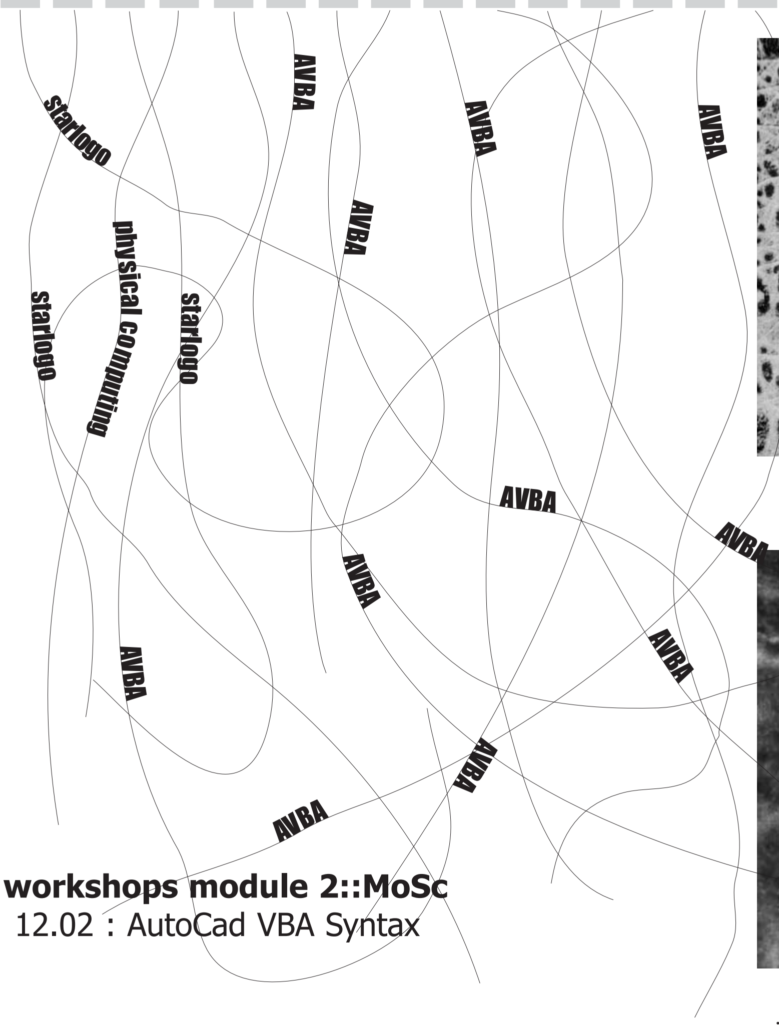
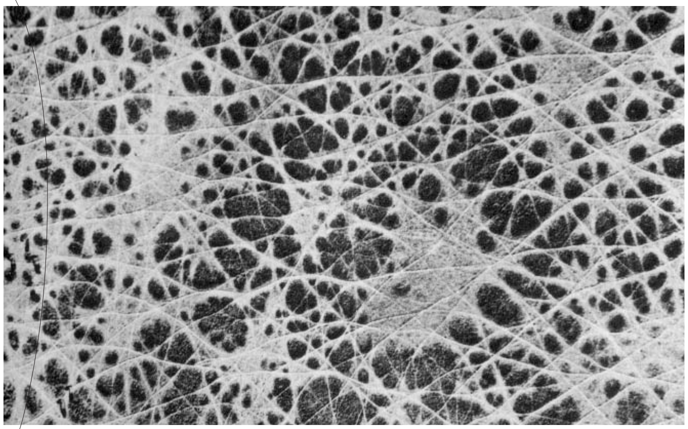
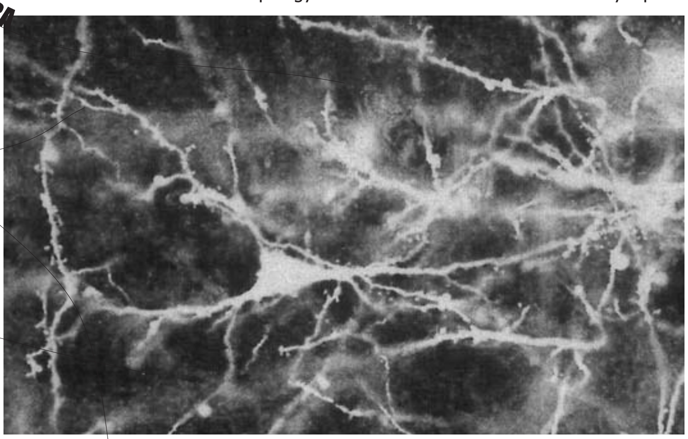



