and all X and Y spaces were the same size. In Bill Erickson's version the x and y space are square, but can be at varying orientations. Running the model generates a set of connected objects, which can be seen as 'connection space' and buildings in 3D.



explain 'settlement patterns' and 'urban fabric' and 'village types' but the form creating processes were always non geometric. Where geometry was invoked was once definitely 'planned' arrangements were under discussion, with observable (and planned) geometrical properties (squares, crescents facades, colonnades).

The new idea was that there might be geometrical rules that constrained and informed the eventual arrangement of 'unplanned' architecture as well as the planned sort. We thought that if you could determine these more subtle determinants, then perhaps a better understanding of how to add to and develop existing examples might develop. It was (and still is) a kind of particle physics of Architecture.

It seemed obvious that the outcomes observed were the result of the interaction of many forces, each of which contributed to the end disposition of the houses. The lie of the land, the size of the available timbers,(for floors and roofs). But there was no obvious way of analysing this apparently pleasing arrangement, no way of understanding not just this but large segments of most of the urban spaces of the region, including Sienna and Volterra.

The failure of the top down approach

What seemed the most telling comparison for me was to contrast the spatial complexity and unending opportunities for new spaces and places to be, in the most rudimentary village, with any 20th century housing that I knew of. There was no contest: on the one hand complexity, variety and surprise, on the other sameness boredom and banality. The problem seemed to be that increasingly large amounts of cheap urban fabric were being designed by increasingly smaller and smaller numbers of people. Inevitably they decided to just do a little bit, and repeat it over and over again. This contrasts with the bottom up way the existing arrangements had evolved, where the arrangement of individual bits of the urban landscape had been the result of many people's decisions over time.

Bottom up Anarchic Organisation

Anarchy, when applied to the circumstances surrounding the development of Italian hamlets, must be assumed to be something less than the bomb carrying sort, but anything less than total con-



## Determinism versus Non-Linearity

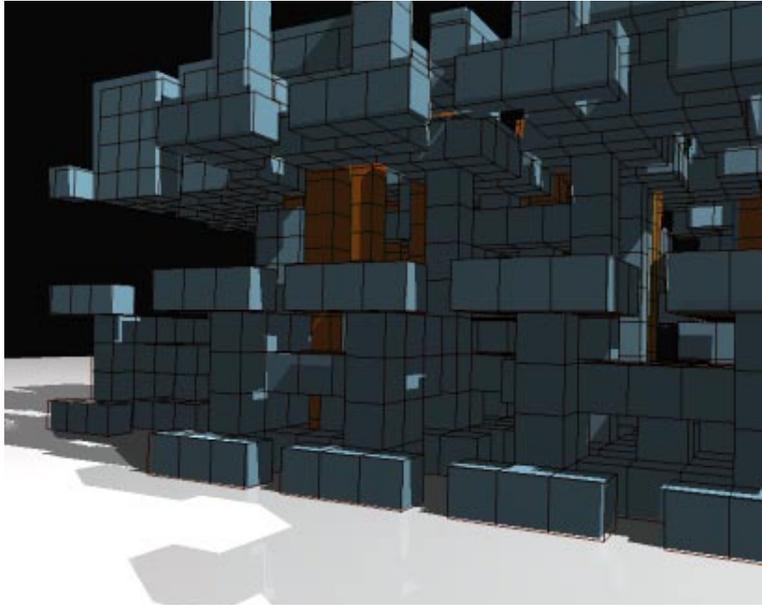
8

The hallmarks of the new science - fractals, chaos, and complexity- are regulars in our media. Their colourful beauty and bizarre shapes did not just intrigue scientists. Their significance goes beyond the thrill of being strange mathematical creatures. They subversively soften our traditional understanding of the world, our way of thinking. Their importance reaches beyond the realm of technology and science, into our most fundamental social, moral, and philosophical beliefs. 'It will force us to re-examine our place in the universe and our role in nature.'(Langton, *Artificial Life*) In order to reach that point it was, however, necessary to first break the mighty dogma of the Newtonian era: the universality of the natural laws and determinism.

During the Newtonian era the dominating concept in science was its ability to relate cause and effect. On the base of the natural laws, it is possible to calculate the trajectory of a bullet or predict astronomical events such as the collision of the Schumacher-Levi comet with Jupiter. On the other hand there are other natural phenomena that seem to be unpredictable and chaotic. A classical example is the weather. Innumerable and unsuccessful efforts to calculate the weather lead to speculations that it is governed by randomness. Yet, there was a strong belief that in general it was

quite possible to calculate a reliable weather forecast, with the help of high performance computers and a dense network of weather stations. Some of the first conclusions of complexity theory, however, dramatically altered this point of view. Simple deterministic systems are able to create a random behaviour. And this randomness is system inherent. More data or closer look don't make it disappear. Random behaviour created by a deterministic system that is generated by rules, which themselves do not contain any element of chance, seems to be a paradox. In principle the future is completely determined by the past. But in reality small uncertainties such as the slightest errors of measurement enter into calculations and are amplified with the effect that although predictable in the short term, it is unpredictable in the long run.

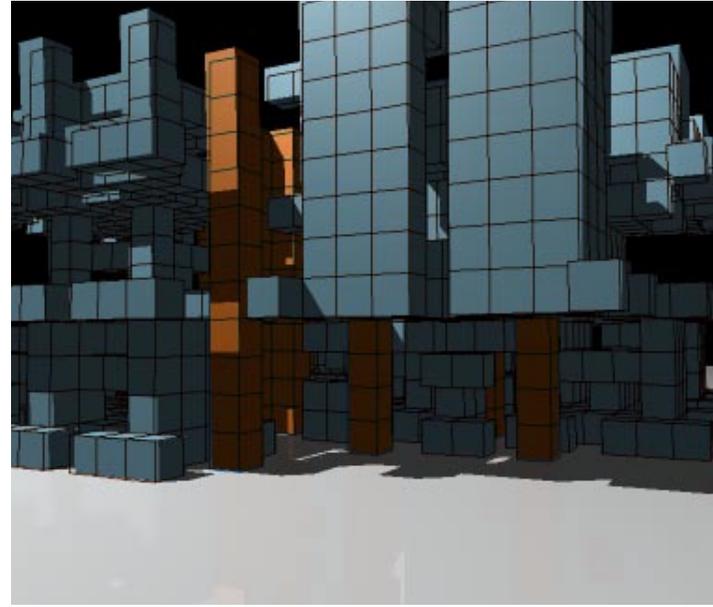
For all that time the terms determinism and predictability were equivalent. The Parisian mathematician and astronomer Simon de Laplace captured the credo of the era of determinism most vividly with what it today known as the 'Laplace demon.' 'If we can imagine a consciousness great enough to know the exact locations and velocities of all the objects in the universe at the present instant, as well as all forces, then there could be no secrets from this consciousness. It could calculate anything about the past and future from the



laws of cause and effect.” (Peitgen, Chaos and Fractals)

The metaphor dominating this understanding of the world, is that of a tremendously precise running clock, where the present state is simply a result of the previous state and the cause for its future state. Present, past, and future are tied together by causal relation. The problem of exact prognosis is just a matter of the difficulty to collect all the relevant data with the right precision. In this sense reliable weather forecast was thought to be achievable through a tighter net of weather stations and massive computations. The maxim of classical natural science was shattered through the insights of Werner Heisenberg in 1927 formulated in his ‘uncertainty principle’. This principle states that it is impossible, even in theory, to determine the exact position and velocity of an object simultaneously. This is true for systems of any size, but for systems of ordinary size the uncertainties are too small to be observable. The importance of the principle lies in systems of subatomic level such as the motion of electrons.

Heisenberg wrote: “In the strict formulation of the causality law - ‘When we know the present precisely, we can calculate the



future’ - it is not the final clause, but rather the premise, that is false. We can not know the present in all its determining details.”

This was the first important blow against determinism. The other came 30 year later when Ed Lorenz discovered the crux of numerical weather forecast, popularised by the term ‘butterfly effect’, which is a metaphor for the exponential progression of errors. Lorenz has shown through his studies that the conclusion of the causality principle - the ability to calculate the future- is also wrong. Natural laws and therefore determinism do not exclude the possibility of randomness or chaos. A system that is precisely determined is not necessarily predictable. And this is true for far simpler systems than the weather. It can be observed in very simple feedback systems such as the iteration of the quadratic equation of

$$x \rightarrow x^2 + c$$

To visualise this system’s behaviour the resulting points of each iteration are marked on the complex plane. Depending on the initial value of  $x$  and the value of  $a$  it results not just in a random chaotic behaviour, but more importantly in ordered patterns of unpredictable complexity. Ed Lorenz’s and Werner Heisenberg’s deconstruction of the causality principle did not just bring true chaos and random-

## Feedback and Self-referential systems

---

10

The difference between traditional and new science manifests itself in the difference between linearity and non-linearity. It seems inevitable to explain linear and non linear systems.

A linear system can be sufficiently described by the behaviour of its constituent parts. In more scientific terms: they obey the superposition principle. That allows the following method of analysis: the linear system is broken down into smaller parts. To get an understanding of the system as a whole these parts in isolation become subject of study. Once all the parts are known in their character and behaviour, they are reassembled and provide a thorough knowledge of the complete system.

Nonlinear systems, in contrast, do not obey the superposition principle. Their primary quality lies in the behaviour based on the interaction of their constituent parts, rather than being property of the parts. Breaking the system down and analysing the parts in isolation does not enlighten the system's mechanism. In fact, taking the system apart can cause the loss of its characteristic behaviour. Therefore the interesting properties of nonlinear systems are emergent. That is, the global behaviour is a result of short range interactions of many identical parts on a local level. Unlike the behaviour of linear systems, their behaviour exhibits nonintuitive traits, it can not be anticipated.

'Life is a property of form, not of matter, a result of the organisation of matter rather than something inherent in the matter itself.'(Langton, Artificial Life) In Chris Langton, from the Santa Fe Institute for Non Linear Studies point of view, a living system is a sophisticated non linear system. Amino acids and other carbon chain molecules are the systems components. Since none of these are alive the understanding of these parts tells nothing about their most interesting behaviour: Life. Life is the emergent behaviour on the system's global level created by local interaction of lifeless molecules. Langton concludes: 'Behaviours themselves constitute the fundamental parts of nonlinear systems- virtual parts, which depend on nonlinear interaction between physical parts for their very existence.'(Langton, Artificial Life) These virtual parts are in the centre of the study of nonlinear systems. But virtual parts exist in different grades of complexity.

'Life' or life-like-behaviour, the virtual parts Chris Langton and his fellow ALifers are searching for, are on the top of the list of possible behaviours of non linear systems. As a matter of fact, all the possible behaviours can be assigned to three classes of behaviours.

The crystal state, the liquid state, and the gas state.

**Crystal class:** the system evolves into a finite state, a constant pattern or form, some more interesting and complex than others. The single parts are capable of organising themselves in very sophisticated structural patterns. A snowflake is an example for that class. No central force is assembling the water molecules, nor do they carry a blueprint of the overall form. Patterns on a large scale emerge entirely from local interactions of identical molecules. The water molecules can configure an endless variety of different shaped snowflakes depending on their initial condition. Interaction on local level makes such self-organised structures extremely adaptive to different environments. They inherit the ability to grow around obstacles in an ordered and 'intelligent' way.

**The gas class:** the system evolves into a random and chaotic behaviour. It is not dependent on the initial state, even a highly ordered seed eventually ends up in cycles of random noise.

**The liquid class:** its the most interesting one, systems evolve into structures of substantial spatial and temporal complexity. Langton believes living organisms are example of such systems. But even very simple systems such as the Conway cellular automaton based 'Game of Life' shows complex 'life-like' behaviour. Out of a few transition rules a whole world of organisms emerges: constant configurations, moving, eating configurations, producing ones, and so on. An endless variety. Hierarchies are created in a seemingly purposeful behaviour. Society or flocks are examples of such nonlinear systems. Their hierarchical structure and behaviour as a complete organism emerges from rule based interaction of individuals. A good illustration is the simulation of flocking behaviour of birds by Craig Reynolds, a computer scientist working at Symbolics, a graphics hardware manufacturer. In a virtual environment a large number of autonomous but interacting objects - Reynolds calls them 'Boids'- move around in a highly ordered manner. (Kelly, Out Of Control) Obstacles are no problem for the flock of Boids, they split up into sub-flocks, which reassemble themselves in a similar way to the whole, move around the object and reunite into the original formation. Can this be called an intelligent behaviour?

The observation of such behaviour calls for the need to redefine the term intelligence or intelligent behaviour. The intelligent behaviour of the Boids is not the result of reasoning. The Boids do

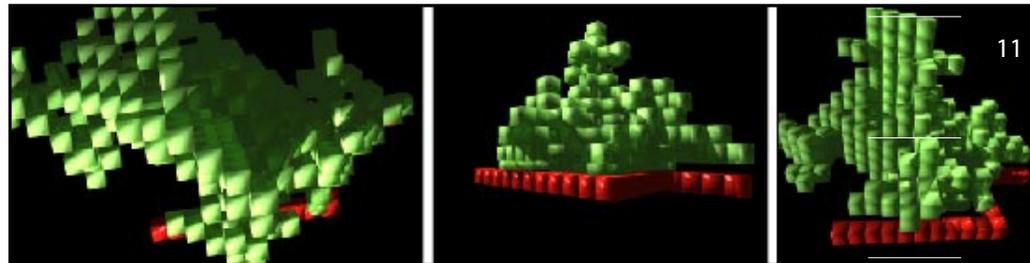
not have an understanding of their environment, the notion of an obstacle is alien to them. A single Boid, like a bird, while airborne has no overarching concept of the shape of its flock. Moving in formation and splitting up into organised subflocks happens without thought. This intelligence is an emergent property. If we consider the rules that each Boid follows:

1. Maintain a minimum distance from other objects in the environment, including other Boids.
2. Match velocities with Boids in your neighbourhood
3. Move towards the perceived centre of mass of the Boids in your neighbourhood,

we can not, however, define this property.

These are the only rules governing the behaviour of the flock. "Flockness" emerges from creatures completely oblivious of their collective shape, size, or alignment.

The Boids represent a biological system with the capability



of organising itself into adaptable and efficient structures that can adjust to different environments; buildings and urban structures embody the architectural system of an almost equal complexity. Learning from the biological system, it could be argued that it is possible to reinterpret what is perceived as difficulties, hindering the design process, into the constituent forces of a complex system. The design of the urban structure or buildings becomes then the emergent property of the system. Instead of painstakingly designing in a top down manner, architects could use the emergent 'cleverness' of such a system and observe the design constructing itself in an effortless process. Equipped with rules the raw matter would therefore organise itself in the most sensible and appropriate way.

But before we can experiment with complexity and model

## Thinking algorithmically

If you have never written a computer program before, then you will be unfamiliar with the idea of instructing a machine to think for you. We all solve problems and do it with a greater or lesser degree of insight, but to make a computer do some thinking, we have to abstract the basic building blocks of problem solving and see it as something 'out there'.

The concept of the algorithm is the one we must grasp, in order to get our minds in gear to start programming in any language. Before you can make a computer do anything, you have to give it some instructions. These instructions have to be given using a language.

All computer languages consist of just a few basic elements, which are combined to express a method for completing some task. They are artificial languages, which have their own SYNTAX and LEXICON (method of combining words, and a dictionary of understood words).

When you combine these words into a set of syntactically correct expressions, so that the task is actually accomplished, you have

12



Self-organising structure based on a three dimensional cellular automata. Generated with AutoLisp and rendered in Strata Vision.

defined an algorithm.

An algorithm is a set of instructions which, when followed according to the conventions of the language guarantees the correct execution of the problem. A set of instructions are usually written down as a program which the computer obeys. We can imagine such a set of instructions as:

- Clear the screen
- Draw a line
- Make three beeps

In any program there is always an implicit flow of control which determines the order in which these instructions are obeyed. In the simplest languages this flow of control starts at the first line, and continues, line by line until the last line is reached, just like reading a text. If a program (such as the one above) is written in this way, it will always do the same thing, and if you think about it, it will

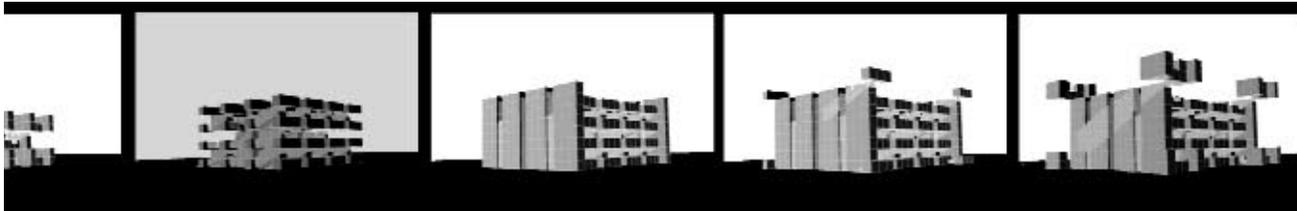
always produce the same result.

1. Clear the screen
2. Draw a line
3. Make three beeps

Usually programs will require some input from the outside world, and will do some calculations on these inputs, so that it makes sense to have a way of representing values within the program. These are known as variables - because the contents are variable, from one run of the program to another.

If a program doesn't contain any variables, it will always give the same result, and as a consequence it will be pointless to use it more than once. This is stupid - why go to the trouble of writing a program, just to use it once?

Variables allow the program to do work on a variety of values, and thus produce a variety of outcomes. Often in a program we need to make decisions and decide to do one thing or another depending on the values of variables, we also need to do some manipulation of the values.



The main elements of any language which allow us to do this are:

- Conditional statements

```
IF <something is true>      THEN do this
                           ELSE do that
```

A conditional statement can be thought of as a switching point.

If an algorithm contains a conditional statement, then the program can do at least two different things, depending on the values of its inputs. In the above IF - THEN - ELSE statement the stuff in brackets <something is true> is known as a conditional expression, which takes the form, usually of some comparison such as "is value A less than value B". Each language has its own syntax for conditional expressions, but whatever it is, the result is always either TRUE

or FALSE.

- Control statements.

The flow of control, which in the simplest case just trundles down line by line reading a text, can also be altered by breaking the whole program down into small chunks, which you can choose to do in different orders, depending on the conditions that obtain. For instance, if we break up the algorithm into several functions we could say

Introduction

```
IF ( Louis Kahn )      THEN Generate concrete towers
                       ELSE Generate wooden shacks
```

where "Introduction", "Generate concrete towers" & "Generate wooden shacks" are three separate functions, coded as three chunks of text. In this case, the introduction would always be executed, but

the rest of the process would depend on the result of the truth or not of the assertion "Louis Kahn".

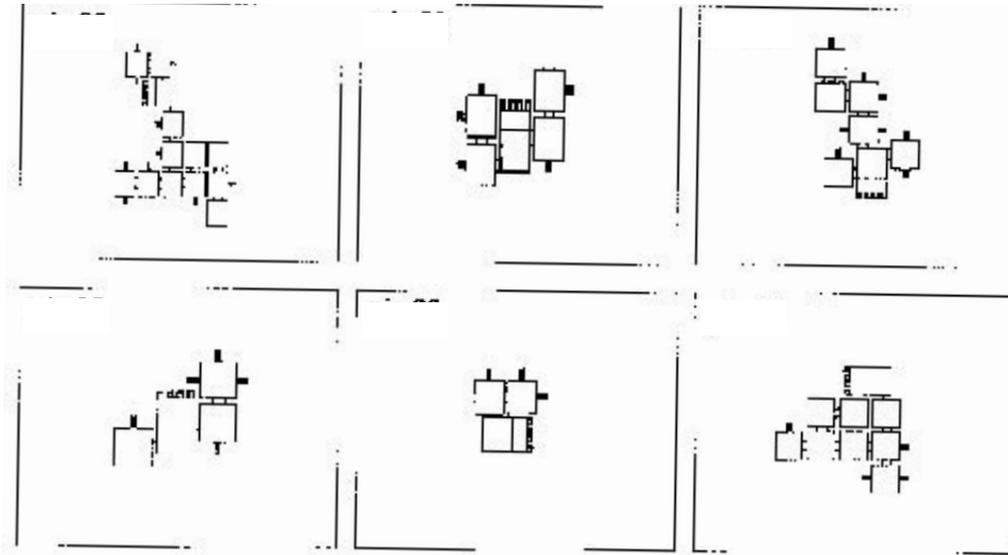
The main other control statements are to do with LOOPING - ie. doing something over and over again. Most languages provide constructs such as REPEAT ...UNTIL, WHILE <something is true>. Basically a

loop provides a way of reducing the amount of code you need to write, since you only need to write it once, and the program will cycle around reading & obeying it over and over again. Recursion is a special case of looping, where a function calls itself - more on this later.

With any repetitive process we have to define an end condition which defines the terms under which the process will end, otherwise the loop will go on for ever!

- Arithmetic/Geometric statements

These are needed to do sums and geometric calculations, so as to manipulate the values to solve some problem or other. Most languages can add, subtract, divide, multiply, and a range of ready made functions are usually available for things like square roots,



cosines, tangents, minimum & maximum etc. etc.

14

### The Special Features of Built in Scripting Languages

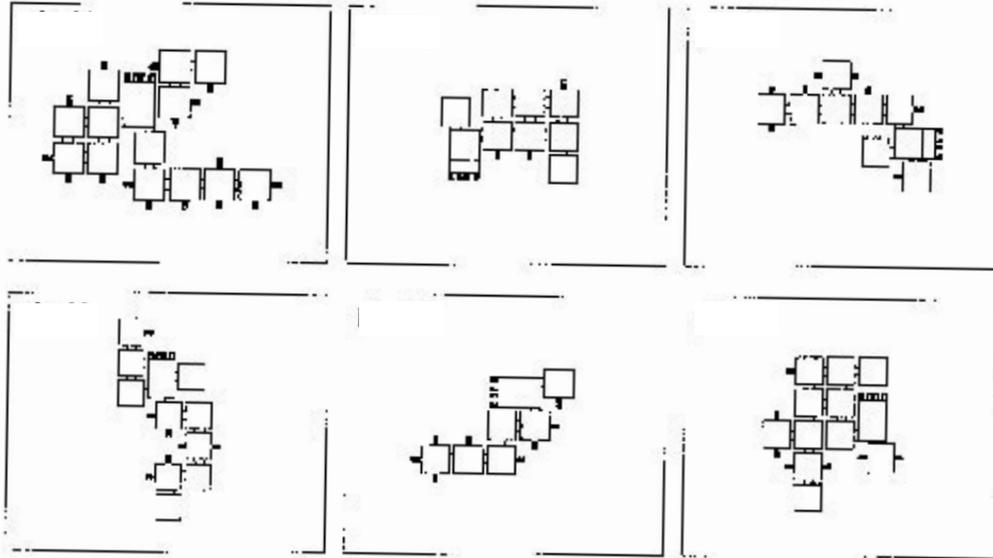
All computer languages have these components, and the ones we use are no exception, but in addition the built-in scripting languages of ArchiCad, MiniCad, and AutoCad have had a wide variety of additional functions built on top which makes them much more useful for creating form. These extra facilities come in two flavours;

- 1) Geometric manipulations
- 2) Access to the 3D database.

These two aspects are interrelated, since being able to do geometry in the abstract is useless, what we need is the ability to do geometry to something which the language knows about. The languages illustrated in the following sections (Autolisp, MiniPascal and GDL) are embedded in their respective modelling packages, and provide ways of searching and/or creating 3D lumps of stuff. In order to do this 'algorithmic production of form' thing, we not only have to design algorithms, but also have the ability to produce form. Using a 3D modeller means that the technical problems of building and maintaining a 3D database are taken care of by the software itself, and we only have to worry about the algorithms.

The geometric manipulations have to do with moving around

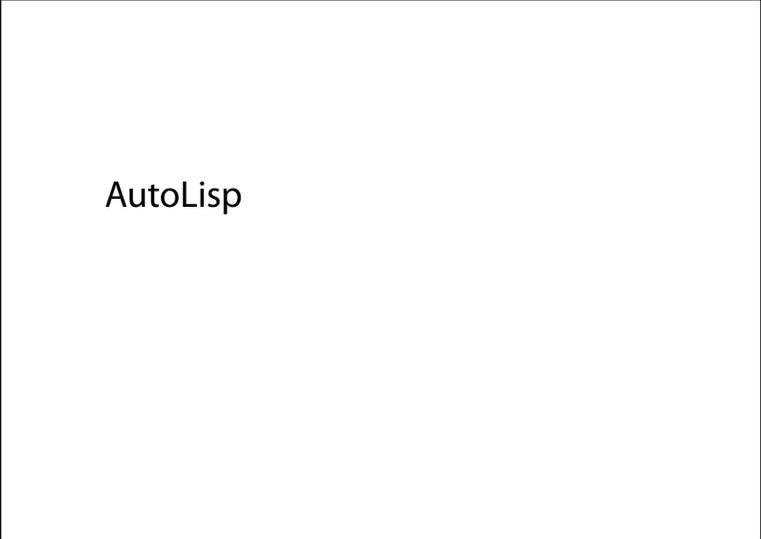
Rule based Kahn-like  
architecture genera-  
tions by Colin Wong.



a 3D world in XY and Z coordinates, defining measuring systems and calculating positions in space. These are necessary operations, but to

have any degree of control over the evolving form, we also need to know what objects already inhabit this space. To do this we need to be able to make enquiries about the number, type and location of forms. This allows us to make up rules that can express ideas such as 'on top of', 'between', '2 Meters from', 'opposite' and so on. It is one of the most difficult tasks of the scriptwriter to be able to express such concepts in a way the underlying software can understand, and it is currently quite cumbersome using the software at our disposal. The idea of the 3D database is also something which the normally implicit human based thinking finds hard to grasp - why can't you just say 'between the altar and the east door' if that is what you mean? Essentially the problem is that, while the modeler can remember and display the lumps of stuff, it can't on it's own invest any one of such lumps with a meaning. We automatically do this just by looking ( we can see from the display that the dome is upside-down, but the computer needs to be taught what that means). Initially all three dimensional objects 'inside' the computer are assigned to just a few broad classes, such as cuboid, polyhedron, surface of revolution and so on. In any complex model there will be hundreds of such objects. One way to give added meaning to an object is to give it an unique name - quite literally in some cases, by implication in other languages. Naming objects according to some generative rules allows us to 'know' more about them when we





AutoLisp

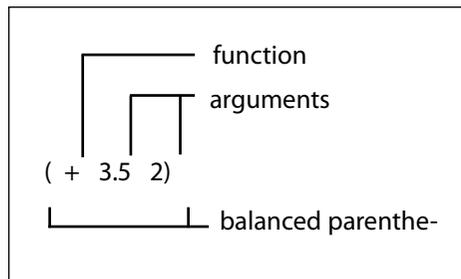
We will give here just a brief introduction to AutoLisp. You will find a more detailed introduction to that language in the AutoLisp Programmers Reference or in one of the many well written introductory books on the language, such as The ABC of AutoLisp by George Omura.

One reason for AutoCad's popularity is its adaptability. Due to its transparent architecture just about every aspect of AutoCad's operations can be controlled. The software is therefore open for customisation to suit specific needs and allows full accessibility by Third party developers. The centre piece of this adaptability is Autocad's built-in programming language, AutoLisp. AutoLisp is derived from Common Lisp, an updated version of the oldest artificial-intelligence programming language and is considered to be an easy to learn programming language because of its simple syntax. With AutoLisp people can write their own commands and redefine the existing ones.

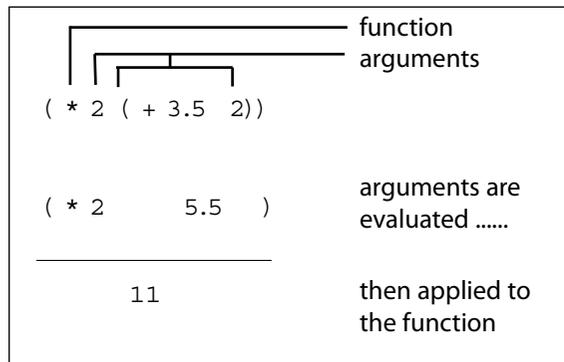
AutoLisp can be used as macro-building facility (Macros are scripts that automate sequences of repetitive keystrokes) or as a software development environment that allows you to devise complex programs, such as programs for the generative creation of form.

18

This is an example of an AutoLisp expression, the fundamental AutoLisp structure consisting of a function and a number of arguments (here two).



Example of a nested expression. AutoCad evaluates the arguments first and then applies them to the operator or function.



## The Syntax of AutoLisp

AutoLisp evaluates expressions. For instance, if you want to add the numbers 3.5 and 2, you must enter the formula as followed on the command line:

(+ 3.5 2)

AutoLisp returns the value of the expression on the command line. 5.5. Everything, from the simplest expression to the most elaborate program, must be written with this structure. Or in more general terms an AutoLisp expression must include an operator, such as the mathematical operators +, -, \*, /, followed by arguments. All AutoLisp expressions are enclosed by parentheses. Important: Notice that you need a space between an operator and each individual argument to distinguish them for the AutoLisp interpreter. (To visualise the space we use Δ) The expression above with visualised space:

+Δ3.5Δ2Δ

This is the correct way to insert the expression. You get an error message if the expression is typed in like this:

(+3.5Δ2)

No space between operator and the first argument causes confusion: the term +3.5 is seen as an unknown operator for the AutoLisp interpreter. In contrast it is permitted to use more spaces than necessary:

(Δ+Δ3.5Δ2Δ)

is correct, as well as

(+ΔΔΔΔ3.5Δ2Δ).

## Using nested expressions

AutoLisp evaluates everything inside the parenthesis. It checks the value of each argument, for example for the number 3.5, 3.5 is returned unchanged. Because of that expressions can also be used as arguments:

```
(* 2 (+ 3.5 2))
```

Here, the multiply function (\*) is given two arguments. The integer 2 and the expression (+ 3.5 2). This type is called a nested expression, because one expression is contained within another. AutoLisp evaluates the nested expression first and then applies the resulting value together with the other argument to the function of the containing expression. A program is a set of expressions that perform a task when executed from the AutoCad command line. But they can also be considered user-defined functions. Defun is the special function that allows you to define your own programs and functions. Defun is treated like any other AutoLisp expression: It is enclosed by parentheses and it has arguments. The first argument is the name of the function, then a list of variables followed by expressions. Usually the name has the form c:Name, the c: tells AutoLisp that this function is to act like an AutoCad command. If the program name is entered at the command line it will be executed as an AutoCad command.

Important: Avoid giving your programs names reserved for AutoLisp's built-in functions. Otherwise you will override the built-in functions so that you are not able to use them any more. Restarting AutoCad restores the built-in functions.

## Data types

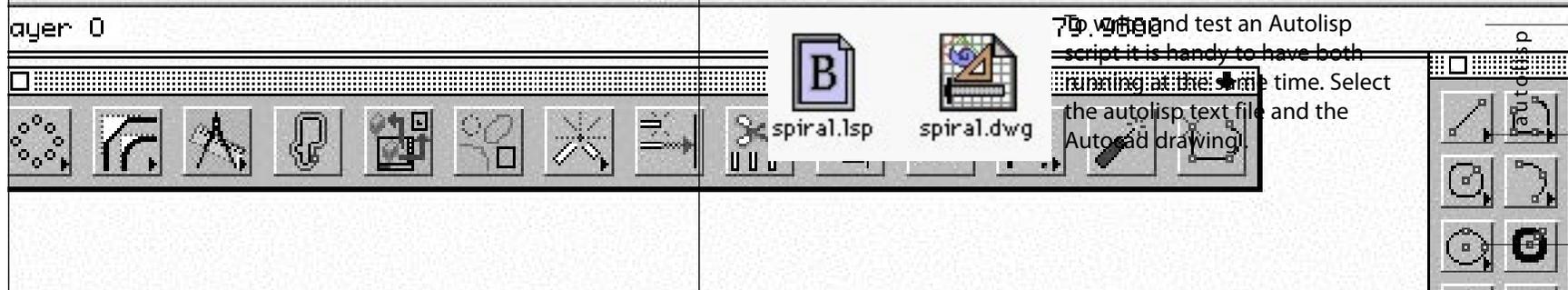
Variables used in your program can be of different types of data. In contrast to other programming languages, such as Pascal, AutoLisp doesn't require up-front definition of the type of variables you are using in your program. The text box shows the most common data types in AutoLisp

Integer	38	a whole number used for counting things
Real Number	7.5463	a number with a floating point such as 3.14 used for coordinates and dimensions
String	"Paul"	a series of characters, enclosed by double quotes
List List bers,	( 23 74 59)	data elements, such as numbers, points, list .....enclosed in parentheses
BOOLEAN	t	Just TRUE or FALSE - used in logical expressions as part of a conditional statement
Selection set	<selection set: 5>	a group of objects



Quickstart Edit Modify Dimension View Settings ASE Render Model Special ?

robot's drive:Applications:AutoCAD Release 12: titled



To write and test an Autolisp script it is handy to have both the text editor as well as AutoCad running at the same time. Select the autolisp text file and the AutoCad drawing.

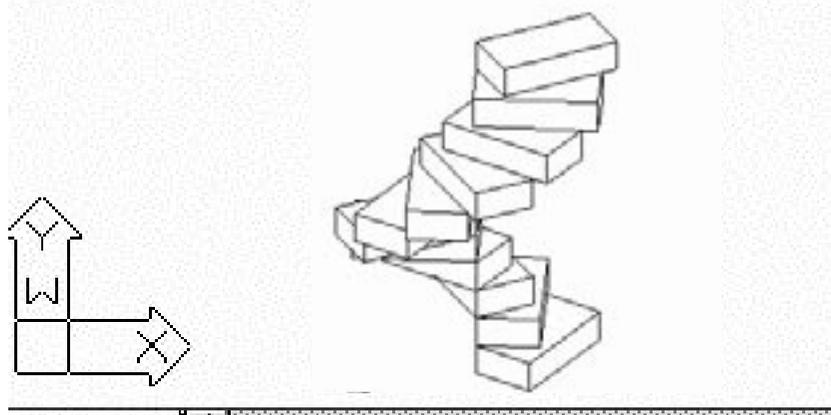
The actual scripting happens within a text editor such as BBEEdit- the "bare bones" editor. It is handy to have both the text editor as well as AutoCad running at the same time, so you can write and test an AutoLisp script. To store an AutoLisp program you save it as a text file with the extension .LSP. Make sure that this text file is stored in the AutoCad folder otherwise AutoCad will have difficulties in finding it.

```

spiral.lsp
Last Saved: 29/11/94 10:54:09

(defun C:sp ()
  (setq "codecho" 0)
  (command "erase" "all" "")
  (setq angle 0
        n 0)
  (repeat 12
    (progn
      (setq p1 (list 0 0 n)
            p2 (list 2 4 (+ 1 n))
            thing (solbox p1 p2 ""))
            n (+ n 1))
      (command "rotate" thing "" p1 angle)
      (setq angle (+ angle 30))
    )
  )
)

```

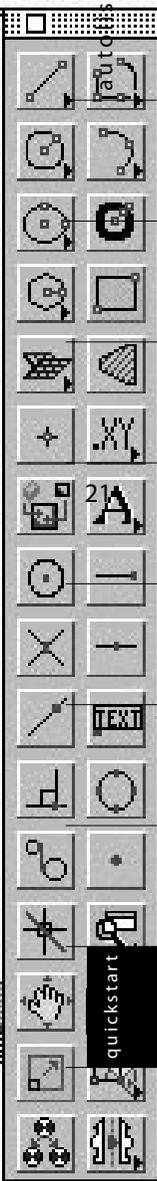


Help

Command: \*Cancel\*

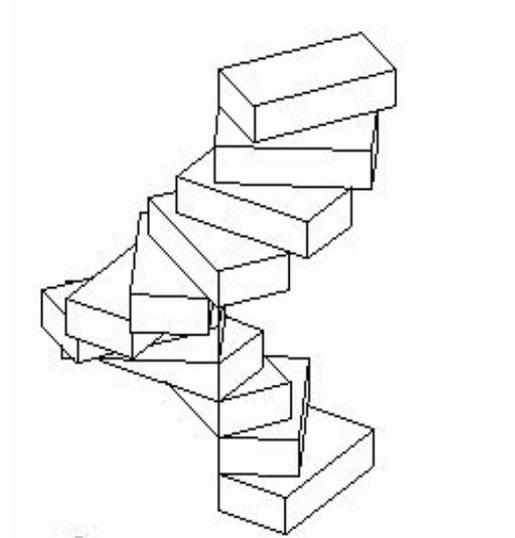
Command: Regenerating drawing.

The AutoLisp program is stored as a text file. Save your BBEEdit file with the extension .LSP -in this example spiral.lsp- and put it in the AutoCad folder. To run the program type (load"spiral") in the command line and press return.



quickstart

## 1.0 Spiral



- 22 The first program draws a simple spiral staircase by stacking solid boxes on top of each other with an constantly increasing rotation angle. Before you start AutoCad and a text editor (BBEdit) should be open. Additionally you have to activate AutoCad's solid modeller AME within AutoCad.

The algorithm:

### 1. Housekeeping - delete all objects from drawing

```
(command "erase" "all" "")
```

erase is an AutoCad command. You can incorporate AutoCad commands in your AutoLisp script by using the command function, the name of the AutoCad command in double quotes, followed by all the inputs and key strokes usually typed on the command line. Using the erase command directly on the command line AutoCad prompts Select objects. To clean the drawing you type all and AutoCad prompts the number of found objects. Pressing the return button will execute the deletion. "" represents the press of the return button.

### 2. Initialising - set variables to their initial value

```
(setq ang 0  
n 0)
```

ang is the variable for the angle with the starting value 0. n is the increment.

### 3. REPEAT- the part in the repeat expression is executed 12 times:

- Define the corner points of the box.

```
(setq p1 (list 0 0 n)  
p2 (list 2 4 (+ 1 n)))
```

- Draw a box and name it thing.

```
(setq thing (solbox p1 p2 ""))
```

NOTICE: The solbox command is part of thesetq expression. thing is the entity name for the solbox. This way of assigning an entity name to an object is unique to AME objects.

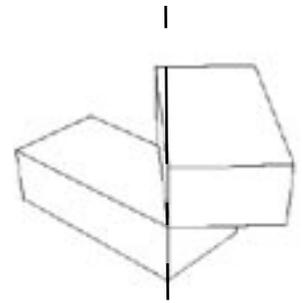
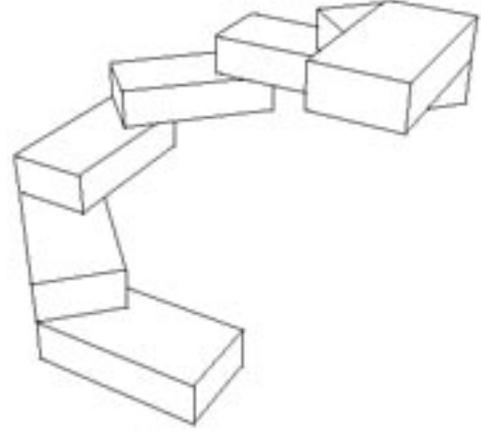
The use of rotate is an other example of an AutoCad command utilised in an AutoLisp script. The following line turns the object with the name thing around point p1 by the angle ang.

```
(command "rotate" thing "" p1 ang)
```

```
(defun C:ro ()  
  (command "erase" "all" "")  
  (setq ang 0  
        n 0)  
  (repeat 12  
    (setq p1 (list 0 0 n)  
          p2 (list 2 4 (+ 1 n))  
          thing (solbox p1 p2 "")  
          n (+ n 1))  
    (command "rotate" thing "" p1 ang)  
    (setq ang (+ ang 30))  
  )  
)
```

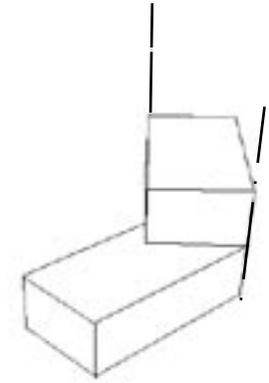
1.0	In order to make the routine more flexible it is necessary to prompt the user for the main parameters. Important parameters here are: The size of the blocks (defined by the corner points), the rotation angle, and the number of runs through the loop. Use the <code>getreal</code> and <code>getint</code> functions to make user input from the command line possible.								
Did you complete assignment 1.0? Did you need help?  What new topics did you learn?	<table border="0"><tr><td>Yes</td><td>No</td><td></td><td></td></tr><tr><td></td><td>Yes</td><td>No</td><td></td></tr></table>	Yes	No				Yes	No	
Yes	No								
	Yes	No							
	What old topics did you rehearse?								
Help provided	Tutors comments								

1.1	Shift the point of rotation from the inner left corner of the oblong to the outer right corner.						
Did you complete assignment 1.1? Did you need help?	<table border="0"> <tr> <td>Yes</td> <td>No</td> <td></td> </tr> <tr> <td></td> <td>Yes</td> <td>No</td> </tr> </table>	Yes	No			Yes	No
Yes	No						
	Yes	No					
What new topics did you learn?							
What old topics did you rehearse?							
Help provided	Tutors comments						



In the original spiral example the rotation point (p1) remains the same throughout the execution at the inner

In this exercise the point of rotation has to be moved dynamically to the outer right corner of the oblong. The rotation point p1 needs to be recalculated throughout the loop.



autolisp

## 2.0 The Chaos Game

- 26 The Chaos Game program draws a Sierpinski gasket point by point: First the attractors are defined, in the case of the Sierpinski gasket the three vertices of an isosceles triangle. The user then inserts the gamepoint, by clicking on the screen. The following procedure is repeated in a loop of predetermined length: random selection of one attractor, calculation of the new gamepoint halfway between selected attractor and recent gamepoint, and drawing of the new game point.

### The algorithm

#### 1. Housekeeping

2. Set attractors- In the setq expression the three attractor points are defined. With the layer AutoCad function you switch to another layer. NOTICE: The layer rot has to be created in the AutoCad drawing before running the program.

```
(command "layer" "set" "rot" "")
```

Before the three attractors are drawn the PDMODE is set to 96 that changes the way points are drawn on the screen.

```
(command "pdmode" "96" )  
(command "point" f1)  
(command "point" f2)  
(command "point" f3)
```

After that PDMODE and LAYER are set back to default.

```
(command "pdmode" "0" )  
(command "layer" "set" "0" "")
```

#### 3. Insert gamepoint

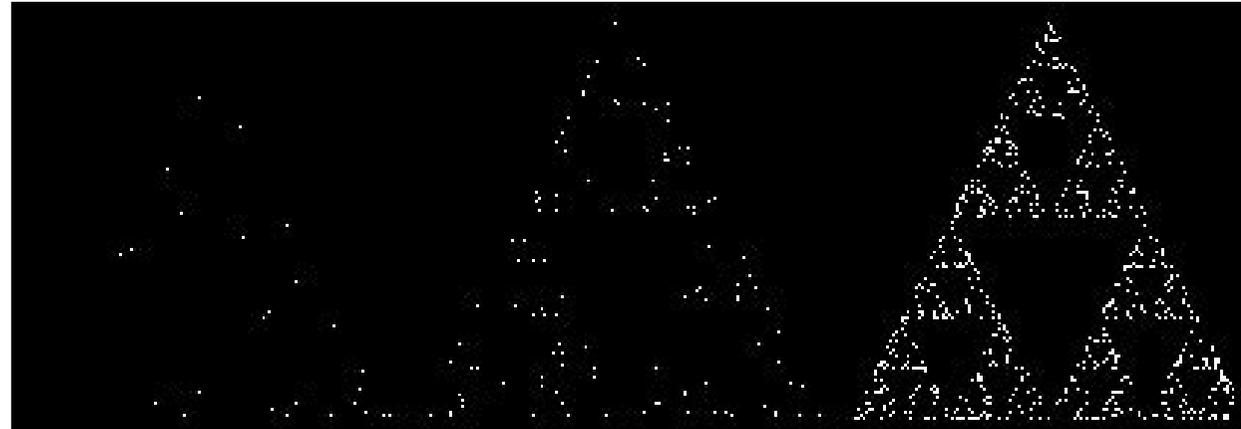
```
(setq gamepoint (getpoint "\nINSERT GAMEPOINT"))
```

getpoint allows you to insert a point by clicking on the drawing.

#### 4. REPEAT

- In the repeat loop three operations are executed over and over

Snapshots of the Chaos Game after 100 iterations, 1000 iterations and 10 000 iterations: The Sierpinski gasket emerges slowly.



autolisp

again. An attractor is chosen, the new gamepoint calculated and drawn. To choose an attractor the random number function `rand` is used:

```
(Defun rand (bot top / x z rn)
  .
  .
)
```

`rand` has two arguments `bot` and `top`. They define the bandwidth of the random number. With `(setq dice (rand 0 3))` the variable `dice` is set to a random number between 0 and 3. The conditional statement function `cond` determines according to `dice` which attractor will be chosen.

The calculation of the new game point is also done in the `cond` expression.

```
(setq newgamepoint (polar f3 (angle gamepoint f3)
  (- 0 (/ (distance f3 gamepoint) 2))))
```

`newgamepoint` is calculated as half the distance between current game point and the chosen attractor by using the distance and the

polar function.

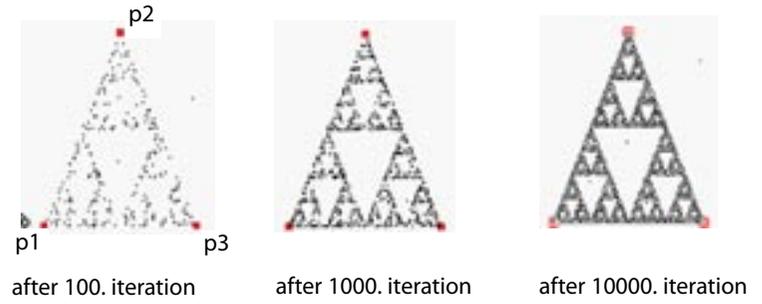
27

scripting

```
(Defun rand (bot top / x z rn)
  (if (NOT seed)(setq seed 758))
  (setq x (1+ (* seed 2197.0))
        z (fix (/ x 4096.0))
        seed (fix (- x (* z 4096.0)))
        r (* (/ seed 4096.0)(- top bot))
        n (+ bot r))
  )
```

scripting

```
defun c:CHAOS()
  (setvar "cmdecho" 0)
  (command "erase" "all" "")
  (setq f1 (list 0 0 0)
        f2 (list 2 5 0)
        f3 (list 4 0 0))
  (command "layer" "set" "rot" "")
  (command "pdmode" "96" )
  (command "point" f1)
  (command "point" f2)
  (command "point" f3)
  (command "pdmode" "0" )
  (command "layer" "set" "0" "")
  (setq seed (getint"\nRANDOM NUMBER GENERATOR")
        gamepoint (getpoint "\nINSERT GAMEPOINT")
        dice (rand 0 3))
  (repeat 10000
    (cond ((< dice 1) (setq newgamepoint (polar f1 (angle gamepoint f1)(- 0 (/ (distance f1 gamepoint) 2))))
          ((and (>= dice 1)(<= dice 2)) (setq newgamepoint (polar f2 (angle f2 gamepoint) (/ (distance gamepoint f2) 2))))
          ((> dice 2) (setq newgamepoint (polar f3 (angle gamepoint f3)(- 0 (/ (distance f3 gamepoint) 2))))))
    )
  (command "point" newgamepoint)
  (setq gamepoint newgamepoint
        dice (rand 0 3))
  )
)
```



28

2.0	Two ways of manipulation allow you to create an infinite number of interesting patterns. A) Alter the equation that calculates the new game point. B) Change the number of attractors.		
Did you complete assignment 2.0? Did you need help?		Yes	No
What new topics did you learn?		Yes	No
What old topics did you rehearse?			
Help provided			

e

s

i

c

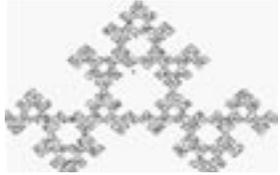
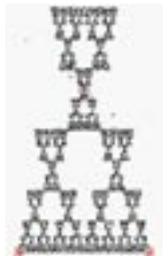
i

e

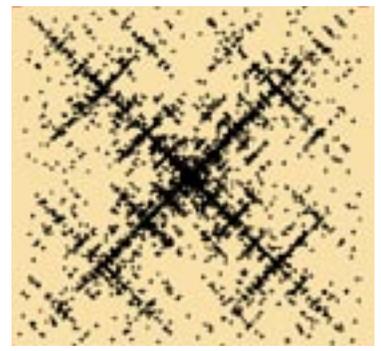
x

e

Tutors comments



A) You find the equation to calculate the new game point in the setq expression in the cond statement of the repeat loop. Altering here can produce surprising results.