tracks of hoofed animals in the Mesopotamian Lowland

topology of the cortex :: neurons and their synapses



workshops module 2::MoSc
12.02 : AutoCad VBA Syntax

Modifying Coordinates of Objects

It has been so far almost impossible to modify coordinates of objects in AutoCad or Microstation, unless done in a crude way. VBA gives the designer the chance to manipulate every single vertex of an object independently and then update the object - superb!
The new program tests one of two ways of modifying vertices within the context of a polygonal mesh and shows you where you can't use coordinates to modify an object. Also, there are three more syntactical goodies (boolean variable type, select case statement and casting variables) and one conceptual method for systems' design (use-principle).

Coordinate(s)

The program generates a surface-mesh based on an array of points and has a ball, which moves about in a separate plane, determine through its x/y coordinates, how the mesh deforms - a dynamic surface. You will quickly see how to make a polygonal mesh in the main procedure dynasur():

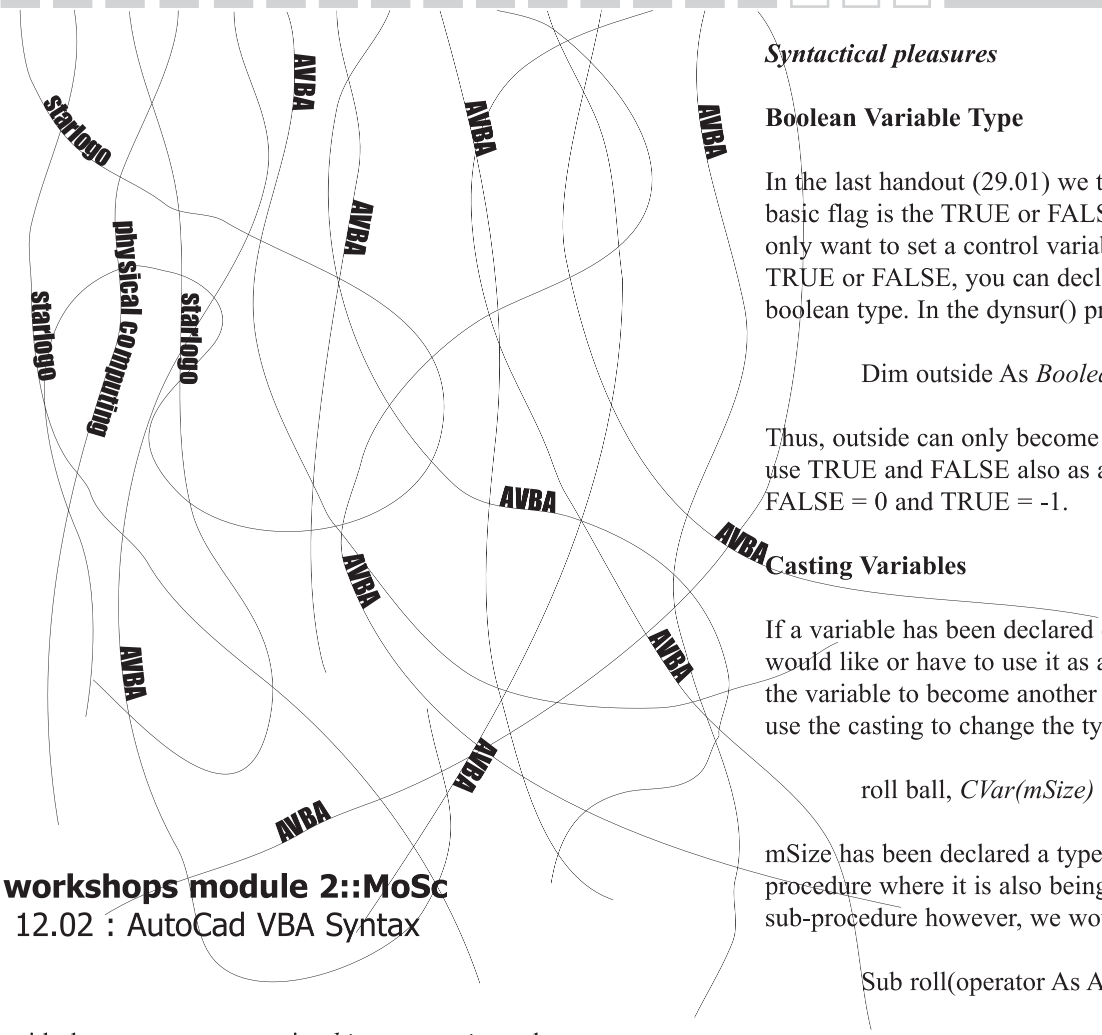
Set *meshObj* =
ThisDrawing.ModelSpace.*Add3DMesh(mSize, nSize, points)*