B) The image above shows the Chaos game played with four instead of three attractors.

autolisp

29

exercise

scripting

## 2.1 Another Chaos Game: The Fern



30

The method above can create even more complex structures for example a fern leaf. The algorithm and parameters are borrowed from one of the most comprehensive books about fractals "Chaos and Fractals, New Frontiers of Science". Here is a brief description of how the program works, however for a thorough understanding of this method have a look at the book.

In the same way as example 2.0 we have a gamepoint -here the game point's coordinates,  $x$  and  $y$ , are defined in the beginning of the program and not inserted through mouse click by the user. A number of transformations describe different ways to calculate the new game point. This program has four such transformations: One for the stem, one for the right leaf, one for the left, and one for the top of the fern. First a random number  $r$  between 0 and 1 is computed with the `rand` -function. According to the the value of  $r$  transformation stem, right, left, or top is applied to the game point. The point is drawn and the next iteration calculated.

```
(defun rand) ..;random number function rand see previous example
```

```
(defun c:fern()
```

```
  (setvar "cmdecho" 0)
```

```
  (command "erase" "all" "")
```

```
  (setq left 30      w 300
```

```
        wl (+ w left) e1 (* w 0.5)
```

```
        e2 (* w 0.57)   e3 (* w 0.408)
```

```
        e4 (* w 0.1075) f1 (* w 0)
```

```
        f2 (* w -0.036) f3 (* w 0.0893)
```

```
        f4 (* w 0.27) x e1  y 0
```

```
        depth (getint "\nNUMBER OF ITERATION——"))
```

```
  (repeat depth
```

```
    (setq r (rand 0 1))
```

```
; transformation stem
```

```
  (if (< r 0.02)
```

```
    (progn
```

```
      (setq xn (+ e1 left)
```

```
            yn (- (+f1 (* y 0.27)) wl)
```

```
            pu (list xn yn 0))
```

```
            (command "point" pu)
```

```
    )
```

```
  )
```

```
; transformation right leaf
```

```
  (if (and (>= r 0.02) (< r 0.17))
```

```
    (progn
```

```
      (setq xn (+ (+ (* -0.139 x) (* 0.263 y) e2) left)
```

```
            yn (+ wl (+ (* x 0.246) (* 0.224 y) f2))
```

```
            pu (list xn yn 0))
```

```
            (command "point" pu)
```

```
    )
```

```
  )
```

scripting

```
;transformation left leaf
```

```
  (if (and (>= r 0.17) (< r 0.3))
```

```
    (progn
```

```
      (setq xn (- (+ (* 0.17 x) (* -0.215 y) e3) left)
```

```
            yn (- wl (+ (* x 0.222) (* 0.176 y) f3))
```

```
            pu (list xn yn 0))
```

```
      (command "point" pu)
```

```
    )
```

```
  )
```

```
;transformation top
```

```
  (if (>= r 0.3)
```

```
    (progn
```

```
      (setq xn (+ (- (* x 0.781) (* 0.034 y) e4) left)
```

```
            yn (+ wl (+ (* -0.032 x) (* y 0.739) f4))
```

```
            pu (list xn yn 0))
```

```
      (command "point" pu)
```

```
    )
```

```
  )
```

```
  (setq x xn
```

```
        y yn)
```

```
  ) ; end of repeat
```

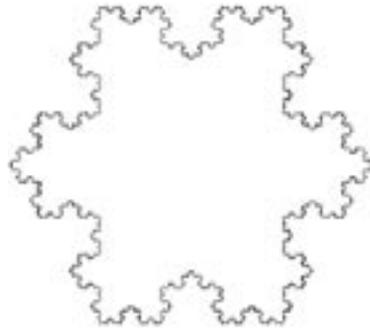
```
)
```

autolisp

31

scripting

### 3.0. Recursive Koch Curve



0. Iteration



1. Iteration



32

The recursive version of the Koch program is short and elegant, simply because it is able to rely on Autocad to keep track of the current position and calculate the endpoint of a polar coordinate line for us. The program has three main parts: KOCH II starts the program. All the housekeeping and the setting of the variables are done here. The initial motif is drawn by the function 'KOCH'.

```
(defun c:KochII()  
  (setvar "cmdecho" 0)  
  (command "erase" "all" "")  
  (command "regen")  
  (koch 0.0 5.0 5)  
)
```

The function 'KOCH' is called with the variables 0.0 = the starting angle (horizontal), 5.0 = the length of the base triangle, and 5 = the number of recursions.

KOCH then draws a line from 1 1 to 1 1 (it's of zero length, just to position the 'drawing point' so AutoCad has got somewhere to start drawing from), setq creates a variable called turn which is given the value 60 degrees, and then we call wiggle once each for the three base sides of the triangle.

```
(DEFUN koch(angl lngth depth)  
  (setq origin (list 1 1)  
        turn (/ pi 3))  
  (command "LINE" origin origin "")  
  (Wiggle angl lngth depth)  
  (Wiggle (+ angl (* 2 turn)) lngth depth)  
  (Wiggle (+ angl (* 4 turn)) lngth depth)  
)
```

Drawing is done with the RELATIVE POLAR LINE command, which has the syntax (line @ length < angle). The functions strcat and rtos are unfortunately necessary to convert the internal representation of our length variable into a line of text that the line command can understand.

To make the nested brackets a bit more explicit boxes are drawn around the various components in the diagram on the next page. 1 is the outer level which encloses the whole function, 2 is the conditional expression, and 3 is the dependent part which is executed if the cond expression isn't true - the main recursive calls.



2. Iteration



3. Iteration

```
(DEFUN Wiggle (ang lng dep)
  1
  (cond
    ((= dep 0) (command "LINE" ""
      2
      (strcat@(rtos lng)"<" (rtos ang)) ""))
    (T
      (wiggle ang (/ lng 3) (- dep 1))
      (setq ang (- ang turn))
      (wiggle ang (/ lng 3) (- dep 1))
      3
      (setq ang (+ ang (* 2 turn)))
      (wiggle ang (/ lng 3) (- dep 1))
      (setq ang (- ang turn))
      (Wiggle ang (/ lng 3) (- dep 1))
    )
  )
)
```

scripting

```
(DEFUN koch(ang lng depth)
  (setq origin (list 1 1))
  turn 60)
(command "LINE" origin origin "")
(Wiggle ang lng depth)
(Wiggle (+ ang (* 2 turn)) lng depth)
(Wiggle (+ ang (* 4 turn)) lng depth)
)

(DEFUN Wiggle (ang lng dep)
  (cond ((= dep 0) (command "LINE" " (strcat "@" (rtos lng)
    "<" (rtos ang)) ""))
    (T
      (wiggle ang (/ lng 3) (- dep 1))
      (setq ang (- ang turn))
      (wiggle ang (/ lng 3) (- dep 1))
      (setq ang (+ ang (* 2 turn)))
      (wiggle ang (/ lng 3) (- dep 1))
      (setq ang (- ang turn))
      (Wiggle ang (/ lng 3) (- dep 1))
    )
  )
)

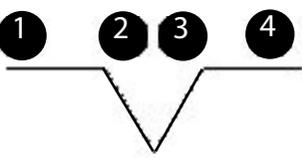
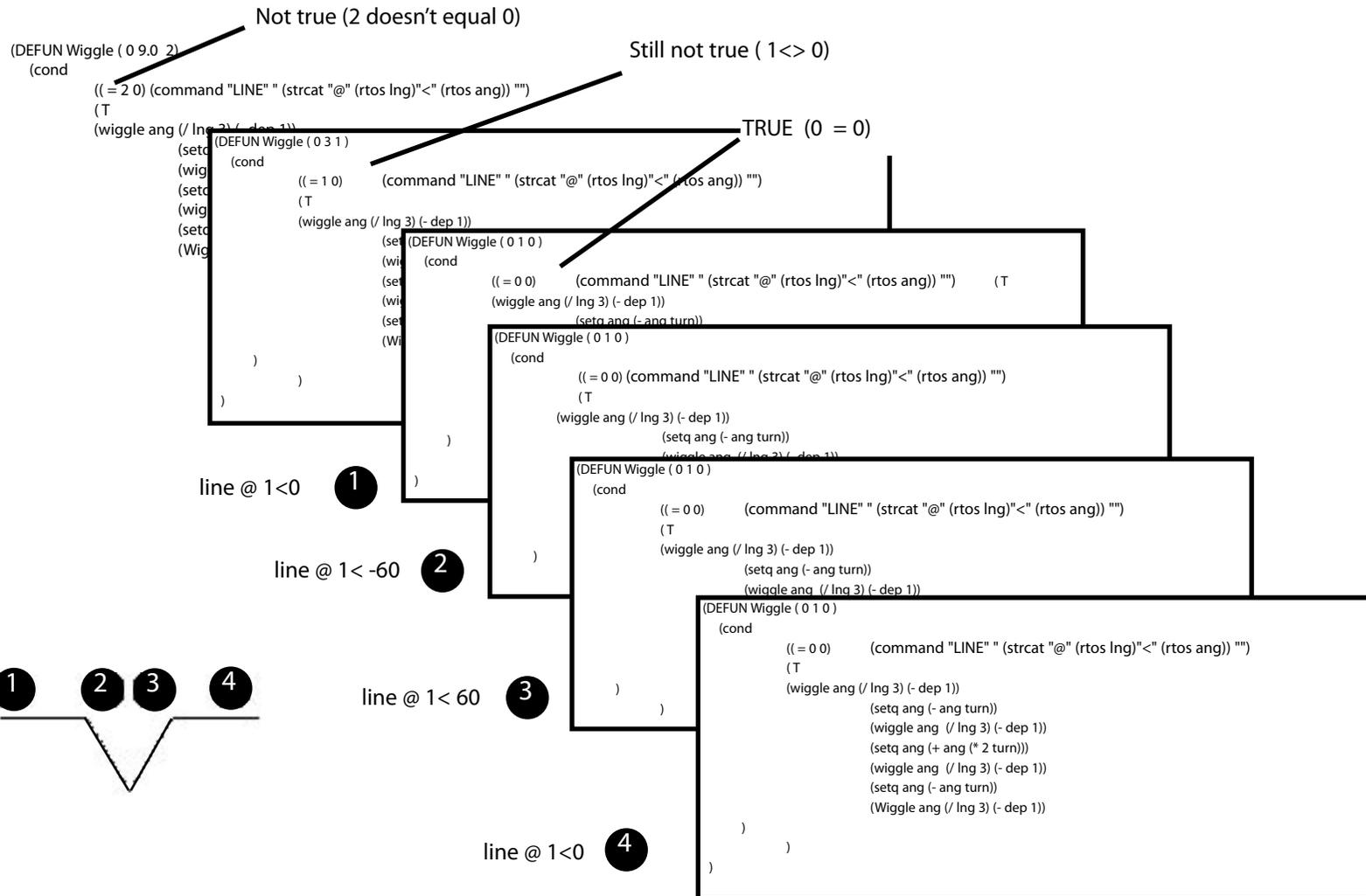
(DEFUN c:KochII()
  (setvar "cmdecho" 0)
  (command "erase" "all" "")
  (command "regen")
  (koch 0.0 5.0 5)
)
```

autolisp

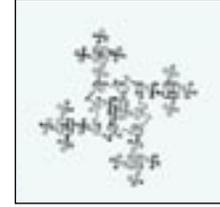
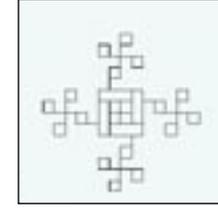
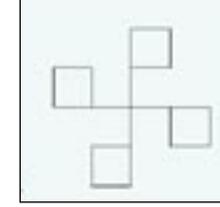
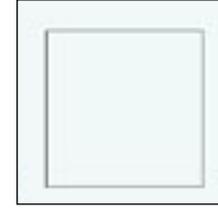
scripting

Assume that the depth = 2 again, and the angle = 0, length = 9.0.  
 Than the flow through the program would look something like this.

34



3.0	Experiment with different starting figure, a square instead of a triangle, for example. That involves the change of the number of calls to wiggle in the koch function as well as the alteration of their passed parameters. In order to change the motif the recursive calls in wiggle need to be changed.						
Did you complete assignment 3.0? Did you need help?	<table border="0"> <tr> <td>Yes</td> <td>No</td> <td></td> </tr> <tr> <td></td> <td>Yes</td> <td>No</td> </tr> </table>	Yes	No			Yes	No
Yes	No						
	Yes	No					
What new topics did you learn?							
What old topics did you rehearse?							
Help provided	Tutors comments						



Different initiator and motif give rise to other self similar pat-



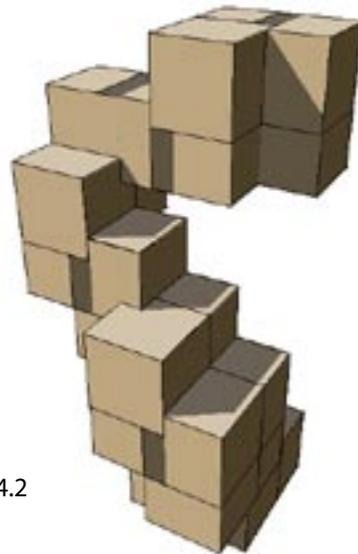


figure 4.2



figure 4.3

```
to 'biglist'
  (while (and (< c end)(not (equal (nth c biglist) current)))
    (setq c(1+ c))
  ); end while
  (if (>= c (length biglist))(insertbox))
```

It's worth looking at this loop in a bit more detail:  
 First, the list BIGLIST is constructed as a list of points, which are themselves lists. If you run the WANDER program for 15 iterations, and then use the 'pling' (!) operator to print out the contents of the three lists Biglist, current and last on the AutoCad commandline you will see something like this:

```
Command: !biglist
<<0 0 0> <0 0 1> <0 1 1> <0 1 2> <0 0 2> <-1 0 2> <-2 0 2>
<-2 1 3> <-3 1 3> <-3 0 3> <-3 0 4> <-3 -1 4> <-3 -1 5>

command: !current
(-3 -1 4)

command: !last
(-3 -1 5)
```

The biglist is seen to be made up of triples, (0 0 0) (0 0 1) etc. which are the individual insertionpoints of each successful addition of a

cube and are themselves lists, making biglist a list of lists. Each one is one unit away from the last one in either x y or z. Starting at (0 0 0) the first addition was in the Z axis, giving (0 0 1) and then in the Y axis from there giving (0 1 1). Notice that current is the same list as the penultimate one, and last is the same as the last one.

To check to see whether we can place a block at the new location the while loop runs through the list comparing each element in biglist with current. To do this, we first put zero into a counter 'c' the number of elements in biglist into the variable 'end'.

```
(setq c 0 end (length biglist))
```

The loop then uses

```
(while (and (< c end)(not (equal (nth c biglist) current)))
  (setq c(1+ c))
)
```

which means "while the value of C is less than end, and the C'th element of biglist is not equal to the current position, go on adding 1 to C". Remember that each element of biglist is itself a list, so taking the list shown above,

```
(Nth 1 Biglist)
```

```
(defun rand ; Random number function as in previous example
```

```
(Defun insertbox()
```

```
  (setq biglist(append biglist(list current))
```

```
    last current)
```

```
  (command "insert" "lump" current "" "" "0")
```

```
)
```

```
(defun C:worm ()
```

```
  (setvar "cmdecho" 0)
```

```
  (setq rubbish(ssget "x")
```

```
    seed (getint "seed for random series :")
```

```
    n (getint "How many replays ")
```

```
    last '(0 0 0)
```

```
    biglist (list last)
```

```
    choice 0)
```

```
  (IF rubbish (command "erase" rubbish ""))
```

```
    (command "insert" "lump" last "" "" "0")
```

```
    (repeat n (setq choice (+ (fix (* 6.0 (rand))) 1))
```

```
      (cond ((= choice 1)(setq current (list (car last)(cadr last)(+ (caddr
```

```
last) 1))))
```

```
        ((= choice 2)(setq current (list (car last)(cadr last)(- (caddr
```

```
last) 1))))
```

```
        ((= choice 3)(setq current (list (car last)(+ (cadr last)
```

```
1)(caddr last))))
```

```
        ((= choice 4)(setq current (list (car last)(- (cadr last) 1)(caddr
```

```
last))))
```

```
        ((= choice 5)(setq current (list (+ (car last) 1)(cadr last)(caddr last))))
```

```
        ((= choice 6)(setq current (list (- (car last) 1)(cadr last) (caddr last))))
```

```
      )
```

```
      (setq c 0 end (length biglist))
```

```
      (while (and (< c end)(not (equal (nth c biglist) current)))
```

```
        (setq c(1+ c))
```

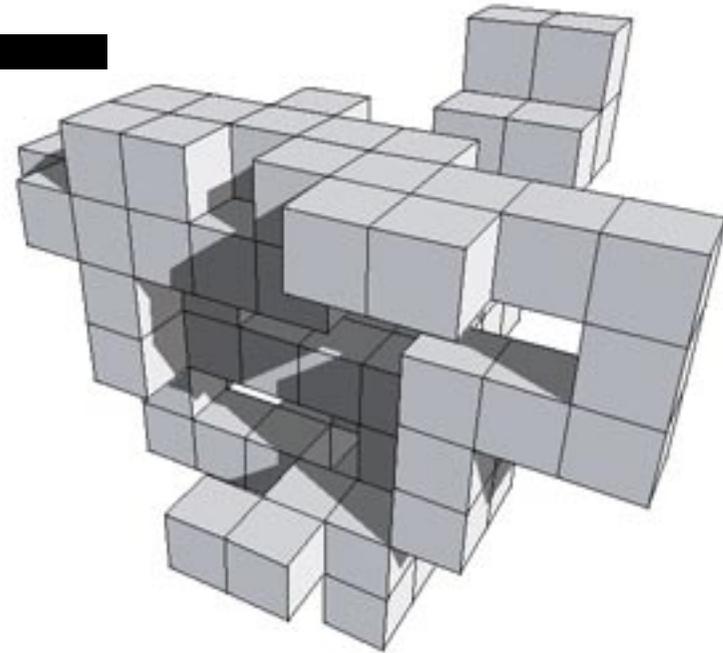
```
      ) ; end while
```

```
      (if (>= c (length biglist))(insertbox))
```

```
    ) ;end repeat
```

```
)
```

scripting



38

4.0	Change the program so that adding on one direction is preferred to the other five.						
Did you complete assignment 4.0? Did you need help?	<table border="0"> <tr> <td>Yes</td> <td>No</td> <td></td> </tr> <tr> <td></td> <td>Yes</td> <td>No</td> </tr> </table>	Yes	No			Yes	No
Yes	No						
	Yes	No					
What new topics did you learn?							
What old topics did you rehearse?							
Help provided	<p>Tutors comments</p>						

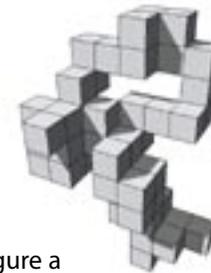


figure a



figure b

Image a shows wander assembling 70 cubes with equal probability for all six directions. In image b the probability of an addition in horizontal forward direction was five times higher

## 5.0 Scatter

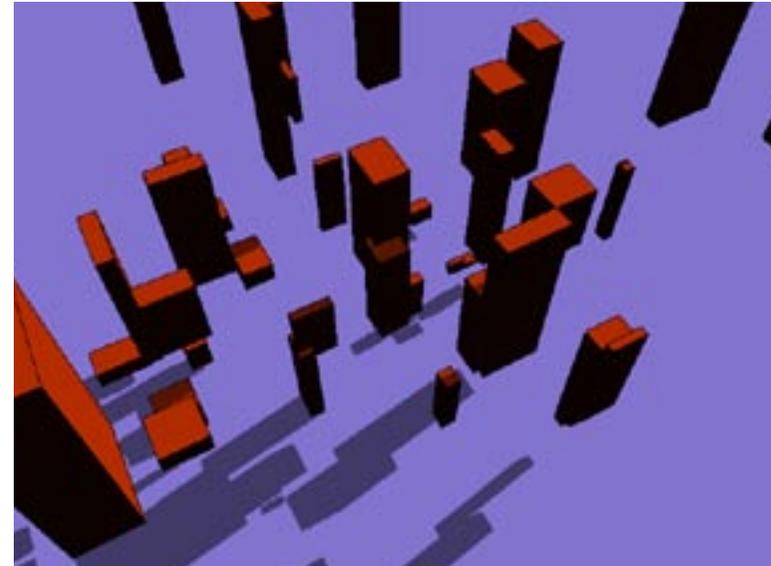
- 40 Scatter is a simple random insertion loop. With the help of the rand function it scatters down blocks with randomly varying dimensions. scatter is made up of three parts the rand function which we encountered in earlier examples, the function scatter and c:go. IMPORTANT: To run the program you need a block called "box" which is a cube of side 1, whose origin is at 0 0 0.