You have to tell the mesh how many rows and columns it will contain - mSize and nSize - and give the coordinates for those points (totalpoints = mSize * nSize; totalcoordinates = totalpoints * 3).
The longwinded nested loops to set up the points of the mesh, that put the points into a vertices() array are a good method to be able to control the structure of the mesh in a rigorous fashion if needed (just for the future). That way, we can work with the topology of the the mesh because no matter what the coordinates of the points - if they are neighbours or not - we can access the topological neighbours, like a matrix. For our program now we only need the points() array. But you also notice how the points() array doesn't differentiate coordinates that belong to the same point, whereas the vertices() array does.
Having stuffed the points() array into the Add3DMesh() method, you can see in the following line that you can determine what kind of mesh you want to work with:

meshObj.*Type* = *acBezierSurfaceMesh*
meshobj.*update*

The points in one row or column of your mesh will define a bezier curve now intstead of a polyline.This means that you will get control points that rule the curve. There are four different types of surface modes that you can chose (see help files). Don't forget to update the object after changing its type!

A surface as a static entity is half as helpful as one that you can change according to some contextual input. In order to change the surface at a local scale, we can access its coordinates - not its control points! There are two different ways to access coordinates: a) by accessing one point with its three components as a variant or b) by extracting all coordinates and then modify one or more. Afterwards, you can update the surface with the new coordinates. Accessing coordinates is allowed for all entities apart from lines, arcs and ellipses since they only really have two points to play



workshops module 2::MoSc
12.02 : AutoCad VBA Syntax

with that you can access via *object.startpoint* and *object.endpoint*:

```
        line.StartPoint = pta
        line.EndPoint = ptb
        line.Update
```

Since the two points are specific properties of the object, notice that you don't have to move them like an *object.centroid*, but simply do *object.update* afterwards.

Extracting all coordinates at once from an object via *object.coordinates* can be helpful but is less transparent and therefore more difficult to use. That's why we use the *object.coordinate(index)* method. The index indicates the vertex number with all three components -x,y,z. Thus, you can change the components separatly and replace the old vertex. In the new program we do that in check():

```
        verti = slave.Coordinate(i)
        ...
        slave.Coordinate(i) = verti
        slave.Update
```

We extract the vertex coordinates of the i-th index of the all the points of the surface and place them temporarily into the variant verti. Having modified the components of verti, we replace the i-th points with verti again and update the surface.

This method is valid also for solids. Thus, one could change the coordinates of each of the 8 cube vertices separatly.

Syntactical pleasures

Boolean Variable Type

In the last handout (29.01) we talked about flags. The most basic flag is the TRUE or FALSE boolean expression. If you only want to set a control variable to indicate if something is TRUE or FALSE, you can declare the variable to be of boolean type. In the dynsur() procedure we declare:

```
        Dim outside As Boolean
```

Thus, outside can only become TRUE or FALSE. You can use TRUE and FALSE also as a numerical operator, where FALSE = 0 and TRUE = -1.

Casting Variables

If a variable has been declared of a certain type and you would like or have to use it as a different type, you can *cast* the variable to become another type. In the new program we use the casting to change the type of the variable *mSize*:

```
        roll ball, CVar(mSize)
```

mSize has been declared a type integer in the dynsur() procedure where it is also being used as such. In the roll() sub-procedure however, we would like to use it as a double:

```
        Sub roll(operator As Acad3DSolid, limit As Double)
```

We need the double because a coordinate can also have decimals whereas the integer doesn't. In order to compare
        there(i) > *limit*
where there(i) is a coordinate as a double, we need to convert mSize to a double before sending it to roll(). CVar() means literally: **C**onvert the expression in parentheses **()** to **Var**iant/Double.
You can convert any variable type to any other. If you convert to a smaller type though, you will *truncate* information! For example, if a is a double of value 3.5 and you convert it to an integer by casting *CInt(*a*)*, the resulting integer a will truncate the decimals and therefore read 3.

Two dimensional arrays

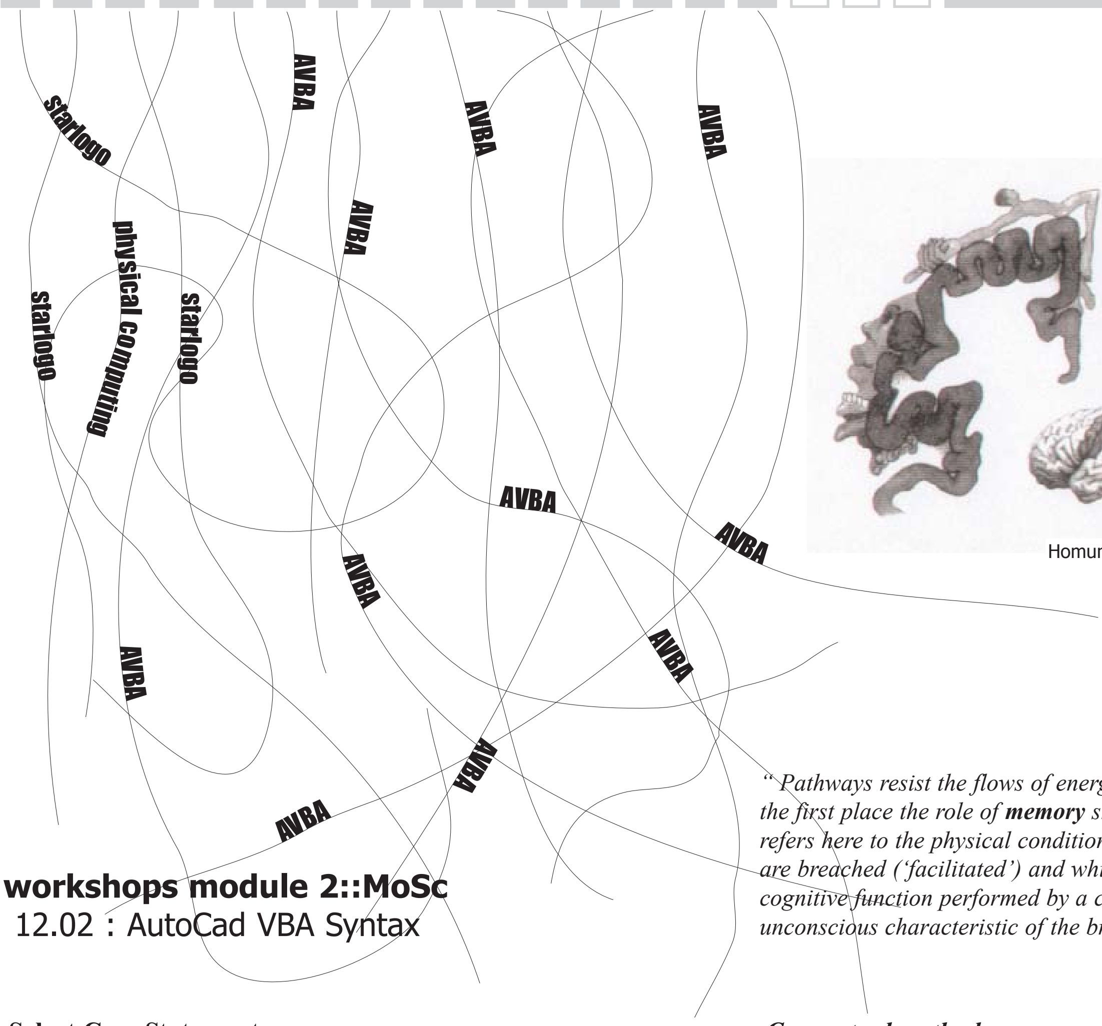
So far we have been using one dimensional arrays - a chest with one column of drawers. Imagine you had a chest with several columns of drawers, where each column has several rows. That's what we have in the declaration:

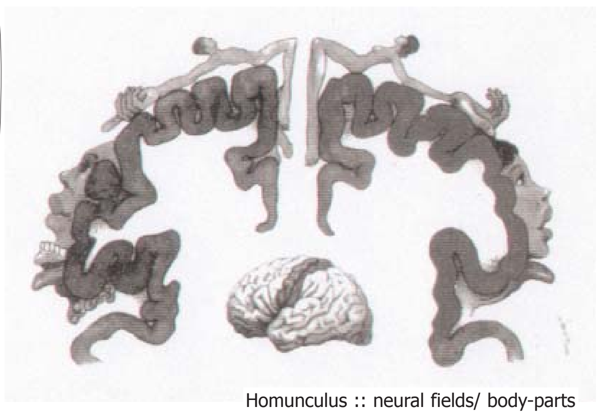
```
        ReDim vertices(mSize, nSize) As vertex
```

mSize are the number of columns and nSize the number of rows. That's why we loop through this array with the nested loop:

```
        For m = 0 To mSize - 1
                For n = 0 To nSize - 1
                        vertices(m, n).coo(0) = m
                        ...
```

It's like a matrix where each address consists of a 'm' and a 'n' component.




Homunculus :: neural fields/ body-parts

workshops module 2::MoSc

12.02 : AutoCad VBA Syntax

“ Pathways resist the flows of energy, unless it is used often. (...) In the first place the role of ***memory*** *should be underscored. ‘Memory’ refers here to the physical condition of the brain: which pathways are breached (‘facilitated’) and which are not. Memory is not a cognitive function performed by a conscious subject, but an unconscious characteristic of the brain.” Paul Cilliers*

Select Case Statement

The Select Case statement does the same job as the *ElseIf* in an If statement. But in a Select Case statement one only compares one expression to a variety of values. In the sub-procedure shoot() we use it to evaluate the variable *war*:

```
Select Case war

    Case 0
        this
    Case 1
        that
End Select
```

We could also write that statement as following:

```
If war = 0 then
    this
ElseIf war = 1 then
    that
End If
```

With the Select Case statement though, you can more elegantly evaluate a variable when the variable has a large range of values, say if you are going through a counter variable or array indices.

Task

Anti-war march, Embankment - Hide Park, Saturday 15. Feb

Conceptual method

The dynamic surface mesh in the mesh-walk program reflects through its shape the path the ball has been travelling. The feedback the ball has on the surface increases the z-coordinates of mesh points whose x/y coordinates are within a specified radius in 2D to the ball’s x/y coordinates. All mesh points that are not within that radius have their z coordinate value reduced.

```
If (dis <= rad) Then
    verti(2) = verti(2) + 0.6
Else
    verti(2) = verti(2) - 0.05
End If
```

The more often a point is close to the ball (or rolled over) the more salient the point and its surrounding becomes - we can see that peaks of the map are more often used by the ball than the troughs. When the Viennese psycho-analyst Sigmund Freud tried to understand how memory works, he believed that the more often a human being perceives or does something, the more clearly it remembers. He was right; neuroscientist later found that synapses connections between neurons in the brain become stronger and afford ‘memory’ where they are being used more often. Others are slowly weakened again because they get used for other ‘meanings’. Freud called his idea the Use-principle, also Hebb’s rule.
If one records traces of people walking in a defined square while it is snowing, one will find that the most used paths will remain more trampeled whereas the ones not used very often get snowed in again - bit like territorial memory.