### The algorithm

1. Housekeeping- All the tidying up is done in c:go. This is the command function, you can type its name in the command line and it will start to execute. Here the user is prompted for an integer ('seed') to start the random function rand, and the number of blocks, called 'counter'.

```
(setq seed (getint"\nPlease enter a random number generator:")
          counter (getint"\nEnter the number of boxes to be inserted:")
)
```

At the end scatter is called and passed the variable counter.



```
(scatter counter)
```

### 2. Get scatterpoint

The scatter function uses a while loop, which is controlled by the counter (counter) input by the user, passed as a parameter and here called 'number'. Local variables are used to store the new random position and scalefactors.

```
( while (> number 0)
  (setq x (rand 0 10)
        z (rand 0 10)
        randompoint (list x y z)
        xscale (rand 1 5)
        yscale (rand 1 5)
        zscale (rand 1 3)
  )
)
```

The box is then inserted, and the number decremented

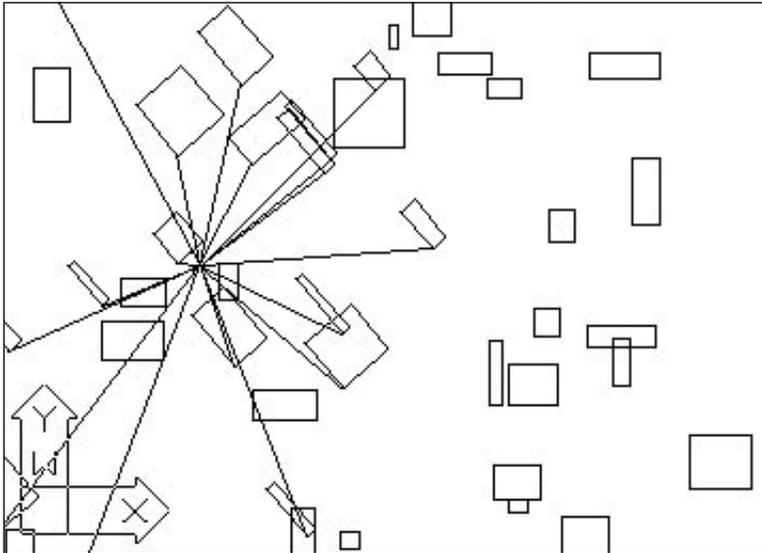
```
( command "insert" "box" randompoint "xyz" xscale yscale
zscale "0"
  (setq number(1- number))
)
```

```
(defun rand ) ;random number function as in previous examples

(defun scatter ( number / x y z xscale yscale zscale)
  ( while (> number 0)
    (setq x (rand 0 10) ;xcoordinate randomly between 1 and 10
          y (rand 0 10) ;ycoordinate randomly between 1 and 10
          z (rand 0 10) ;zcoordinate randomly between 1 and 10
          randompoint (list x y z) ;put in list called 'randompoint'
          xscale (rand 1 5) ;stretch x between 1 & 3 times
          yscale (rand 1 5) ;stretch y between 1 & 3 times
          zscale (rand 1 3) ;stretch z between 1 & 5 times
          )
    (command "insert""box" randompoint "xyz" xscale yscale zscale "0")
    (setq number(1- number))
  )
)

(defun c:go()
  (setvar"cmdecho"0)
  (setq rubbish(ssget"x"))
  (if rubbish(command"erase"rubbish""))
  (setq rubbish nil
        seed (getint"\nPlease enter a random number generator:")
        counter (getint"\nEnter number of boxes to be inserted:")
        )
  (scatter counter)
)
```

## 5.1 Near



42

The routine 'near' is an addition to the previous example. It is a development of the "scatter" program that involves scanning the database after the random cuboids have been inserted, so as to demonstrate the use of the data manipulation functions. To prove that they work, the program asks the user to click the mouse somewhere in the boxes (HOTSPOT), and then the 'nearest' boxes to the mouseclick are rotated a fixed amount. Also a line is drawn from the insertion point of each of the cuboids to the hotspot to show where they are.

### The algorithm

The addition consists of three parts eg, scan a nd rotateobject, with eg being the main part. It . . The selectionset everything is built containing all cuboids and the function scan is called.

1. Housekeeping- the eg replaces the function c:go and contains all the housekeeping.

2. Create cuboids- From eg the previously defined function scatter is called with counter as the argument.

```
(scatter counter)
```

3. Define HOTSPOT- Then the user is asked to define HOTSPOT by mouse-click. The variable HOTSPOT then is the list containing the point that you click on.

```
(setq hotspot (getpoint "click the mouse"))
```

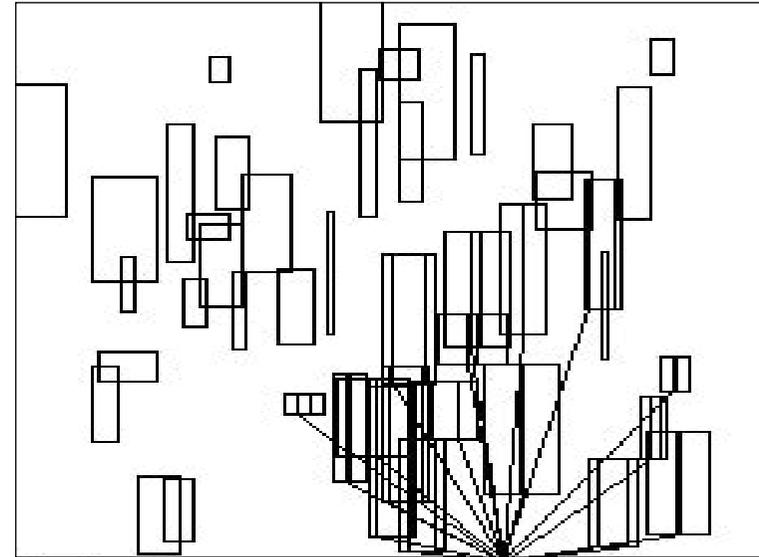
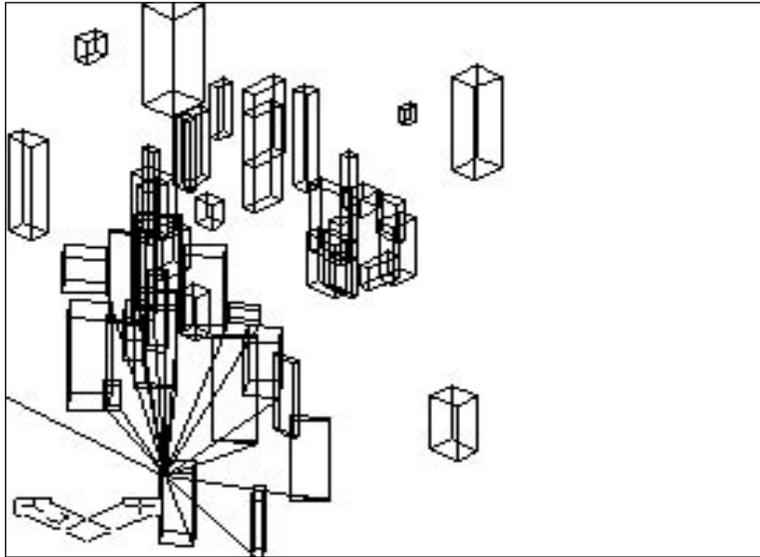
4. Build selectionset- The selectionset containing all the objects in the drawing is built and put in the variable everything.

```
(setq everything (ssget "x"))
```

5. Scan selectionset- The user defined function scan is called with two required arguments: The selectionset everything and HOTSPOT. The function begins with an IF statement which means that if the selectionset is empty the whole function will be skipped.

```
(if selectionset
  (progn .....))
```

progn is a 'glue' function which joins the statements inside it together so that they are all done when the IF is true.



The repeat function loops through the selectionset. In the loop each individual object is examined, using counter to choose successive objects. It finds the insertionpoint of the cuboids and the distance from the point the user clicked on. This information is stored in the variables insertionpoint and faraway.

```
(setq objecthandle (ssname selectionset counter)
      objectdata (entget objecthandle)
      insertionpoint (cdr (assoc 10 objectdata))
      faraway(distance hotspot insertionpoint)
  )
```

If the distance is less than 60 units then draw a line from HOTSPOT to insertionpoint of this object, and rotate it.

```
(if (< faraway 60.0 )
    (progn
      (command "line" insertionpoint hotspot "")
      (rotateobject objectdata 0.7)
    )
  )
```

RotateObject is a little function that alters the rotation angle in the block -passed as a parameter called this- to angle.

```
(Defun rotateobject ( this angle )
  (setq this (subst ( cons 50 angle )(assoc 50 this)
                    this))
  (entmod this)
  )
```

Manipulating the database is essential for generative modelling. Here is a summary of the new functions the script makes use of:

To alter an object in the database

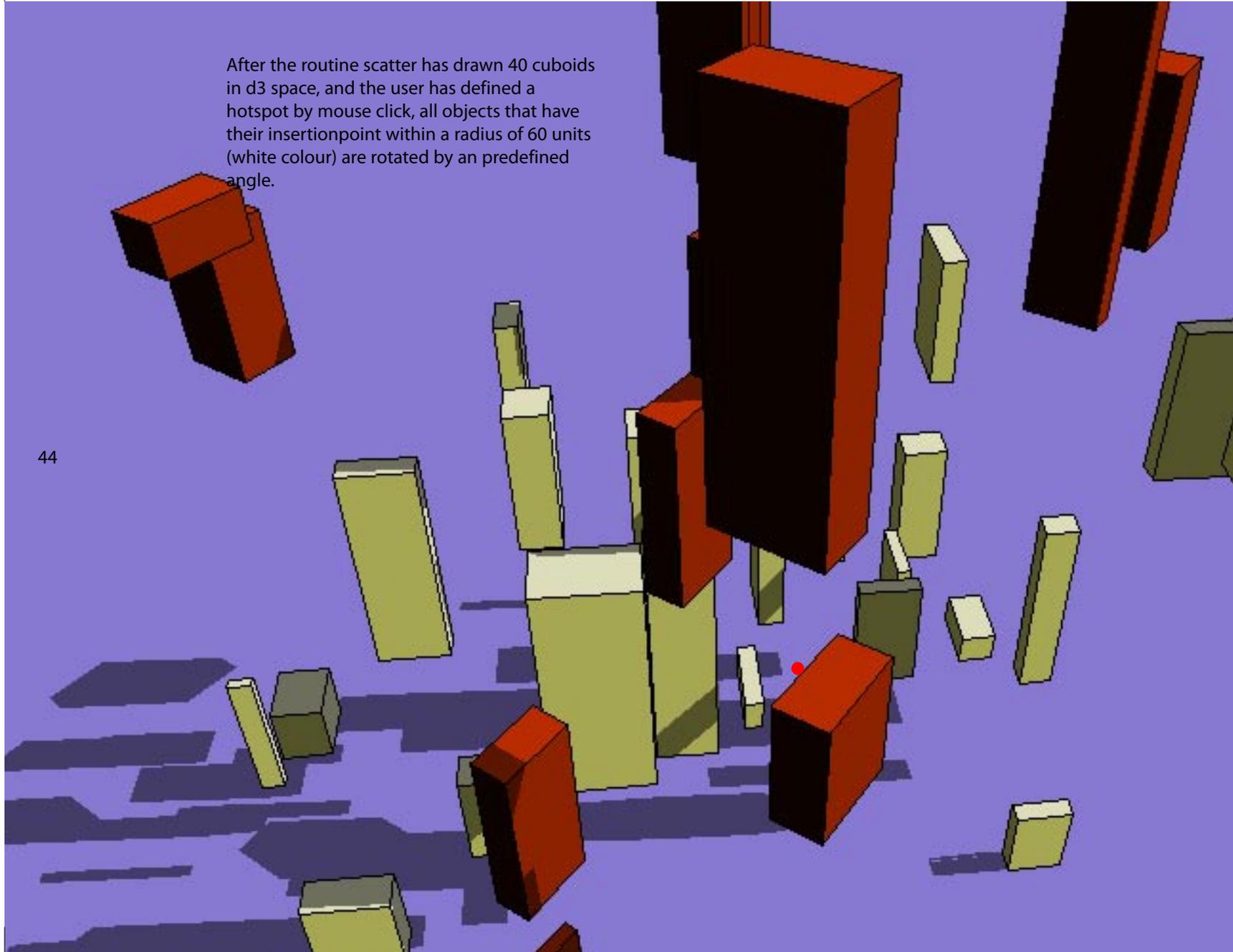
- subst      substitute one element for another
- cons       construct a dotted pair
- entmod     modify entity

To extract data from the drawing

- ssname     gets a handle to an object in the selection
- entget     gets all the data about the object
- assoc       extracts a particular bit of data

After the routine scatter has drawn 40 cuboids in d3 space, and the user has defined a hotspot by mouse click, all objects that have their insertionpoint within a radius of 60 units (white colour) are rotated by a predefined angle.

44





5.0	In near point hotspot is a point in x-y plane. Ask user for the height to be appended to the hotspot.
Did you complete assignment 5.0? Did you need help?  What new topics did you learn?	Yes    No Yes    No
	What old topics did you rehearse?
Help provided	
	Tutors comments

5.1	In the function rotateobject the angle of the object is altered. Choose an other attribute to modify.( see table)						
Did you complete assignment 5.1? Did you need help?  What new topics did you learn?	<table border="0"> <tr> <td>Yes</td> <td>No</td> <td></td> </tr> <tr> <td></td> <td>Yes</td> <td>No</td> </tr> </table>	Yes	No			Yes	No
Yes	No						
	Yes	No					
	What old topics did you rehearse?						
Help provided	Tutors comments						

Some adresses and their associated values:

- 8 layer
- 10 insertionpoint
- 41 X scale factor
- 42 Y scale factor
- 43 Z scale factor
- 50 rotation angle
- 62 colour

## 6.0 Growth Generator

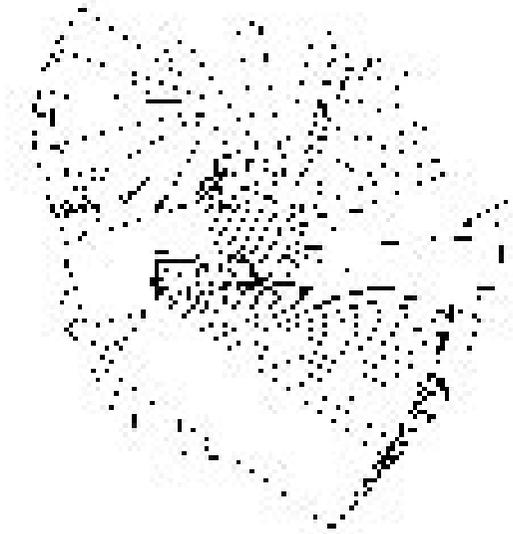
48

William Latham is an artist who has been working in close association since 1987 with Stephen Todd, a programmer at the IBM UK Scientific Centre in Winchester, together they are producing "Evolutionary Art" using the computer. Latham was inspired by natural systems, and how they often relied on very simple steps, such as crystal growth, or the creation of stalagmites by water dripping in underground caverns. Even biological processes are related to simple geometry's as shown by D'Arcy Thompson (1961), and the repeated small changes in mutation and natural selection give rise to a huge variation of biological forms. Latham was aware that these natural systems have a huge potential for creating artistic forms, he wanted to exploit these by the creation of drawings based on random dice throws for synthetic organic form.

This program follows a similar approach and produces similar three dimensional objects. They are generated using a simple set of rules, these rules can be applied by choice or randomly.

### The algorithm

1. Housekeeping- The function `c:generator` erases all entities and inserts the first primitive. Variables are set and a call to `growth` is made.

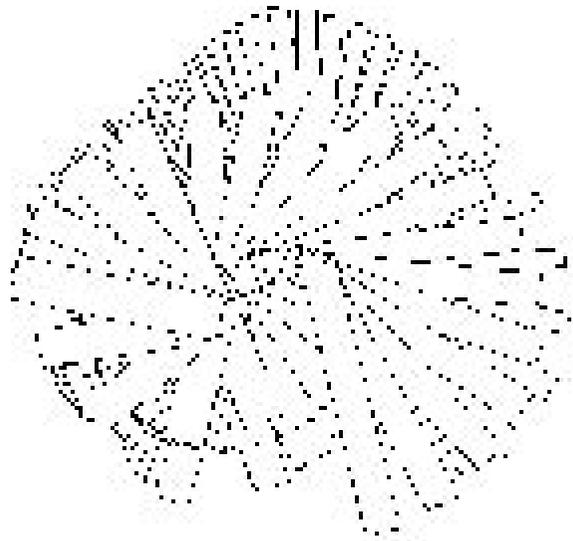


2. Repeat- Before the repeat-loop, variables for the rules are randomly decided, using the random number function `rand`.

```
(defun growth()
  (setq count(+ 1 count)
        num (fix(rand 4 15))
        rotate_factor (rand -270 270)
        .....
        growz_factor (rand -1 0.8)
  )
```

Calls to the transformation-rule functions `CLIMB`, `ROTATE`, `STACK`, `MOVE` and `GROW` are made.

```
(repeat num
  (setq last (cdr (entget (entlast))))
  (stack)
  .....
  (grow)
  (entmake last)
)
```



At the end a recursive call to growth is made in an IF statement in order to repeat the routine 10 times.

49

```
(if (< count 10) (growth))
```

3. The rules- The five rule functions work all in a similar way. They first establish data for the last entity added, adjusts the x, y, z or angle data with variance and then makes a new entity with the new values using the function entmod.

```
(defun grow()  
  (setq old_xscale (cdr(assoc 41 last))  
        old_yscale (cdr(assoc 42 last))  
        old_zscale (cdr(assoc 43 last))  
        new_xscale (+ old_xscale growx_factor)  
        new_yscale (+ old_yscale growy_factor)  
        new_zscale (+ old_zscale growz_factor)  
        last (subst (cons 41 new_xscale) (assoc 41  
last)|last)  
        last (subst (cons 42 new_yscale) (assoc 42  
last)|last)  
        last (subst (cons 43 new_zscale) (assoc 43
```

```
(defun c:generator()
  (setvar "cmdecho" 0)
  (command "erase" (ssget "x") "")
  (command "insert" "box5" "0,0" "xyz" "1" "1" "1" "0")
  (setq count 0)
  (growth)
)
```

```
(defun growth()
  (setq count(+ 1 count))
  (setq num (getint "\nHow many blocks: "))

  (setq num (fix (rand 4 15)))
  rotate_factor (rand -270 270)
  stack_factor (rand 0 5)
  move_factor (rand -3 3)
  climb_factor (rand -5 1)
  growx_factor (rand -1 1.2)
  growy_factor (rand -1 1.2)
  growz_factor (rand -1 0.8)

  (repeat num
    (setq last (cdr (entget (entlast))))
    (stack)
    (move)
    (rotate)
    (climb)
    (grow)
    (entmake last)
  )
  (command "zoom" "e")
  (if (< count 10) (growth))
)
```

```
(defun grow() ;rule function
```

```

      (setq old_xscale(cdr(assoc 41 last))
            old_yscale(cdr(assoc 42 last))
            old_zscale(cdr(assoc 43 last))
            new_xscale(+ old_xscale growx_factor)
            new_yscale(+ old_yscale growy_factor)
            new_zscale(+ old_zscale growz_factor)
            last (subst (cons 41 new_xscale) (assoc 41
last)last)
            last (subst (cons 42 new_yscale) (assoc 42 last)last)
            last (subst (cons 43 new_zscale) (assoc 43 last)last))
      (entmod last)
    )
  (defun climb(/ z) ;rule function
    (setq x (car (cdr (assoc 10 last)))
          y (cadr (cdr (assoc 10 last)))
          z (caddr (cdr (assoc 10 last)))
          new_position(list x y (+ z climb_factor))
          last (subst (cons 10 new_position)(assoc 10 last)last))
    (entmod last)
  )
  (defun rotate() ;rule function
    (setq old_angle(cdr(assoc 50 last))
          new_angle(- old_angle (dtr (/ rotate_factor num)))
          last (subst (cons 50 new_angle)(assoc 50 last)last))
    (entmod last)
  )
  (defun stack(/ x y z) ;rule function

```

scripting

```

    (setq x (car (cdr (assoc 10 last)))
          y (cadr (cdr (assoc 10 last)))
          z (caddr (cdr (assoc 10 last))))
    (setq new_position(list x (+ y stack_factor) z))
    (setq last (subst (cons 10 new_position) (assoc 10 last)last))
    (entmod last)
  )
  (defun move(/ x y z)
    (setq x(car(cdr(assoc 10 last))))
    (setq y(cadr(cdr(assoc 10 last))))
    (setq z(caddr(cdr(assoc 10 last))))
    (setq new_position(list (+ x move_factor) y z))
    (setq last (subst (cons 10 new_position)(assoc 10 last)last))
    (entmod last)
  )
  (Defun DTR(a)
    (* pi(? a 180))
  )

```

autolisp

51

scripting

6.0	In growth the various factors are obtained randomly from a set of constants (0 5) -3 +3 etc. Define setq variables and get user input for the 7 factors and substitute the variables for the constants.
Did you complete assignment 6.0? Did you need help?  What new topics did you learn?	Yes    No Yes    No
	What old topics did you rehearse?
Help provided	Tutors comments

6.1	Add one other factor to change colour or layer.						
Did you complete assignment 6.1? Did you need help?	<table border="0"> <tr> <td>Yes</td> <td>No</td> <td></td> </tr> <tr> <td>Yes</td> <td>No</td> <td>No</td> </tr> </table>	Yes	No		Yes	No	No
Yes	No						
Yes	No	No					
What new topics did you learn?							
What old topics did you rehearse?							
Help provided	<p>Tutors comments</p>						

## 7.0 Shape Grammar

54

The aim here is to create a program that generates three dimensional form using a simplified use of shape grammar rules (see G.Stiney and see G.D.L. for complete analysis). The main vocabulary elements of the grammar are to be three dimensional rectangular blocks. The rules describe transformations of these blocks in space by taking their symmetrical properties into account.

The method used here for creating shape grammar incorporates the movement and rotation of the User Co-ordinate System (UCS) in Autocad as shown in figure 7.1. Instead of actually moving and rotating the blocks, the UCS is translated into the new position and the new block inserted with (0 0 0) as the insertion point. The program consist of four functions: The three functions rule3, rule4, and rule5 contain information how to move the UCS and then inserts two different sized blocks. The main function c:345 combines those three rules randomly, using the random number function rand. But additionally the routine allows the user to run the rules separately by calling the rule functions directly in the form of (rule3). When this program has been loaded the following messages are displayed.

```
(prompt "\n.....box loaded")
(prompt "\nType 345 to run all 3 rules")
(prompt "\nor type RULE3 or RULE4 or RULE5 in brackets")
```

Example of the random application of rule 3, 4, and 5 using the main function c:345

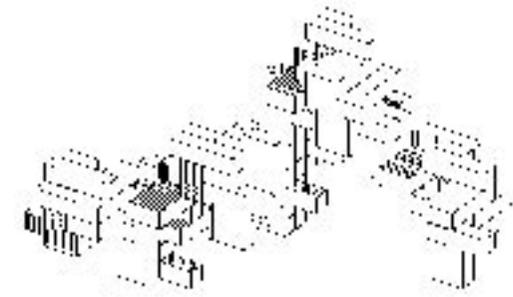


figure 7.1

```
(prompt "\nto run one rule at a time.")
```

The algorithm

The main function c:345 - This function has calls to RULE3, 4 and 5 each being chosen at random all have a 33% chance of occurrence. At the end it displays a hidden 3D view of the generated form.

1. Housekeeping- The view is set to 3d view, everything on layer "0" is erased. The global variable all is set to 1. all functions as an indication for the rule functions that they are used as part of the main function and not as separate functions.

```
(command "vpoint" "2,-3,2")
(command "erase" (ssget "x" ((8 . "0"))) "")
(setq all 1)
```

Resetting the the UCS to world and asking the user for the number of replays is also part of the housekeeping.

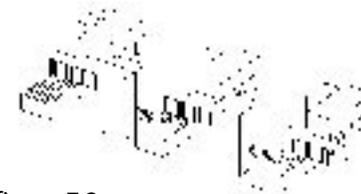
```
(command "ucs" "w")
(setq allnum( getint "How many replays: "))
```

2. Initialisation- The first block is inserted.

figure 7.2



figure 7.3



Rule 3 - First three steps and repeated 12 times (above)



Rule 4 - First three steps and repeated 12 times (above)

(command "insert" "wide" "0,0,0" "" "" "0")  
3. Repeat- At the beginning of this loop a random number is set between 0 and 1..

```
(setq randnum(rand 0 1))
```

With a 33 % chance of occurrence calls are made to the rule functions embedded in an if statement.

```
(if (< randnum 0.33)(rule3))
(if (> randnum 0.66)(rule4))
(if (and(> randnum 0.33)(< randnum 0.66))(rule5))
```

4. Clean up- At the end the zoom "e" command makes sure that the whole drawing is displayed before a hidden line rendering is initiated.

```
(command "zoom" "e")
(command "hide")
```

The rule functions are all based on the same structure- The functions are for inserting blocks in accordance with different translation rules (see figure 8.1, 8.2, and 8.3), they are called from the main function c:345. Rules 3,4 + 5 can also be run separately by typing the rule name at the command prompt enclosed with brackets ie. (rule5).

1. Housekeeping- Embedded in an if-statement we find the same housekeeping commands as in the main function. If a rule function is called from c:345 then all has the value 1 and the housekeeping is ignored.

```
(if (/= all 1)
  (progn
    (command "ucs" "w")
    .....
    .....
  )
)
```

2. Repeat- In the loop the two blocks "wide" and "narrow" are inserted after the UCS is moved and rotated using the ucs Autocad com-



figure 7.4



Rule 4 - First three steps and repeated 12 times (above)

56 mand.

)

```
(command "ucs" "o" "4,-1,1" "ucs" "x" "90")
(command "insert" "narrow" "0,0" "" "" "270")
```

meaning that the UCS's origin is shifted into point (4 -1 1) and rotated around the x-axis by 90. Then block narrow is inserted. Block wide is treated in the same fashion.

```
(command "ucs" "o" "-2,3,1")
(command "insert" "wide" "0,0,0" "" "" "0")
```

Note, that repeat is executed only once if entered from the main function. (In c:345 variable num is set to 1)

3. Clean up- Same as in the main function. Again optional depending on how the rule function is entered.

```
(if (/= all 1)
  (progn
    (command "zoom" "e")
    (command "hide")
  )
)
```

```
(defun rand ; using the random number function see example 2.0
```

```
(defun RULE3()
  (if (/= all 1)
    (progn
      (command "ucs" "w")
      (command "erase" (ssget "x" '((8 . "0"))) "")
      (setq num(getint "How many replays:"))
      (command "insert" "wide" "0,0,0" "" "" "0")
    )
  )
  (repeat num
    (command "ucs" "o" "4,-1,1" "ucs" "x" "90")
    (command "insert" "narrow" "0,0" "" "" "270")
    (command "ucs" "o" "-2,3,1")
    (command "insert" "wide" "0,0,0" "" "" "0")
  )
  (if (/= all 1)
    (progn
      (command "zoom" "e")
      (command "hide")
    )
  )
)

(defun RULE4()
  (if (/= all 1)
    (progn
      (command "ucs" "w")
      (command "erase" (ssget "x" '((8 . "0"))) "")
      (setq num(getint "How many replays:"))
      (command "insert" "wide" "0,0,0" "" "" "0")
    )
  )
  (repeat num
    (command "ucs" "o" "4,-1,1" "ucs" "x" "90")
    (command "insert" "narrow" "0,0" "" "" "270")
    (command "ucs" "o" "-2,3,-1")
    (command "insert" "wide" "0,0,0" "" "" "0")
  )
  (if (/= all 1)
    (progn
      (command "zoom" "e")
      (command "hide")
    )
  )
)
)
```

scripting

```
(defun RULE5()
  (if (/= all 1)
    (progn
      (command "ucs" "w")
      (command "erase" (ssget "x" '((8 . "0"))) "")
      (setq num(getint "How many replays:"))
      (command "insert" "wide" "0,0,0" "" "" "0")
    )
  )
  (repeat num
    (command "ucs" "o" "4,-1,1" "ucs" "x" "90")
    (command "insert" "narrow" "0,0" "" "" "270")
    (command "ucs" "o" "-2,0,1")
    (command "insert" "wide" "0,0,0" "" "" "0")
  )
  (if (/= all 1)
    (progn
      (command "zoom" "e")
      (command "hide")
    )
  )
)

(defun c:345( / all)
  (command "vpoint" "2,-3,2")
  (command "erase" (ssget "x" '((8 . "0"))) "")
  (setq all 1)
  (command "ucs" "w")
  (setq allnum( getint "How many replays: "))
  (setq num 1)
  (command "insert" "wide" "0,0,0" "" "" "0")
  (repeat allnum
    (setq randnum(rand 0 1))
    (if (< randnum 0.33)(rule3))
    (if (> randnum 0.66)(rule4))
    (if (and(> randnum 0.33)< randnum 0.66))(rule5))
  )
  (command "zoom" "e")
  (command "hide")
)

(prompt "\n.....box loaded")
(prompt "\nType 345 to run all 3 rules")
(prompt "\nror type RULE3 or RULE4 or RULE5 in brackets")
(prompt "\nto run one rule at a time.")
```

autolisp

57

scripting

7.0	The oblongs narrow and wide are AutoCad blocks. Change their dimension so you end up with a very long narrow and a square wide. To make it work you have to alter the ucs-command in the rule function.
Did you complete assignment 7.0? Did you need help?  What new topics did you learn?	Yes    No Yes    No
What old topics did you rehearse?	
Help provided	Tutors comments

7.1	e	Create your own shape grammars or adjust these ones to generate your own form.						
Did you complete assignment 7.1? Did you need help?	s	<table border="0"> <tr> <td>Yes</td> <td>No</td> <td></td> </tr> <tr> <td></td> <td>Yes</td> <td>No</td> </tr> </table>	Yes	No			Yes	No
Yes	No							
	Yes	No						
What new topics did you learn?	i							
What old topics did you rehearse?	c							
Help provided	e	Tutors comments						







## Debugging

62

### What happens when you make a mistake

As you start writing programs on your own, you will find out that you are bound to make mistakes. Instead of the desired output the autoCad command line is presenting you an error message. Then you must go through your code line by line in order to detect the offending expression.

### Wrong number of parentheses

The most common error is due to the wrong number of parentheses or the wrong positioning of parentheses in the program. Usually you will get the following error messages:

error: malformed list

or

error: extra right paren

There is no easy remedy other than checking your program very carefully for number and placement of brackets. BBEdit provides a tool to check balanced bracket. Highlight one bracket and press B and BBedit shows you the balanced bracket.

Another common error is to forget a space between the operator of an expression and the arguments.

Misspelling of symbols is another common source of errors. All you can do is examine the code carefully line by line. Often it is easier to detect such errors on a piece of paper than on the computer screen, especially working with long programs.

If you get the error message

error: Insufficient string space

usually you didn't provide a closing quotation mark after a string value. This tends to happen using the command function, for example

(command "erase" "all )

missing " after "all

Again, the only solution is to take a careful look.

If you attempt to feed the wrong type of argument to a function you will get

error: bad argument type

AutoLisp helps you to find errors by printing the trouble causing line along with the error message. But sometimes this is not enough. With 'bad argument type' error, for example it is not always obvious which type a variable is during the execution of the program.

The 'pling'-function (exclamation mark) provides a means to check what value the variable in question obtained before the program aborted. Type on the command line:

! <Variable name>

With increasing length of the program, it becomes harder to keep track of the changing values of variables. The princ function can be used to print variables to the command line as the program runs. By placing this function in strategic locations within your program, you can see dynamically what your variables are doing as the programs runs.

(princ "value of a")(princ a)(princ)

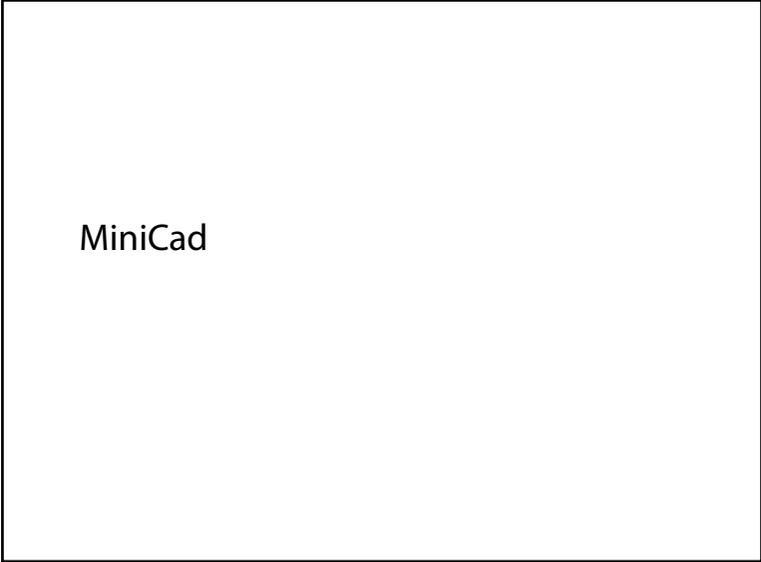
In lengthy program more than one user defined function is used. Sometimes it is important to know when a function is entered, what values passed arguments have at the time of entering and what the result of the function is. The trace function allows you to trace the use of a function during the run of a program. In order to initialise the function type the following on the command line:

(trace <name of function>)

The function untrace disables trace. Programs using recursion are hard to follow. Here the trace function is especially instructive:



minipascal



MiniPascal is a sub set of regular Pascal, and the best introduction to that language is by means of one of the many well written introductory books on the language, in particular: Illustrating Pascal, by Donald Alcock, Cambridge University Press, 1987

MiniPascal omits pointers, arrays with dimensions greater than 2, user defined types, records, and does not support recursion directly (though recursive routines can be written using the database as a stack - see tree program). It has a basic set of datatypes, including the Handle, which permits access to all the entities of a drawing.

## The Syntax

A minipascal program has a block structure. These blocks can be nested inside each other. In MiniPascal the outermost block is a PROCEDURE, which is called with a RUN statement. A procedure block has two parts:

The DECLARATION where it is given a name and the list of variables to be used and a BODY, where the statements to be executed are written.

Pascal insists on pre declaring and typing all variables to be used in a procedure. Putting the wrong type of data into a variable is always an error, except in the special case of real and integer types which can be assigned (and truncated or expanded automatically).

**REAL** a number with a floating point such as 3.14 used for coordinates and dimensions

**INTEGER** a whole number - used for counting things

**STRING** a series of characters- used for sending messages to people via alerts and putting names in records

**BOOLEAN** Just TRUE or FALSE - used in logical expressions as part of a conditional statement

**HANDLE** establishes a connection with a minicad object

In general, most variables used as coordinate values, angles and areas etc. should be REAL. Counters and other whole numbers

should be integer. The HANDLE is used to maintain a connection with an object in the Minicad Database. This will be explored in the tree and other examples.

One and two dimensional arrays are also allowed with the standard syntax:

```
ARRAY [1..n.1..m] of REAL;
```

## Statements

A pascal program is made up of statements. A statement is a line of text ending with a semi-colon (;). There are two types of statement, SIMPLE and COMPOUND

```
a := 1;
```

A simple statement is any line of syntactically correct pascal which does one thing. A compound statement can be either a PROCEDURE, a FUNCTION or a BLOCK statement. A BLOCK statement is a series of statements surrounded by the words BEGIN and END

```
BEGIN
  a := 1;
  b:=2;
  c:= sqrt (a*b);
END;
```

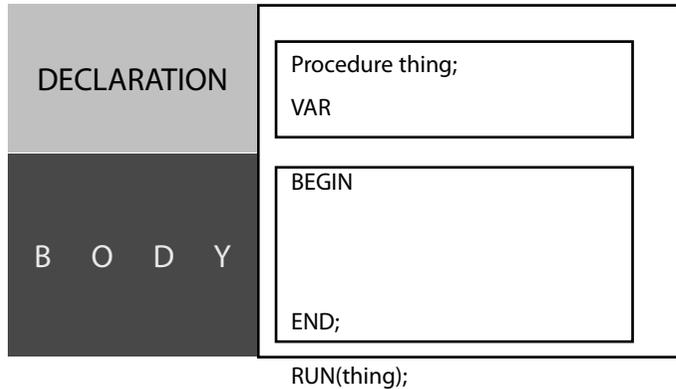
(notice that it ends in a semi colon)

## Procedures

A PROCEDURE is a compound statement which has the following structure:

```
Header part      PROCEDURE somename (parameter list);
Declaration part VAR variablename:variable type;
Body of procedure BEGIN
(a block statement) statements;
END;
```

In Minicad's development environment you write a Procedure, inside which can be nested other pro-



cedures which can contain other procedures, all of which contain compound statements which contain statements.

#### Built-in Library

Much of MiniPascal programming consists of using the built-in library of procedures and functions which are listed as they should be used in the header part - that is with type declarations .

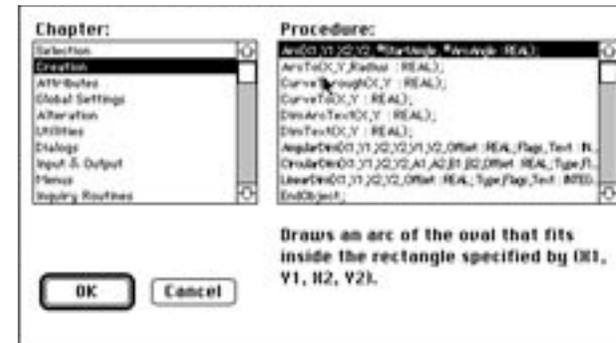
```
Arc(X1,Y1,X2,Y2, #StartAngle, #ArcAngle : REAL);
```

```
LineTo(X,Y : REAL);
```

Functions are listed with their type

```
ObjectType(<Search Criteria>) : INTEGER;
```

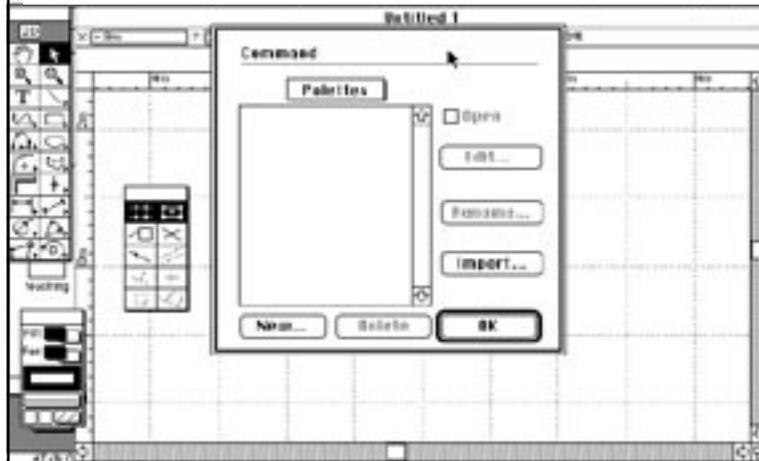
The command editor provides buttons for accessing a library of procedure names which can be pasted directly into your code, so that you don't make any spelling mistakes. The arguments are also listed in 'procedure declaration' style, so that you know how many and what type of arguments to use.



For details on how to use the commands consult the MiniPascal Manual.

The first procedure illustrated is a simple list of MiniCad drawing commands all of which are procedures called with the appropri-

# Quickstart

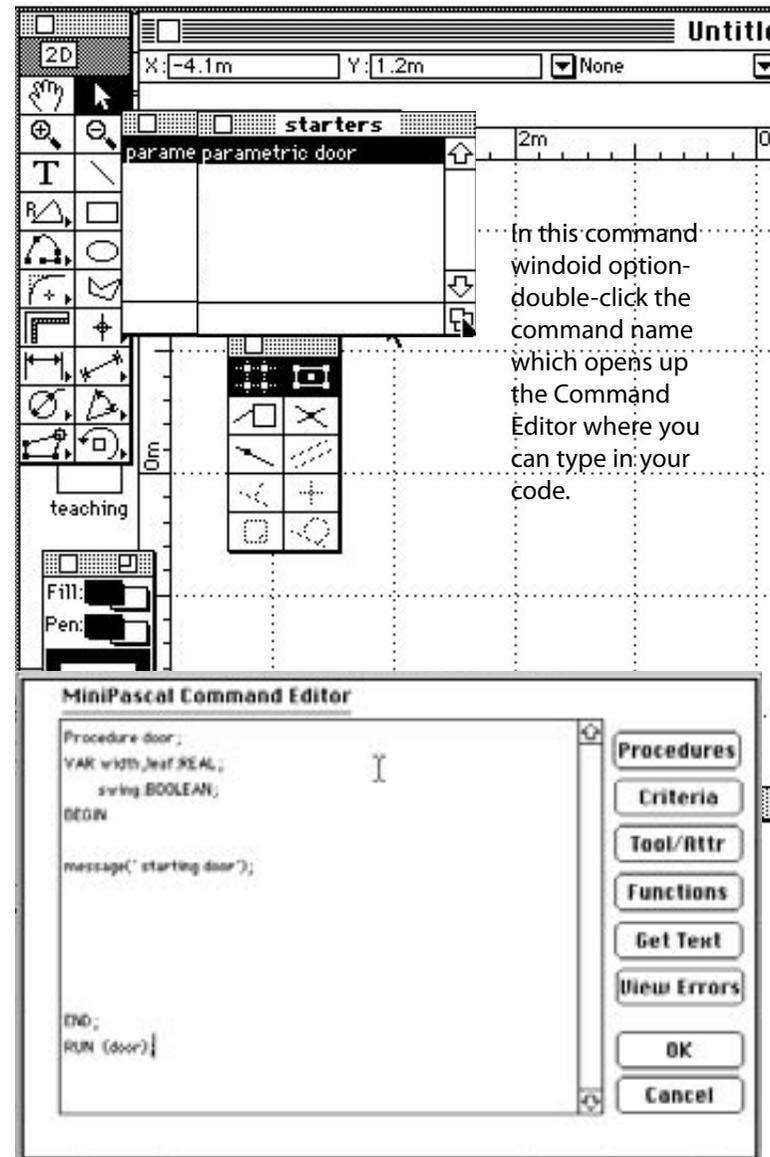
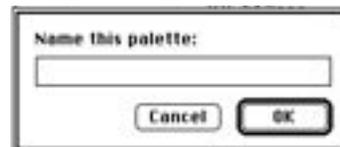


Choose COMMAND... from the Data Menu, which will bring up this dialog box.



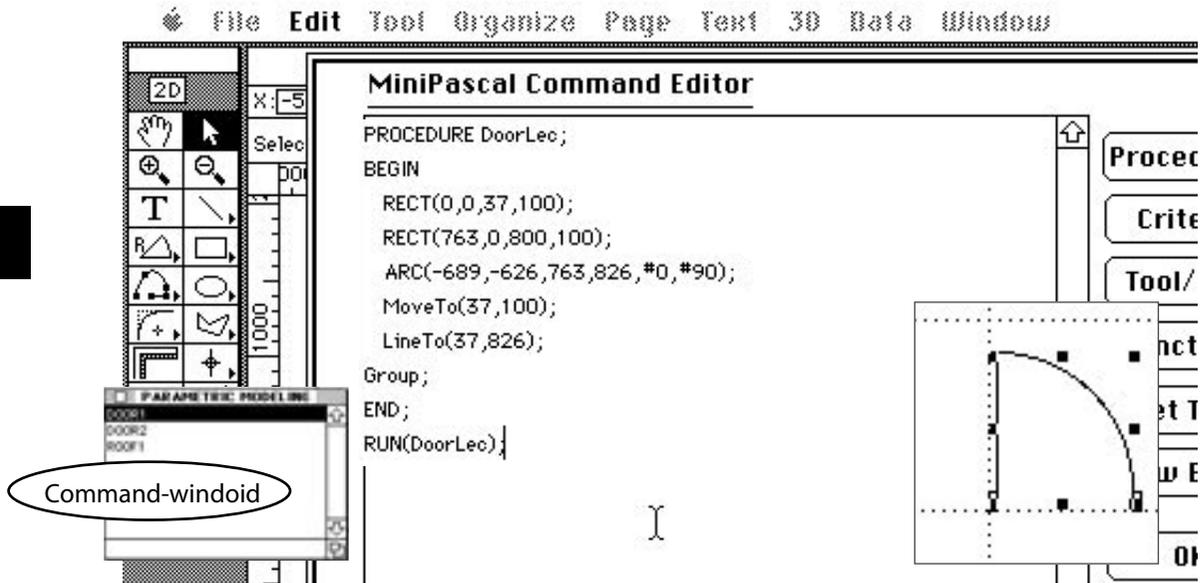
Click on the NEW button to create a new floating Palette, which is a windoid containing your commands.

Then provide a name for your first command, that is the first Mini Pascal procedure that you will write. Hit the OK button.



In this command window option-double-click the command name which opens up the Command Editor where you can type in your code.

1.0 Door



70

This short procedure, called DoorLec, draws a 2d door symbol: door swing and door jamb. NOTICE: This command should be run on a drawing set to metric at a scale 1:10 and units set to mm.

The statements you put in between the BEGIN and END block markers are read and acted upon by MiniCad. Here the statements are all calls to internal MiniCad Graphics functions, which draw two rectangles an arc and a line. There are no variables defined in the declaration part, because there are no variables been used.

```

Rect (0,0,37,100)
Rect (763,0,800,100)
Arc(-689,-626,763,826,#0,#90);
MoveTo(37,100);
LineTo(37,826);
  
```

The Group command ties all objects together in one selectable entity.

```

Group;
  
```

To run the routine DoorLec type the code in, using the MiniPascal Command Editor. Then close the Editor dialogbox and double click

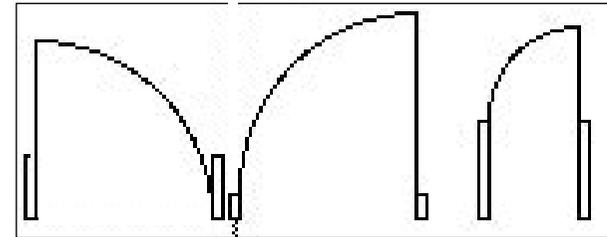
scripting

```
Procedure DoorLec;  
  Begin  
    Rect (0,0,37,100)  
    Rect (763,0,800,100)  
    Arc(-689,-626,763,826,#0,#90);  
    MoveTo(37,100);  
    LineTo(37,826);  
    Group;  
  End;  
Run(DoorLec);
```

minipasca

scripting

## 1.1 Parametric Door



72 The parametric version of doorLec is slightly more advanced. By using the DIALOG functions the user can be asked for information, so that certain parameters are flexible. VARIABLES are used to store them.

```
VAR  
width, thick, doorwidth: REAL;
```

width will contain the width of the doorset, thick the thickness of the wall doorwidth is door width (calculated by subtracting the frame thickness from width).

```
swing:BOOLEAN;
```

swing is of type Boolean (True or False) used in conditional statements. It is TRUE if it's a right hand swing. The variables width, thick and swing are set up:

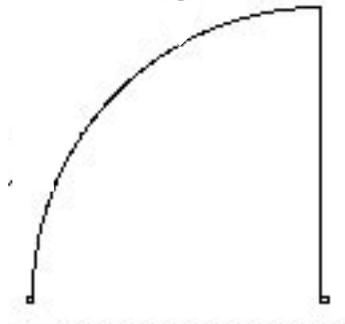
```
width:=DISTDIALOG('Width of doorset:','800');  
thick:=DISTDIALOG('Thickness of wall:','100');  
swing:=YNDIALOG('Right-hand swing?');
```

Using the conditional statements in the language we can execute one part of the program or another

```
IF swing THEN  
  BEGIN  
    ARC(width-37-doorwidth,thick-doorwidth,width-  
37+doorwidth,thick+doorwidth,#90,#90);  
    MOVETO(width-37,d);  
    LINETO(width-37,d+doorwidth);  
  END  
ELSE  
  BEGIN  
    ARC(37-doorwidth,thick-doorwidth,37+doorwidth,  
thick+doorwidth,#0,#90);  
    MOVETO(37,thick);  
    LINETO(37,thick+doorwidth);  
  END;
```

In this example the width of the doorframe is fixed at 37mm and is drawn with two calls to the RECT procedure

```
RECT(0,0,37,thick);
```



Running the command Door2 with differing replies to the dialogs, and hence differing values for s,d and swing.

```
RECT(width-37,0,width-,thick);
```

scripting

```
PROCEDURE Door2Lec;
  VAR
    width,thick,doorwidth:REAL;
    swing:BOOLEAN;
  BEGIN
    width:=DISTDIALOG('Width of doorset;');800);
    thick:=DISTDIALOG('Thickness of wall;');100);
    swing:=YNDIALOG('Right-hand swing?');
    RECT(0,0,37,thick);
    RECT(width-37,0,WIDTH,THICK);
    doorwidth:=width-2*37;
    IF swing THEN
      BEGIN
        ARC(width-37-doorwidth,thick-doorwidth,width-
          37+doorwidth,thick+doorwidth,#90,#90);
        MOVETO(width-37,thick);
        LINETO(width-37,thick+doorwidth);
      END
    ELSE
      BEGIN
        ARC(37-doorwidth,thick-doorwidth,37+
          doorwidth,thick+doorwidth,#0,#90);
        MOVETO(37,thick);
        LINETO(37,thick+doorwidth);
      END;
    GROUP;
  END;
RUN(Door2Lec);
```

minipascal

73

scripting

1.0

When the wall thickness is large, the result are rather odd looking shapes. It would be preferable if the width was some proportion of the wall thickness. To do this we must do three things: 1) Declare a new variable to hold the frame width; 2) Set up a calculation which puts the wallwidth divided by 3 into this new variable; 3) Replace each occurrence of "37" (the current frame width) with the new variable name.

Did you complete assignment 1.0?

Yes No

Did you need help?

Yes No

What new topics did you learn?

What old topics did you rehearse?

Help provided

Tutors comments

1.1	Alter the program to draw a three dimensional parametric door rather than a 2d symbol.							
Did you complete assignment 1.1? Did you need help?		<table border="0"> <tr> <td>Yes</td> <td>No</td> <td></td> </tr> <tr> <td>Yes</td> <td>No</td> <td>No</td> </tr> </table>	Yes	No		Yes	No	No
Yes	No							
Yes	No	No						
What new topics did you learn?								
What old topics did you rehearse?								
Help provided	Tutors comments							

## 2.0 Tree

- 76 Tree structures are an example of a simple fractal: A geometrical figure that consists of identical motif repeating itself on an ever-reduced scale. Here the motif is a trunk separating into two side branches. In the recursive process each branch acts as a trunk for the following two smaller branches and so on. This program tree demonstrates the use of a handle variable to maintain a connection with a line object, so as to recursively draw a tree.

The algorithm:

1. Housekeeping - Delete all objects from drawing:

```
ANGLEVAR;  
selectall;  
deleteobjs;
```

2. Initialise - Draws a vertical line using the LINE (length-angle) command:

```
moveto(0,0);  
Line(1,#90);
```

3. Make a connection -establishes a link with this line .

```
last:=Lnewobj;
```

4. REPEAT- The For- loop controls the number of times the branching process will be applied to candidate lines.

```
FOR loop := 1 to 4 DO  
BEGIN
```

The contained while loop ensures that all but only all the unselected lines at any stage are candidates (the new ones are all selected as they are made - a Minipascal feature).

```
WHILE (NOT Select) DO  
BEGIN
```

As each unselected line is accessed, it is first located, and angle and length info are retrieved using the Hlength and Hangle functions.

```
GetSegPt1(last, X, Y );
```

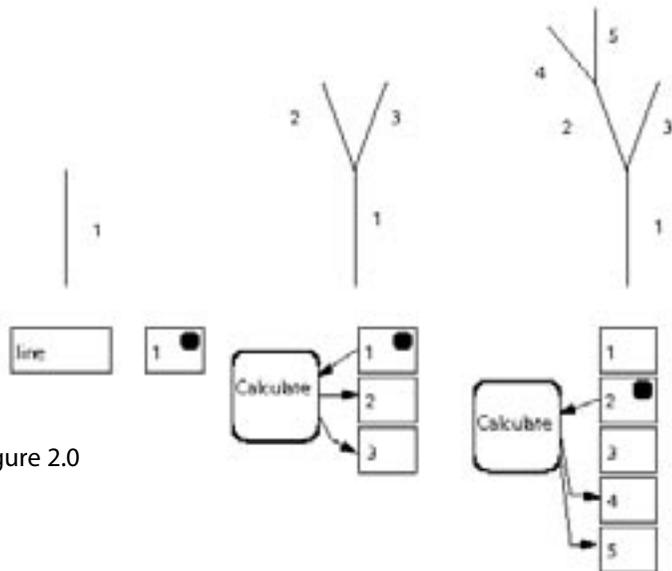


figure 2.0

```
GetSegPt2(last, X1, Y1 );
After info on last line in the MiniCad database is retrieved ang and
length can be calculated:
```

```
Ang:=HAngle(last);
length:=HLength(last)* 0.8;
```

Two new lines branching from the end of the line are drawn and the handle is moved from the current line to the next line.

```
moveto(x1,y1);line(length,#(ang+20));
moveto(x1,y1);line(length,#(ang-20));
last :=nextobj(last);
```

The status of each line is checked as it is accessed.

```
select:= selected (last);
```

if a selcted line is reached, it indicates that the handle is now attached to one of the just created new lines. The while clause monitors this condition by watching the value of select.

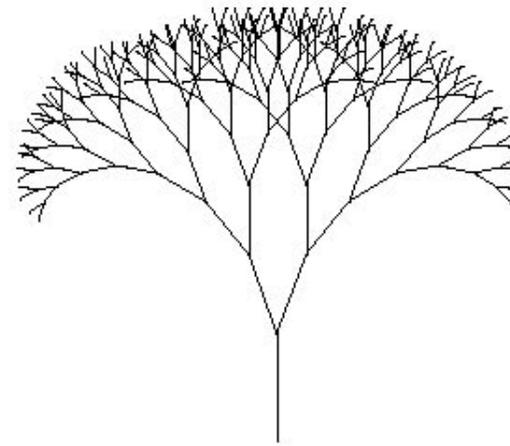


figure 2.1

Once the inner loop is completed in this way,all the new lines are deselected, and the process repeats

```
DSelectAll;
Select:=Selected(last);
```

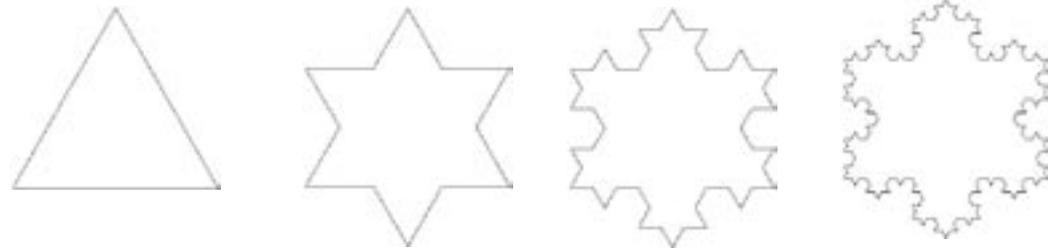
The diagram above shows how the MinPascal database keeps tabs on the lines by storing them in a list. The box with the dot in it is the one used by the program , whose HANDLE is called 'last'. As each object is visited, two more are added to the end of the list, so the one we are dealing with (the black spot) gets progressively further and further behind the end of the list. nextobject moves the handle on to the next object in the list by returning the handle of the current object's next neighbour. In this case the handle is immediately stored in the original variable that was used to reference the original object.

scripting

```
procedure tree;
  VAR last: HANDLE;
      x,y,x1,y1,ang,length: REAL;
      select: BOOLEAN;
      loop: INTEGER;
  BEGIN
    ANGLEVAR;
    selectall;deleteobjs;redraw;
    moveto(0,0);
    Line(1,#90);
    last:=Lnewobj;
    DSelectAll;
    Select:=Selected(last);
    FOR loop := 1 to 4 DO
      BEGIN
        WHILE (NOT Select) DO
          BEGIN
            GetSegPt1(last, X, Y );
            GetSegPt2(last, X1, Y1 );
            Ang:=HAngle(last);
            length:=HLength(last)* 0.8;
            moveto(x1,y1);line(length,#(ang+20)) ;
            moveto(x1,y1);line(length,#(ang-20)) ;
            last :=nextobj(last);
            Select:=Selected(last);
          END;
          DSelectAll;
          Select:=Selected(last);
          redraw;
        END;
      END;
    END;
  run(tree);
```

2.0	The number of times round the loop, the length of the line and the reduction factor are all fixed values in this procedure. Use the DIALOG functions to get values from the user.				
Did you complete assignment 2.0.c? Did you need help?  What new topics did you learn?	<table border="0"> <tr> <td>Yes</td> <td>No</td> </tr> <tr> <td>Yes</td> <td>No</td> </tr> </table>	Yes	No	Yes	No
Yes	No				
Yes	No				
What old topics did you rehearse?					
Help provided	<p>Tutors comments</p>				

### 3.0 Koch Curve



80

This has been written to parallel the AutoLisp example, except that instead recursively calling a line drawing routine we use a procedure to draw the 4 new lines, then rub out the original one. The Koch Curve example uses the same select/unselect method to control the loop as the tree program.

#### The algorithm

1. Housekeeping- Clear the drawing.

```
SelectAll;
DeleteObjs;
Redraw;
```

2. User input- Ask user for recursion depth.

```
numIter:=IntDialog("number of recursions","3");
```

3. Initial figure- Draw three lines to form the initial triangle.

```
moveto(0,0); line(9,#0);
line(9,#120); line(9,#240);
```

4. Repeat-The For-loop controls the iteration depth of the curve according to variable numIter.

```
FOR Count :=1 TO NumIter DO
BEGIN
```

The nested while-loop ensures that only the unselected lines are subject of treatment.

```
WHILE (NOT Select) DO
BEGIN
```

After the functions HLength and HAngle are used to get information on the line four new lines are inserted and the old one deleted.

```
length:=HLength( this );
ang := HAngle ( this );
LINE (length / 3, #ang );          LINE (length / 3, #ang -
60 );
LINE (length / 3, #ang + 60 );    LINE (length / 3, #ang );
this := NextObj( this );
DelObject (PrevObj ( this ));
```

```

PROCEDURE Koch7;
  VAR      Numlter :      INTEGER;
          X,Y,length,ang : REAL;
          this :      HANDLE;
          Select :      BOOLEAN;

```

```

BEGIN
  SelectAll;DeleteObjs;REDRAW;
  Numlter := INTDIALOG("No. of Iterations:'3");
  MoveTo(0,0);
  line(9,#0);
  this :=LObject;
  line (9,#120);
  line (9,#240);
  Select := Selected ( this );
  REDRAW;

```

```

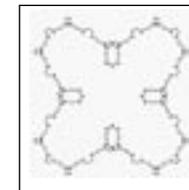
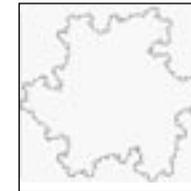
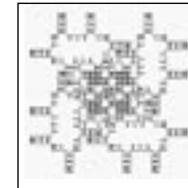
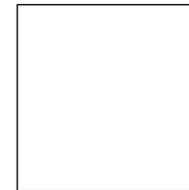
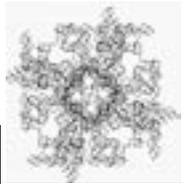
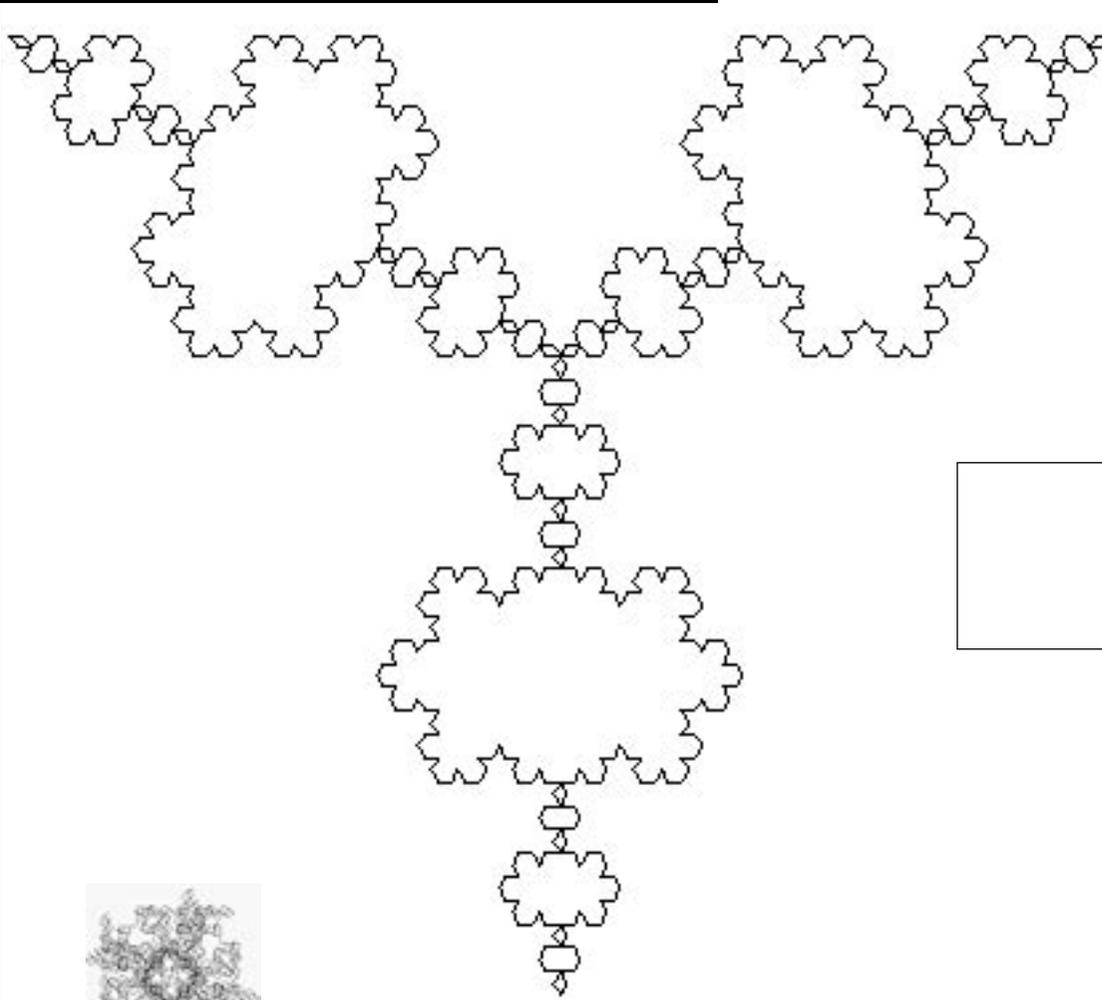
FOR Count :=1 TO Numlter DO

```

```

      BEGIN
        WHILE (NOT Select) DO
          BEGIN
            GetSegPt1(this,X,Y);
            length:=HLength( this );
            ang := HAngle ( this );
            AngleVar;
            MOVETO(X,Y);
            LINE ( length / 3 , #ang );
            LINE ( length / 3 , #ang - 60 );
            LINE ( length / 3 , #ang + 60 );
            LINE ( length / 3 , #ang );
            this := NextObj( this );
            DelObject (PrevObj ( this ) );
            Select := Selected ( this );
          END;
          DSelectAll;
          Select := Selected ( this );
        END;
        REDRAW;
      END;
END;
RUN(Koch7);

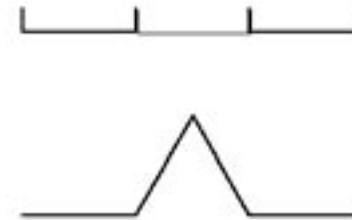
```



Koch's Variations

3.0	On the opposite page are a selection of possible variations. The best known one, the Koch island (big image), involves only alteration of the direction of the angle in the line(...) command. For other variations the reduction ratio of the lines has to be recalculated. Here this is done for you for the so called Koch Forest. Implement the calculation in the program.						
Did you complete assignment 3.0? Did you need help?  What new topics did you learn?	<table border="0"> <tr> <td>Yes</td> <td>No</td> <td></td> </tr> <tr> <td></td> <td>Yes</td> <td>No</td> </tr> </table>	Yes	No			Yes	No
Yes	No						
	Yes	No					
	Once all unselected lines have been traversed, the new lines are deselected and they become candidates for the treatment.						
What old topics did you rehearse?							
Help provided	Tutors comments						

Calculation of reduction ratio

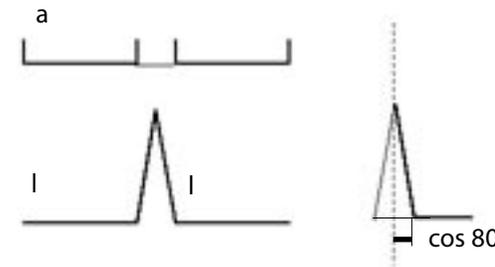


In the original koch curve example the length of the motif is three times the length of a line. Therefore the reduction factor is 1/3.

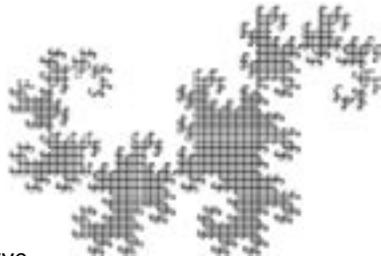
The Koch forest motif has the length a which can be calculated as followed:

$$a = l + l + (2 * \cos 80)$$

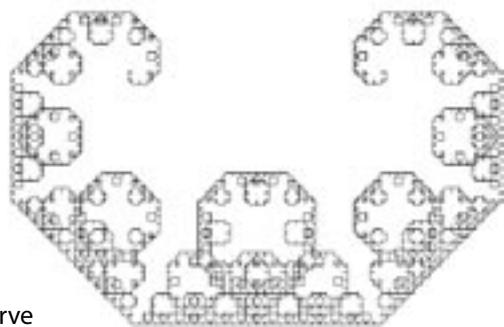
see diagram.



### 3.1 Dragon Curve



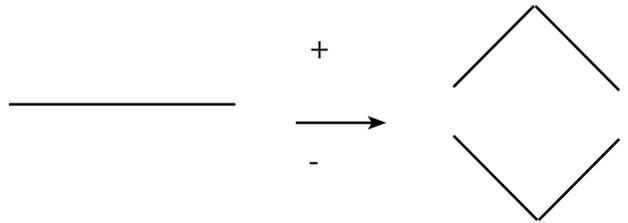
Dragon Curve



levyCurve

84

The Dragon Curve and the Levy Curve are based on the replacement of lines by the same motif. The frenchman Paul Levy was one of the first mathematicians to do research on what is today known as fractals. The interesting thing about the Levy Curve, as shown in figure 3.1.1, is that in it all previous stages can be found, starting with the first iteration at the beginning of the meandering line to the n-1 iteration in the center.



The program Koch is very similar to the previous example Dragon curve. Both are based on the replacement of a original line by a number of new lines, here two. The dragon curve is a development of the Levy curve where the replacement rule is modified to take place alternatively one side or the other of the original line. This is done using the variable sign.  $sign := -sign$  changes from 1 to -1

scripting

```
PROCEDURE Dragon;
  VAR      Numlter,Sign :      INTEGER;
          X,Y,length,ang,SQR : REAL;
          handle :          HANDLE;
          Select :          BOOLEAN;
```

```
  BEGIN
    Numlter :=INTDIALOG('No. of Iterations:"6');
    SQR :=1/SQRT(2);
    MoveTo(0,0);
    LineTo(1000,0);
    handle := LObject;
    DSelectAll;
    Select :=Selected(handle);
    REDRAW;
```

```
  FOR Count :=1 TO Numlter DO
    BEGIN
      Sign:=-1;
      WHILE (NOT Select) DO
        BEGIN
          GetSegPt1( handle ,X,Y);
          length :=HLength(handle);
          ang := HAngle(handle);
          AngleVar;
          MOVETO(X,Y);
          LINE( length * SQR, #ang + 45 * Sign);
          LINE( length * SQR , #ang - 45 * Sign);
          Sign :=-Sign;
          handle :=NextObj(handle);
          DelObject(PrevObj(handle));
          Select:=Selected(handle);
        END;
      DSelectAll;
      Select :=Selected(handle);
      REDRAW;
    END;
  Sysbeep;
  END;
  RUN(Koch6);
```

minipascal

85

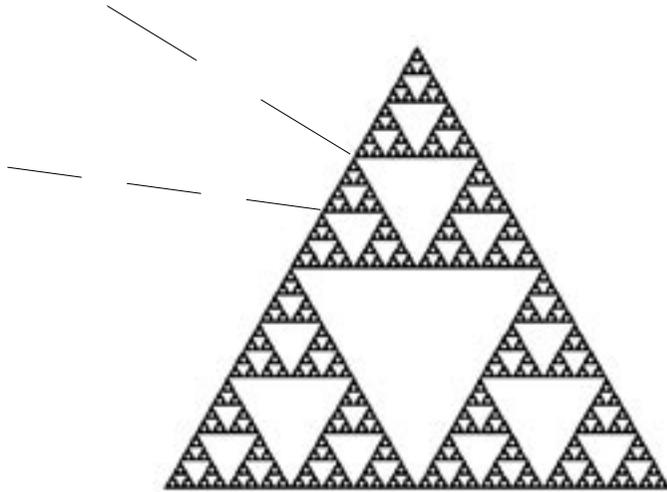
scripting

## 4.0 Sierpinski Gasket

86

The Sierpinski gasket is another example of a fractal. Similar to the Koch curve, its creation is based on the replacement of its constituent parts. The algorithm here uses an approach that Peitgen et al. call the Multiple Reduction Copy Machine (MCRM). The user enters the value for the loop controller  $NG$ . The initial triangle is drawn with side length 1000 (variable  $L$ ). The For-loop carries out the scaling and duplicating of the current object (initially a black triangle) into three copies, which are then grouped. This group is then acted on in the same way, and so on for  $NG$  times.

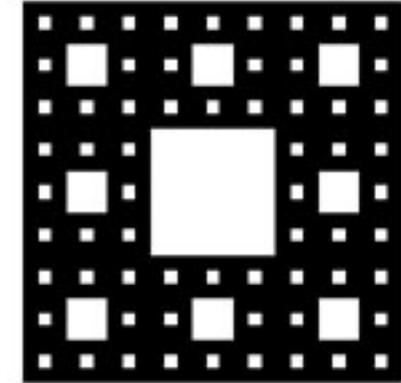
Note: Duplicate offsets must be checked in the preferences dialog box.



scripting

```
Procedure MRCMGasket;  
VAR  
  NG,n      : INTEGER;  
  L         : REAL;  
BEGIN  
  selectall;DeleteObjs;  
  Fillpat(2);  
  CLOSEPOLY;  
  L:=1000;  
  NG:=INTDIALOG('No. of Levels:');  
  POLY(L,#0,  
  L,#120,  
  L,#240);  
  L:=L/2;  
  FOR n:= 1 TO NG DO  
    BEGIN  
      SCALE(1/2,1/2);  
      DUPLICATE(L,#-120);  
      DUPLICATE(L,#0);  
      selectall;  
      GROUP;  
    END;  
  END;  
RUN(MRCMGasket);
```

4.0	<p>Try Modifying the above routine to produce the Sierpinski carpet. This will start with a square, have a scaling factor of one third, and use seven duplicate procedures in the loop.</p>
<p>Did you complete assignment 4.0? Did you need help?</p> <p>What new topics did you learn?</p>	<p>Yes    No Yes    No</p>
	<p>What old topics did you rehearse?</p>
Help provided	<p>Tutors comments</p>



4.1	Instead of positioning the three triangles using move you can also use rotation and mirroring as additional transformations. The first iteration step looks similar to the original example, but then it should develop in a quite different direction.						
Did you complete assignment 4.1? Did you need help?	<table border="0"> <tr> <td>Yes</td> <td>No</td> <td></td> </tr> <tr> <td></td> <td>Yes</td> <td>No</td> </tr> </table>	Yes	No			Yes	No
Yes	No						
	Yes	No					
What new topics did you learn?							
What old topics did you rehearse?							
Help provided	<p>Tutors comments</p>						

Rather different pattern emerge if rotation is used in addition to the move transformation. In the two examples shown on this page the initial figure was a black square and not a triangle.



## 5.0 Pythagoras Tree

90

The routine `pythagotree` uses the same method as the Koch and tree examples, getting the angle & length of a line and then drawing 5 new lines to make a square and triangle. However, in this example the point data is stored and retrieved from the Polygon data structure.

Each call to the procedure reads the point data of the `indexth` vertex of the polygon whose handle is passed to the procedure as this.

The algorithm:

1. Housekeeping- As usual, clean the drawing and set variables, here by prompting the user.

```
SelectAll;DeleteObjs;
a:= AngDialog('angle', '45');
gens := IntDialog('Recursions', '3');
```

2. Initialise- Draw a square, get its handle. (see diagram A)

```
closepoly;
poly(0,0,0,1000,1000,1000,1000,0);
thepoly := LNewObj;
```

C  
E

3. Construct next- Get the angle and length of the second line of this polygon and construct two polygons- a square and a triangle with Procedure (`drawsquareNtriang`). (B)

```
DrawsquareNtriang(thepoly,1000,2);
```

4. Repeat- In a For-loop the following steps are repeated for as many times as we asked for (in variable `gens`).

• Move the handle on to the triangle just drawn. (C)

```
thepoly:=nextobj(thepoly);
thepoly:=nextobj(thepoly);
```

• Use `drawsquareNtriang` with the first line of the triangle. (D)

```
DrawsquareNtriang(thepoly,200+((gens-g)*200),1);
```

• Use `drawsquareNtriang` with the second line of the triangle. (E)

```
DrawsquareNtriang(thepoly,200+((gens-g)*200),2);
```

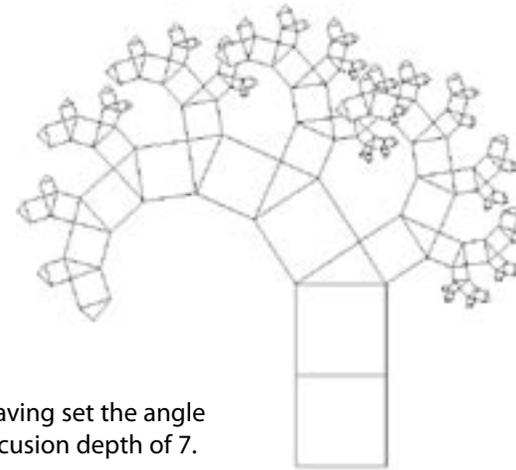
```

procedure pythagotree;
var thepoly: HANDLE;
    a: REAL;
    skip, gens, g: INTEGER;

Procedure DrawsquareNtriang(this:HANDLE;index:INTEGER);
VAR x,y,x1,y1,ang,length,newlength:REAL;
    aline: VECTOR;
BEGIN
    anglevar;
    Dselectall;
    GetPolyPt(this,index, X, Y );
    GetPolyPt(this,index+1, X1, Y1 );
    aline[1]:= x1-x;aline[2]:=y1-y;aline[3]:=0;
    ang:=Vec2Ang(aline) ;
    length:=Distance(X, Y, X1, Y1 );
    closepoly;
    beginpoly;
        addpoint(x,y);
        addpoint(length,#(ang+90));
        Penloc(x,y);
        addpoint(length,#ang);
        Penloc(x1,y1);
        addpoint(length,#(ang-90));
    endpoly;
    closepoly;
    beginpoly;
        addpoint(x,y);
        newlength:=cos(deg2rad(a))*length;
        addpoint(newlength,#(ang+a));
        addpoint(x1,y1);
    endpoly;
END;

```

scripting



The result having set the angle to 30 and recursion depth of 7.

```

BEGIN
    SelectAll;DeleteObjs;
    a:= AngDialog('angle ', '45' );
    gens := IntDialog('Recursions', '3' );
    closepoly;
    poly(0,0,0,1000,1000,1000,1000,0);
    thepoly := LNewObj ;
    DrawsquareNtriang(thepoly,1000,2);
    FOR g := 1 TO gens DO
        BEGIN
            thepoly:=nextobj(thepoly); thepoly:=nextobj(thepoly);
            DrawsquareNtriang(thepoly,1);
            DrawsquareNtriang(thepoly,2);
        END;
    END;
END;
RUN(pythagotree);

```

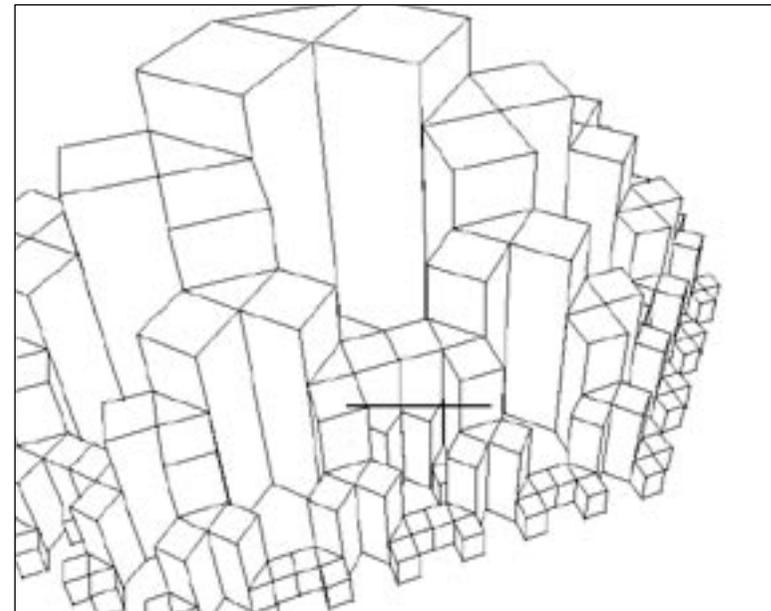
minipascal

91

scripting

## 5.1 Pythagoras Tree 3d

- 92 With some alterations it is possible to transform the flat tree outline into a three dimensional object. By extruding the lines onto another layer they can be turned into vertical fences. So viewed in an axonometric view a composition of differently dimensioned cuboids and prisms appears on the screen (see image). The code remains the same, the boxed bits are the alterations.



```

Procedure pythagotree;
VAR
  thepoly:HANDLE;
  a:REAL;
  skip,gens,g:INTEGER;
Procedure DrawsquareNtriang(this:HANDLE; height:REAL;
                           index:INTEGER);
VAR x,y,x1,y1,ang,length,newlength:REAL;aline:VECTOR;

```

```

BEGIN
  anglevar;
  Dselectall;
  GetPolyPt(this,index, X, Y );
  GetPolyPt(this,index+1, X1, Y1 );
  aline[1]:= x1-x;aline[2]:=y1-y;aline[3]:=0;
  ang:=Vec2Ang(aline) ;
  length:=Distance(X, Y, X1, Y1 ) ;
  closepoly;
  beginpoly;
  addpoint(x,y);
  addpoint(length,#(ang+90));
  Penloc(x,y);
  addpoint(length,#ang);
  Penloc(x1,y1);
  addpoint(length,#(ang-90));
  endpoly;
  closepoly;
  beginpoly;
  addpoint(x,y);
  newlength:=cos(deg2rad(a))*length;
  addpoint(newlength,#(ang+a));
  addpoint(x1,y1);
  endpoly;

```

```

Domenu(Mcopy,nokey);
layer('3d');
Beginxtrd(0,height);
Domenu(Mpaste,optionkey);
Endxtrd;
layer('2d');

```

```
END;
```

```

BEGIN
  layer('2d');
  SelectAll;DeleteObjs;
  layer('3d');
  SelectAll;DeleteObjs;
  layer('2d');
  SelectAll;DeleteObjs;
  a:= AngDialog('angle ', '45') ;
  gens := IntDialog('Recursions', '3') ;
  closepoly;
  poly(0,0,0,1000,1000,1000,1000,0);
  thepoly := LNewObj ;
  DrawsquareNtriang(thepoly,1000,2);
  FOR g := 1 TO gens DO
  BEGIN
    thepoly:=nextobj(thepoly); thepoly:=nextobj(thepoly);
    DrawsquareNtriang(thepoly, 200+((gens-g)*200), 1);
    DrawsquareNtriang(thepoly, 200+((gens-g)*200), 2);
  END;
END;
RUN(pythagotree);

```

5.0	Alter the method of calculation for the height. Currently it is inversely proportion to the generation number + 200. Change it to some other relationship.
Did you complete assignment 5.0? Did you need help?  What new topics did you learn?	Yes    No Yes    No
	What old topics did you rehearse?
Help provided	
Tutors comments	

5.1	How would you alter the program so it grew a tree off each side of the initial square.						
Did you complete assignment 5.1? Did you need help?	<table border="0"> <tr> <td>Yes</td> <td>No</td> <td></td> </tr> <tr> <td>Yes</td> <td>No</td> <td>No</td> </tr> </table>	Yes	No		Yes	No	No
Yes	No						
Yes	No	No					
What new topics did you learn?							
What old topics did you rehearse?							
Help provided	<p>Tutors comments</p>						

## 6.0 Wander

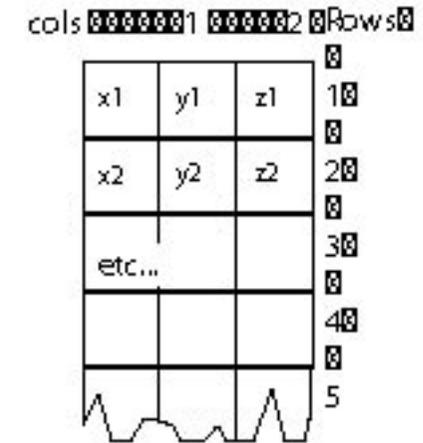


diagram 6.1

96

This program uses VECTORS which are Minipascal's way of storing and manipulating points. The array points is defined as a table of 3 columns by 100 rows, the columns represent x y & z coordinates, each row is the insertion point of a cube (see diagram 6.1).

The random function is equivalent to the AutoLisp example, and returns a random number between 0 and 1.

Function same returns true if the coordinates in the indexth row of array points are the same as those in the global vector variable nextpoint.

Insertcube sets the new vector nextpoint to be thispoint, and adds it to the array points (after incrementing the counter total), then draws the box.

algorithm

1. Housekeeping- After the usual housekeeping and user input

```
selectall;deleteobjs;
seed:=intdialog('random seed','333');
max:= intdialog('insertions','1');
total:=0;
```

the first point is set at 0,0,0.

```
FOR loop:= 1 TO 3 DO thispoint[loop]:=0;
```

2. Choose- A random choice between 1 and 6 is used to calculate the next position for an insertion,

```
choice:= round( rand * 6) +1;
```

3. Repeat- The REPEAT...UNTIL loop controls the number of insertions, based on user input of max.

```
REPEAT
  nextpoint := thispoint; {copy current position}
  choice:= round( rand * 6)+1;
  IF choice = 1 then nextpoint [3] := thispoint [3] -1;
  .....
```

The nested while-loop uses the same logic as in the AutoLisp example to check that the new point is not the same as any point stored in the array points.

```
WHILE((count <= total) AND(NOT same(count)))DO
```

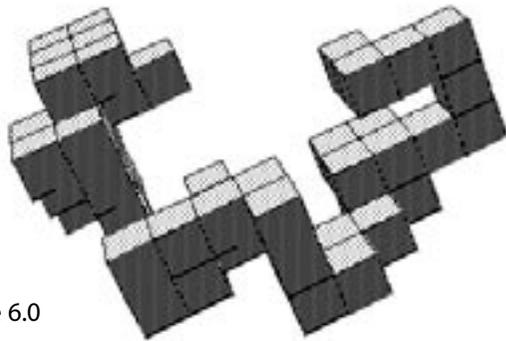
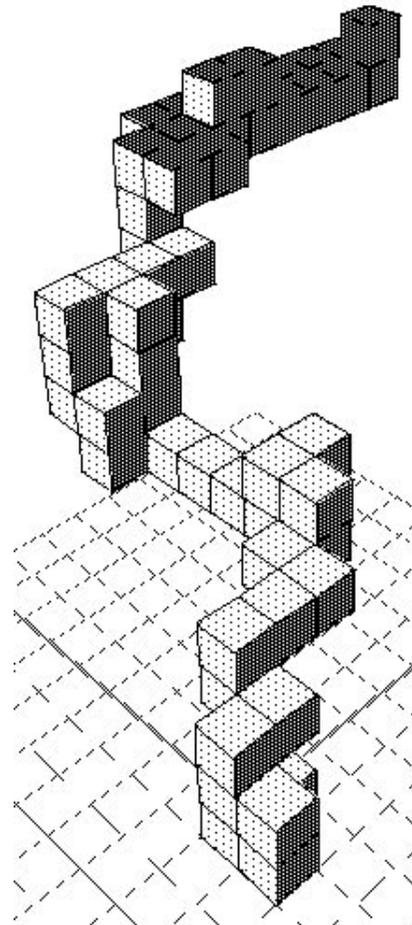


figure 6.0



count:=count +1;

The array points is used to store the insertion points of each cube, just like BIGLIST in the Autolisp example. Individual points are stored in VECTORS, which are Minipascals own data type for x y z triplets, and are equivalent to a one dimensional array of three elements.

The WHILE loop runs through each row of this table (using count as the index to the row) and the boolean function same looks along this row, comparing each element with each element of the vector this.

```
same := ((points[1,index]=round(this[1]))
        AND(points[2,index]=round(this[2]) )
        AND(points[3,index]=round(this[3] )));
```

This reads as : "The first value in the array points on row index is the same as the rounded value of the first element in vector this AND the second value in the array points on row index is the same as the rounded value of the second element in vector this AND the third value in the array points on row index is the same as the rounded value of the third element in vector this."

This is an example of a boolean expression, because the answer is either true or false. Notice that the value is assigned to the name of the function, not a variable. This is so the function can transfer the result back to the calling statement:

```
(NOT same(count,nextpoint))
```

This means that if the two sets of coordinates are the same then same will return TRUE. In which case NOT TRUE will equal FALSE. If

6.0	Rewrite Insertcube so that instead of inserting a cube of side 1, 1, 1, 1 it draws a cross as shown in figure 6.2.						
Did you complete assignment 6.0? Did you need help?	<table border="0"> <tr> <td>Yes</td> <td>No</td> <td></td> </tr> <tr> <td></td> <td>Yes</td> <td>No</td> </tr> </table>	Yes	No			Yes	No
Yes	No						
	Yes	No					
What new topics did you learn?							
What old topics did you rehearse?							
Help provided	<div style="display: flex; justify-content: space-between;"> <span data-bbox="577 1193 676 1407">e x e r c i s e</span> <span data-bbox="779 1193 810 1474" style="writing-mode: vertical-rl; transform: rotate(180deg);">Tutors comments</span> </div>						

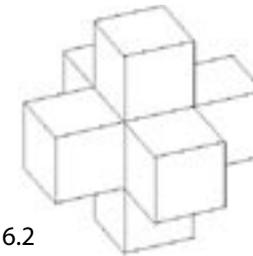


figure 6.2

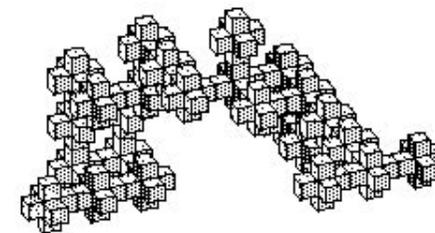


figure 6.3

Something similar to figure 6.3 should appear on the screen after wander is altered as asked in exercise 6.0.

```

procedure wander;
  VAR      points :      array[1..3,1..100] of INTEGER;
          thispoint,
          nextpoint:      VECTOR;
          loop, count, total,
          choice, seed, max :  INTEGER;
          FUNCTION Rand :  REAL;

  VAR      x :          REAL ;
          z :          INTEGER;

  BEGIN
    x := 2197 * seed;
    z := TRUNC ( x / 4096 );
    seed := TRUNC ( x - ( 4096 * z ) );
    rand := ( seed / 4096 );

  END;

  function same ( index : integer ) : BOOLEAN;
  BEGIN
    same := ((points [1,index ] = round( nextpoint [1]))
            AND (points [2,index ] = round( nextpoint [2] ) )
            AND (points[3,index]= round ( nextpoint[3] )));

  END;
PROCEDURE insertcube;
  VAR      L:          INTEGER;
  BEGIN
    total := total + 1;
    thispoint := nextpoint;
    FOR L := 1 TO 3 DO points[L,total] := round( thispoint [L]);
    beginxtrd(thispoint[3],thispoint[3]+1);
    rect(thispoint [1], thispoint [2],
        thispoint [1]+1, thispoint [2]+1);
    endxtrd;
    redraw;
  END;
BEGIN

```

```

selectall;deleteobjs;
seed := intdialog('random seed',333');
max := intdialog('insertions',1');
total :=0;
FOR loop := 1 TO 3 DO thispoint [ loop ]:=0;
insertcube;
  scripting REPEAT
    nextpoint := thispoint; {copy current
position}
    choice:= round( rand * 6)+1;
    IF choice = 1 then nextpoint [3] := thispoint [3] -1;
    IF choice = 2 then nextpoint [1] := thispoint [1] -1;
    IF choice = 3 then nextpoint [3] := thispoint [3] +1;
    IF choice = 4 then nextpoint [1] := thispoint [1] +1;
    IF choice = 5 then nextpoint [2] := thispoint [2] - 1;
    IF choice = 6 then nextpoint [2] := thispoint [2] +1;
    count:=1;
    WHILE ( ( count <= total ) AND ( NOT same
( count ))) DO count := count +1;
    IF count > total then insertcube;
  UNTIL total >= max;
  END;
run(wander);

```

minipascal

99

exercise

scripting

## Debugging

100

ness into the realm of natural science, but also opened up the way to the study of non-linear systems, such as the iteration of  $x \rightarrow x^2 + c$  or the weather.

The first thing to realise is that it is not surprising to make a mistake, but more or less inevitable. In really disastrous conditions you may crash MinCad itself due to the parser getting indigestion from your code.

There are two kinds of mistakes:

### 1) Syntax errors

The first type is fairly simple to correct, providing that you have a suitable reference work. It is important to realise that Pascal is rather useless at telling you what is actually the matter, and one error will inevitably generate several error messages. In general the actual error will be one of

- a. leaving out a semicolon or a comma
- b. mis-spelling a procedure or function name
- c. forgetting to match your BEGINS and ENDS

The error output file is displayed with the "View Errors" button in the Minipascal Editor.

### 2) Syntactically correct code that doesn't do what you want

The main way to sort this sort of thing out is to try and discover what is actually happening by using the `Writeln` (short for write line) to write out the contents of your variables to the output file.

```
GetSegPt1(last, X, Y);
GetSegPt2(last, X1, Y1);
writeln('x y & x1 y1';x,y,x1,y1);
and
aline[1]:= x1-x;aline[2]:=y1-y;aline[3]:=0;
ang:=Vec2Ang(aline);
writeln('angle = ', ang);
```

In this fragment the two `writeln` statements output the value of the four coordinates `x y x1 & y1` on one line, and the calculated angle on the next. The variables are listed out after an optional bit of text

`('x y & x1 y1'`,

which is used to identify what would otherwise be an anonymous







|

|

|

|

g.d.l.

|

103

|

|

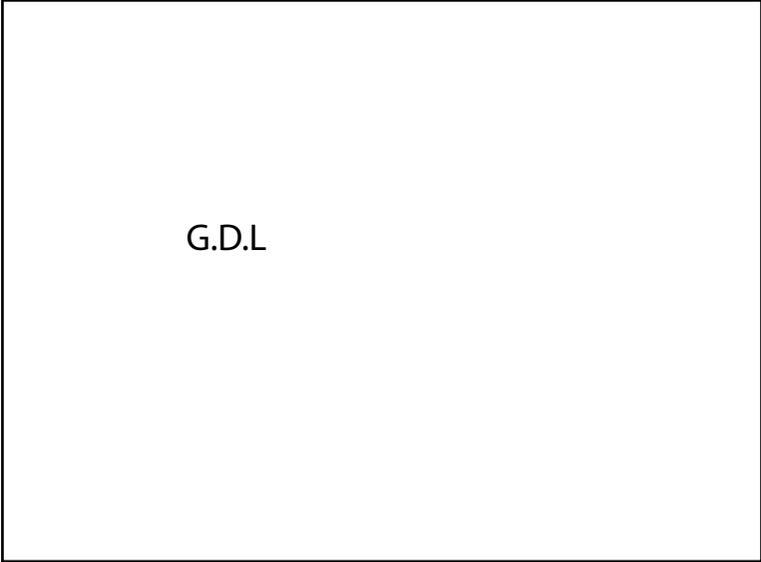
|

|

|

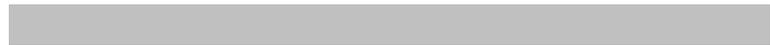
|

|



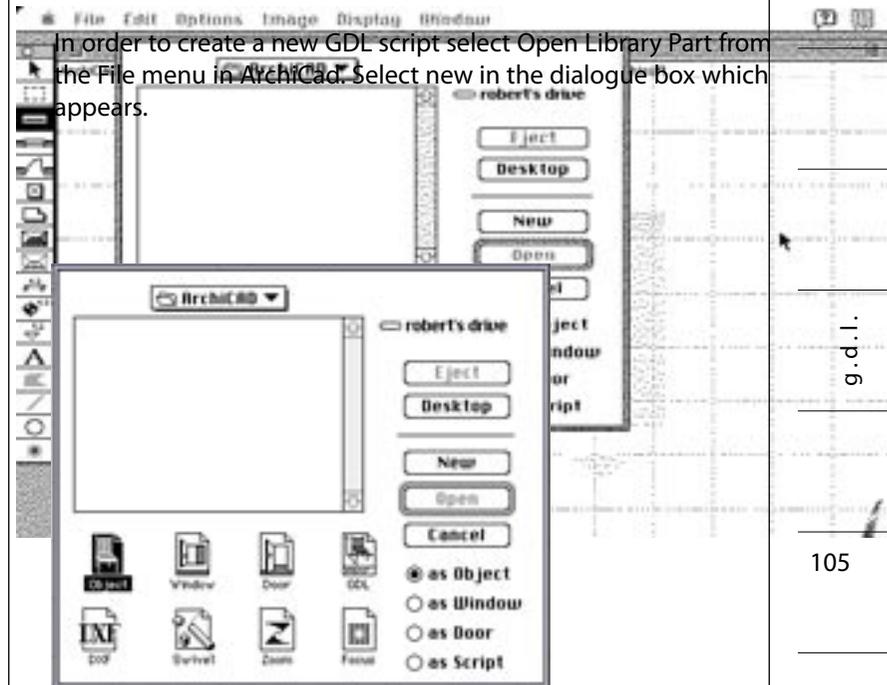
G.D.L

104



## Quickstart

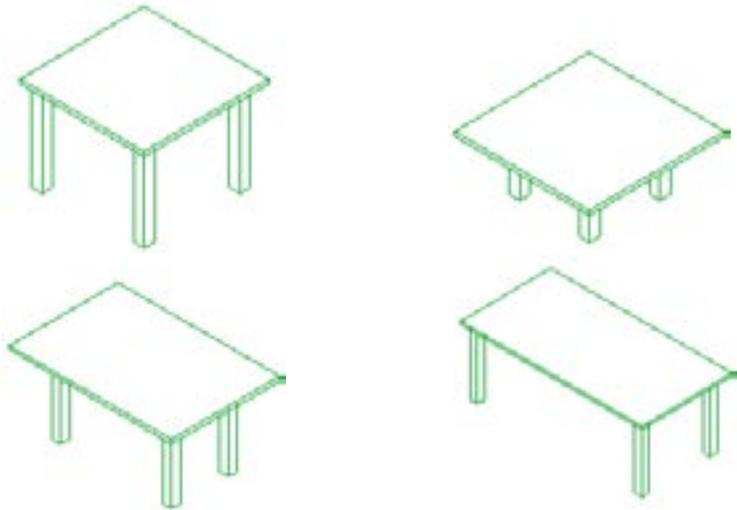
In order to create a new GDL script select Open Library Part from the File menu in ArchiCAD. Select new in the dialogue box which appears.



Click the new-button will bring you to the window on the left. From here you operated GDL: Define parameter, type the script, and run the script.



## 1.0 Parametric table



106

This simple program creates a parametric rectangular table using six parameters, the brick command and simple transformation. The algorithm is simple: Go to certain position in space, draw brick, move to next position, draw brick, move to next position..... END. The six parameters define the dimension of the table: a - the length, b - the width, c - the height, d - inset, e - thickness of legs, f - thickness of the table top.

```

A@CZ (c-1)
Brick e,b,f
Set 1
add d,0,0
brick e,e,(c-1)
Set 1
add d,(b-d-e),0
brick e,e,(c-1)
add c-(2*d)+e
brick e,e,(c-1)
add j -(b-(2*d)-e)
brick e,e,(c-1)
Set top
End
  
```

The dialog box above prompts the user for the values of the parameters. The variables A and B are entered beside the bed symbol, the rest in the scrolling window below

Click the GDL Script - button: The GDL script can be edited in the text window. Clicking the 3D View -button brings up the image.



scripting

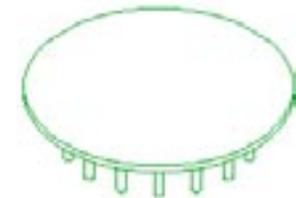
```
Addz (c-f)
Brick a,b,f
del 1
add d,d,0
brick e,e,(c-f)
del 1
add d,(b-d-e),0
brick e,e,(c-f)
addx a-(2*d)-e
brick e,e,(c-f)
addy -(b-(2*d)-e)
brick e,e,(c-f)
del top
End
```

g.d.l.

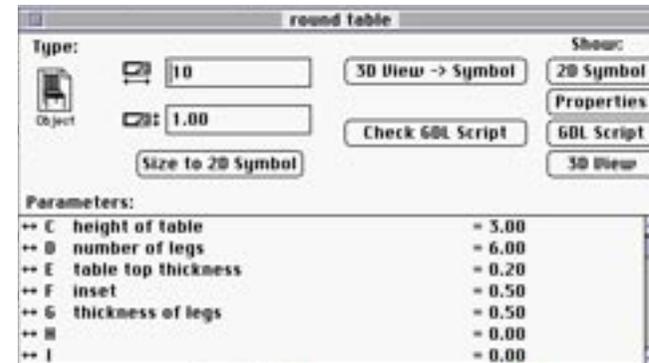
107

scripting

## 1.1 Round table



108 A further example for a scripted parametric objects includes the use of a loop. This time six variables are in use: a- diameter of the table top, c- table height, d- the number of legs, e- table top thickness, f - inset, g- the thickness of the legs. The algorithm is slightly more elegant than the one in the previous example: After the table top is drawn the legs are created in a FOR NEXT-loop. This allows the user to choose any number of legs.



Different outputs produced by the program Round Table varying in the number of table legs, inset, and length of legs.

scripting

```
addz (c-e)
cylind e, (a/2)
del 1
For s=1 to d step 1
  addx ((a/2)-(e+f))
  cylind (c-e), (g/2)
  del 1
  rotz (360/d)
next s
del top
end
```

g.d.l.

scripting

1.0	e	By changing the parameters you can get all sorts of tables. Try it
Did you complete assignment 2.0.c? Did you need help?	s	Yes No Yes No
What new topics did you learn?	i	
What old topics did you rehearse?	c	
Help provided	e	Tutors comments

1.1	e	
Did you complete assignment 2.0.c? Did you need help?		Yes    No Yes    No
What new topics did you learn?		
i		
What old topics did you rehearse?		
e		
Help provided		
e		Tutors comments

## 2.0 Recursive block adding

112 GDL supports recursion, once a macro has been saved and named, by using the CALL syntax. If the macro called is the same one as the one loaded into Archicad, then recursive algorithms can be developed. The first example is one where each instance of the macro calls itself 5 times, after moving the coordinate system and reducing the side length to place a new block on each face of the current block.

This results in a tree like structure of gradually reducing boxes. While this looks convincing in rendered form, the process naturally results in blocks within other blocks. Archicad does not provide clash detection facilities, and this illustrates the fundamental weakness of GDL as a generative modelling system. There is no direct way of finding out where things are.

The algorithm:

The script starts by moving the coordinate system by half the edge-length and making a brick of side  $a$ . This manoeuvre is because the origin of the cube is tracked as though it were in the middle of the cube, whereas in reality it is the bottom left corner.

1. Move origin- In the beginning you move the coordinate system

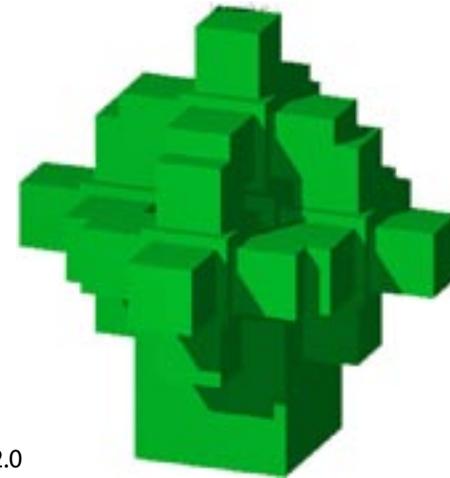


figure 2.0

origin in the centre of the cube. Moving the origin half the edge-length in X, Y & Z will place it here.

```
addx -a/2
addy -a/2
addz -a/2
```

2. Draw brick- The brick command makes the first, big brick

```
brick a,a,a
```

3. The del 3 overrides these last 3 moves, and the addz  $a/2$  moves the coordinate system to the top face of the new cube, at the same time the edglength is reduced by the reduction factor  $d$ .

```
del 3
addz a/2
let a=a*d
```

4. Stop condition- The conditional if statement checks if the edge-length ( $a$ ) is not less than the minimum length ( $e$ ). If it is, go to label 2 and ends the program

The width of this block is  $a*d$

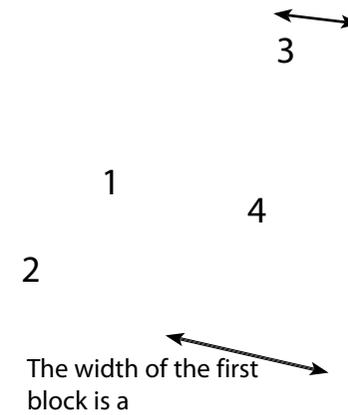


figure 2.1

if  $a < e$  then 2  
5. Recursive calls- Here starts the recursive bit: Move up half the new edgelength ( $addz\ a/2$ ) call this macro again (call "bricks"  $a,b,c,d,e$ ). This will create a smaller block on the top of the original one. Then rotate the coordinate system 90 degrees in  $y$  and call this macro again. This will place a block on the face of the original block. And so on...

```
addz a/2  
call "bricks" a,b,c,d,e  
Roty 90  
Call "bricks" a,b,c,d,e  
Rotx 90  
Call "bricks" a,b,c,d,e  
Rotx 90  
Call "bricks" a,b,c,d,e  
Rotx 90  
Call "bricks" a,b,c,d,e
```

1. Move origin in the centre of the cube.
2. Moving the origin half the edgelength in  $X\ Y\ \&\ Z$  will place it here.
3. Move up half the new edgelength call this macro again. This will create a smaller block on the top of the original one.
4. Rotate 90 degrees in  $y$  and call this macro again. Place a block on the face of the original block.

2.0	e	
Did you complete assignment 2.0.c? Did you need help?  What new topics did you learn?		Yes No Yes No
What old topics did you rehearse?		
Help provided		
e Tutors comments		

scripting

```
addx -a/2
addy -a/2
addz -a/2
brick a,a,a
del 3
addz a/2
let a=a*d
if a<e then 2
addz a/2
call "bricks" a,b,c,d,e
Roty 90
Call "bricks" a,b,c,d,e
Rotx 90
Call "bricks" a,b,c,d,e
Rotx 90
Call "bricks" a,b,c,d,e
Rotx 90
Call "bricks" a,b,c,d,e

2: end
```

g.d.l.

115

exercise

scripting

## 3.0 Recursive Tree

- 116 This program creates a three dimensional tree in a recursive fashion. The program has two parts: The main part, named elm tree, creates a trunk and calls 'winter' four times, one for each main branch. winter, the second part, draws the branches using recursion.

The algorithm of winter

1. Rotate the x axis between 10 and 35 degrees randomly

```
rotx (10+rnd(25))
```

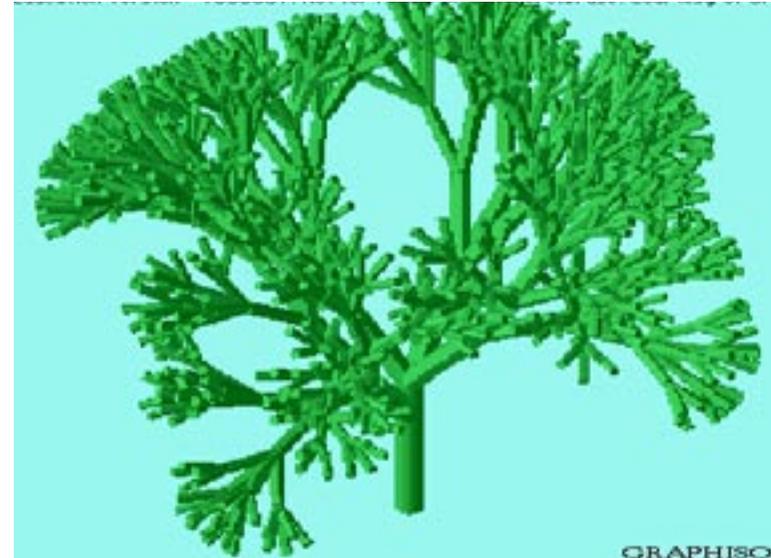
ie 10 plus a random number between 1 and 25

2. Make a cylinder and move to the end of the cylinder.

```
cylind c,d  
addz c
```

3. If the branch length is small, but not so small as to warrant stopping, jump to subroutine at label 4.

```
if d<0.14 and d>f gosub 4
```



4. Reduce the branch length & radius by 80%.

```
c=c*0.8  
d=d*0.8
```

5. If the branchlength is less than the limit then go to subroutine at label 2. This effectively stops the execution and returns to the main, calling part of the program.

```
if d<f then 2
```

6. Call second part, 'winter'- itself- recursively 'g' times, and each time rotate in z by 360/g.

```
for t=1 to g step 1  
  rotz 360/g  
  call "winter" a,b,c,d,e,f,g  
next t
```

The parametric dialog box shows the current values of each variable. By altering these values different forms of branching structure can be generated.

Parameters:

↔ C	length of branch	= 2.00
↔ D	radius of branch	= 0.20
↔ E		= 0.50
↔ F	smallest branch	= 0.80
↔ G	no of branches per recursion	= 3.00
↔ H		= 0.60
↔ I		= 0.00

```
!elm tree
!main part
  cylind 3,0.2
  addz 2.2
  rotx 10
  call "winter" a,b,c,d,e,f,g
  del 1
  addz 0.8
  rotz 180
  rotx 10
  call "winter" a,b,c,d,e,f,g
  del 1
  rotz 90
  rotx 10
  call "winter" a,b,c,d,e,f,g
  del 1
  rotz 180
  rotx 10
  call "winter" a,b,c,d,e,f,g
END
```

scripting

```
!winter
  resol 6
  rotx (10+rnd(25))
  !shadow AUTO ON
  cylind c,d
  body 1
  addz c
  if d<0.14 and d>f gosub 4
  c=c*0.8
  d=d*0.8
  if d<f then 2

  for t=1 to g step 1
    rotz 360/g
    call "winter" a,b,c,d,e,f,g
  next t
END

2:
END

4:
  resol 4
  !shadow off,off
  let g=5
  rotx 5
  return
```

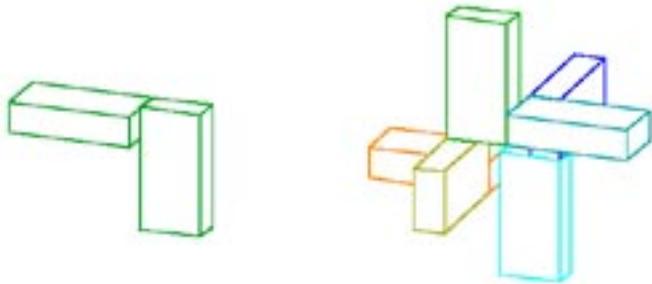
3.0	Add a further call to winter to provide a fifth branch going straight up to form a continuation of the trunk.
Did you complete assignment 2.0.c? Did you need help?  What new topics did you learn?	Yes No Yes No
	What old topics did you rehearse?
Help provided	
Tutors comments	

3.1	Devise further branching rules like the one in step 4 of winter	
e	Did you complete assignment 2.0.c?	Yes    No
	Did you need help?	Yes    No
What new topics did you learn?		
s		
What old topics did you rehearse?		
c		
What old topics did you rehearse?		
i		
What old topics did you rehearse?		
e		
Help provided		
e		
Tutors comments		

## 4.0 Froebel Exploratorium

### 120 GDL Froebel shape grammar

These routines explore the joining possibilities ('design grammars' in the literature) of one block. They depend on the definition of unique spatial relationships between symmetrical objects, as described in Terry Knight & Ulrich Flemming's papers (Knight 1990, Flemming 1986). If the 6 faces are uniquely labelled, then it is possible to calculate the total number of joining possibilities for a block, which in this case is  $8 \times 8 \times 8 (= 512)$ . This program allows you to explore each possibility by choosing settings for the three main controls, symmetry selectors 1 & 2 and rotation.



In this simple example one spatial relation is set up in subroutine 1000, which consists of:

```

add 4, 2, 1
rotz -90
roty 90
mul -1, 1, 1
add -a, 0, 0

```

These transformations are set up once to draw the 'exemplar' (on the left above) and then a stack of 8 blocks is drawn using a loop for s = 1 to 8

```

! draw first brick
brick 4, 2, 1

!move
gosub 1000

!draw second brick
brick 4, 2, 1

!move to draw stack
del top
addx 12
r = 5
!start main program
!DRAW STACK
for s = 1 to 8
!changes colours
r = (r+10)
if r <= 99 goto 100
r = int(r/d)
100:
pen r
brick 4, 2, 1
gosub 1000
next s

end
!end main program

```

```

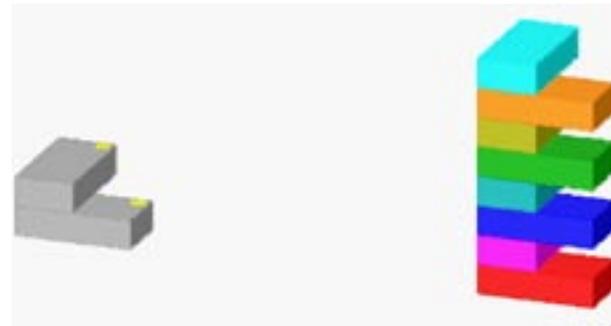
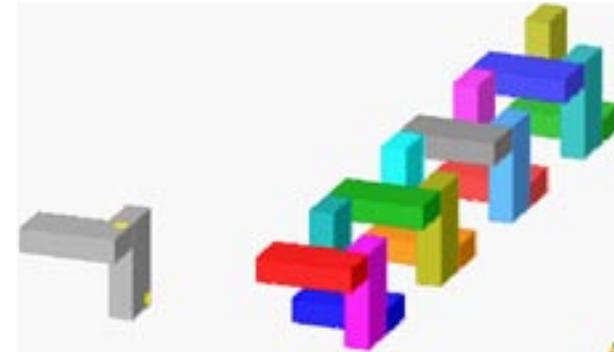
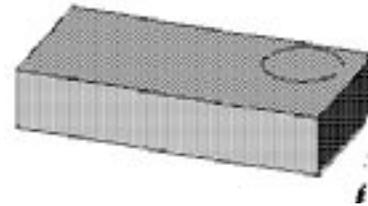
1000:! TRANSFORMATION
! move to corner
add 4, 2, 1
!rotate
!y/x
rotz -90
roty 90
!Symmetry change
mul -1, 1, 1
add -a, 0, 0
return

```

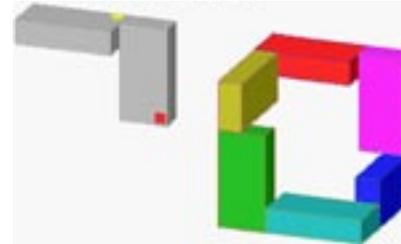
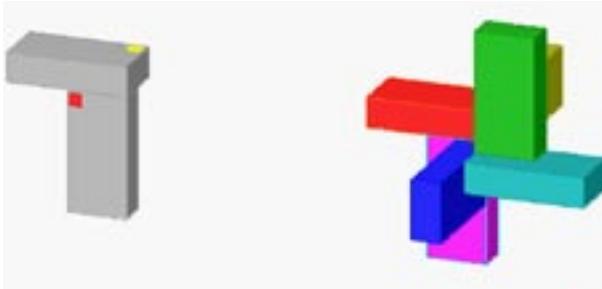
122

In this version the brick is drawn using the macro "frb1" twice as a fully labelled example.

The main transformation is carried out repeatedly, in randomly chosen colours.



```
pen 94  
  
block a, b, c  
add (a-0.35), (b-0.35), (c+0.02)  
pen 81  
circle 0.25  
del 1  
add (a-0.5), (b-0.5), (-0.02)  
pen 15  
rect 0.4, 0.4  
  
end
```



A full blown exploratorium for the two block single relationship grammar.

### The Algorithm

In the general case all possible transformations can be laid out as a series of subroutines. The actual subroutines are chosen with values for the variables F G & H, and exploit the ability of GDL to use a variable as the argument to a GOSUB

In this case , the labels for the three sets of subroutines are arranged in series 1..9,21..28,41..48. The values for F G & H are used with an offset of 0 20 & 40 to address the correct label.

Thus if  $g = 3$

$p = g+40$

$x = f$

$q = h+20$

then

GOSUB p

is equivalent to

GOSUB 43



The three transformations are arranged as three sets of 8 routines,providing 512 alternatives

```

!brick size see parameters
  p = g+40
  x = f
  q = h+20
!draw first brick
  call "frb1" a, b, c
!move
  gosub 1000
!draw second brick
  call "frb1" a, b, c
!move to draw stack
  del top
  addx e
  r = 5
!start main program
!DRAW STACK
  for s = 1 to d
!changes colours
  r = (r+10)
  if r <= 99 goto 100
  r = int(r/d)
100:   pen r
  brick a, b, c
  gosub 1000
  next s
end
!end main program

! TRANSFORMATION
1000:
! move to corner
  gosub x
!rotate
  gosub q
!Symmetry change
  gosub p
  return
! move to corner
  1:  add 0, 0, c
  mul -1, -1, 1
  return
  2:  add a, 0, 0
  mul 1, -1, -1
  return
  3:  add a, 0, c
  mul 1, -1, 1
  return
  4:  add a, b, 0
  mul 1, 1, -1
  return
  5:  add a, b, c
  return
  6:  add 0, b, 0
  mul -1, 1, -1
  return
  7:  add 0, b, c
  mul -1, 1, 1
  return
  8:  add 0, 0, 0
  mul -1, -1, -1
  return
  10:
  return
!rotate
!z/z
  21: rotz -90
  mul 1, -1, 1
  return
  22: !z/x
  rotz -90
  rotx 90
  return
  23: !y/z
  rotz -90
  rotx -90
  return
  24: !z/y
  rotz 90
  roty -90
  return
  25: !x/z
  rotz 90
  roty 90
  return
  26: !y/x
  rotz -90
  roty 90
  return
  27: !x/x
  roty 90
  mul 1, 1, -1
  return

```

```

  28: !y/y
  rotx 90
  mul 1, -1, 1
  return
  29: !x/y
  rotz 90
  rotx -90
  return

!Symmetry change
  41: return
  42: mul 1, -1, 1
  add 0, -b, 0
  return
  43: mul -1, 1, 1
  add -a, 0, 0
  return
  44: mul -1, -1, 1
  add -a, -b, 0
  return
  45: mul 1, -1, -1
  add 0, -b, -c
  return
  46: mul 1, 1, -1
  add 0, 0, -c
  return
  47: mul -1, -1, -1
  add -a, -b, -c
  return
  48: mul -1, 1, -1
  add -a, 0, -c

Return

```

will be able to understand it next year. It is good form to comment your code, simply so that you know what the various bits do. Using the curly brackets to "hide" successive chunks of the code (which have to be chosen carefully to leave a syntactically correct remnant) should lead you to isolate that part which causes the problem by a process of elimination.

ate arguments filled in with simple constants.

non-linear systems on the computer it is necessary to understand the basic structure of such systems and learn how to implement them in the computer .



cdr strips off the front atom of a list

lingo g.d.l. minipascal autolisp i n t r o

scripting quickstart syntax debugging exercise



come to look them up later.

GDL - Geometric Description Language- is ArchiCad's built in programming language. Without a knowledge of GDL, ArchiCad user's are limited to the range of objects that can be created using the ArchiCad toolbox. However using GDL allows access to a variety of shapes which would not otherwise available. GDL facilitates the creation of 3 dimensional parametric objects. Objects contain three parts: A two dimensional symbol, a three dimensional shape, and a description of the objects properties. The 2D symbol may be defined parametrically as well as the 3D shape.

### The Syntax

GDL offers a full range of facilities which allow the programmer to evaluate a condition and select alternative solutions. The syntax of GDL is fairly simple. A convenient object creation and debugging environment allows authors of GDL scripts to create new objects simply. GDL is a very simple language with only 26 variables the letters A to Z, and no variable typing. All scripts begin at the top of the text window (there is no explicit BEGIN ) and control can be altered by:

```
GOTO <label>
```

or

```
GOSUB <label>
```

where <label> is a number . The label is placed in front of the line to be jumped to followed by a colon

```
2:
```

The execution of the program ends when the file ends, or the END statement is reached.

GDL allows the use of conditional statements. Conditional expressions take the form:

```
IF <conditional expression> THEN <label>
```

for example:

```
IF a<e THEN 2
```

Loops are made with the FOR - NEXT statement:

```
FOR<index var>=<startvalue>TO<endvalue>[STEP<increment value>]
```

```
.....  
.....  
NEXT <index variable>
```

All object creation takes place at the origin of the three dimensional coordinate system. If you want to avoid placing things in the same place, you have to move the origin of the coordinate system. This can be done in three ways:

```
ADD,xn,yn,zn
```

which moves the origin n units in either one or all of the dimensions. ADDX, ADDY, ADDZ are all optional expressions if you only want to move in one direction. Thus

```
ADDX 10  
ADDY 5  
ADDZ 1.5
```

is equivalent to

```
ADD 10,5,1,5
```

```
ROT rx,ry,rz
```

which rotates the coordinate system about each axis (ROTZX ROTY and ROTZ are equivalents)

```
MUL mx,my,mz
```

which scales the coordinate axes. 1 leaves things as they are (MULX MULY MULZ again are equivalents). Using negative numbers allows reflection (-1 to reflect without scaling).

```
DEL n
```

The DEL n command is provided to "undo" any number of these transformations. DEL TOP removes all transformations. This is useful to get back to square one.

In the Froebel blocks example the transformations are used to define a spatial relation, which can be achieved by some combination of moves, rotations and reflection / mirroring.

26 parameters are provided which can be a limitation in creating