What is VectorScript ?

• VectorScript is the scripting language component of the VectorWorks software package.

• It is a lightweight programming language which syntactically resembles **Pascal**.

• **VectorScript** is actually a “superset” of the Pascal language, extending basic Pascal capabilities with a number of APIs (application programming interfaces) which **provide access to** the features and funtionality of **the VectorWorks CAD engine.**

Some Background On VectorScript

• VectorScript originated in 1988 as **MiniPascal** in the Mini-CAD+ 1.0 release.

• With the advent of VectorWorks in 1998, MiniPascal became VectorScript.

• The core VectorScript language continues to be developed by **Nemetschek** North America.

Some Background On VectorScript

What VectorScript Can Do

• VectorScript is a relatively general purpose programming language, it provides the ability to perform **most common programming tasks**. Tasks such as
  • computations
  • storing a value, and
  • manipulating data

• VectorScript also provides **extended capabilities** specific to the **VectorWorks product**.

• **Object Creation and Editing**
  • create and edit objects directly
  • primitive objects (lines, rectangles, ...)
  • more complex objects (multiple 3D extrudes, 3D solids

• **Document Control**

• **Extended Data**
  • access to and control over - worksheets
    - data records
    - textures

Some Background On VectorScript

What VectorScript Can't Do

• VectorScript does not have the ability to work **across multiple documents** or **outside of a VectorWorks** document context.

• For reasons of simplicity and stability, VectorScript does not have the ability **to manage or control memory allocation**.

• VectorScript does not support **system level calls** for file-related or other tasks.

• VectorScript does not provide **external database** or other connectivity options.

How does VectorScripts look?

```
PROCEDURE FirstExample;
CONST
        kGREETING = 'Hello ';
VAR
        myMessage : STRING;


BEGIN
        myMessage:='VectorScript';

        Message(kGREETING,myMessage);
        Wait(5);
        SysBeep;
        ClrMessage;
END;
Run(FirstExample);
```

How does VectorScripts look?

.. in the VectorWorks VectorScript Editor.

```
PROCEDURE FirstExample;
CONST
        kGREETING = ‘Hello ‘;
VAR
        myMessage : STRING;


BEGIN
        myMessage:=’VectorScript’;

        Message(kGREETING,myMessage);
        Wait(5);
        SysBeep;
        ClrMessage;
END;
Run(FirstExample);
```

How does VectorScripts look?

.. in a texteditor with Syntaxhighlightning

```
PROCEDURE FirstExample;
CONST
        kGREETING = 'Hello ';
VAR
        myMessage : STRING;


BEGIN
        myMessage:='VectorScript';

        Message(kGREETING,myMessage);
        Wait(5);
        SysBeep;
        ClrMessage;
END;
Run(FirstExample);
```

How does VectorScripts look?

the different parts of a VectorScript

```
PROCEDURE FirstExample;
CONST
        kGREETING = ‘Hello ‘;
VAR
        myMessage : STRING;


BEGIN
        myMessage:=’VectorScript’;

        Message(kGREETING,myMessage);
        Wait(5);
        SysBeep;
        ClrMessage;
END;
Run(FirstExample);
```

Identifies the script to the VectorScript compiler

How does VectorScripts look?

the different parts of a VectorScript

```
PROCEDURE FirstExample;

CONST
        kGREETING = ‘Hello ‘;
VAR
        myMessage : STRING;

BEGIN
        myMessage:=’VectorScript’;

        Message(kGREETING,myMessage);
        Wait(5);
        SysBeep;
        ClrMessage;
END;
Run(FirstExample);
```

Declares data storage for the script

The source code of the script

How does VectorScripts look?

the different parts of a VectorScript

```
PROCEDURE FirstExample;

CONST
        kGREETING = ‘Hello ‘;
VAR
        myMessage : STRING;


BEGIN
        myMessage:=’VectorScript’;

        Message(kGREETING,myMessage);
        Wait(5);
        SysBeep;
        ClrMessage;
END;
Run(FirstExample);
```

Tells the VectorScript compiler to run the script

Is VectorScript easy ?

yes!

..always ?

no.

Can I get sick of VectorScript ?

no.

..really ?

ok, .. it depends.

How to learn VectorScript?

- **write programms** with VectorScript
- **make mistakes**
- **make mistakes**
- **make mistakes**
- **make mistakes**

The Grammatik of VectorScript

Case Sensitivity

• VectorScript **is not case sensitive**. This means that items such as language keywords, variables, function names, and any other identifiers can be specified using uppercase, lowercase, or a mixed case and still be compatible with other variations of the same item.

- APFEL = apfel = Apfel

The Grammatik of VectorScript

speaking Variables

- use **speaking Variables** in your Scripts
  - it makes live more easy
  - it makes your Script more **readable**
  - you and your Colleagues will understand your script faster

  - e.g. rectLength, rectLeftCornerXpos

The Grammatik of VectorScript

space

• Since spaces, tabs, and new lines do not have meaning to the VectorScript compiler, you are free to use them to indent and format your script code. This type of formatting makes your scripts **easy to read** and **understand**.

```
PROCEDURE FirstExample;
CONST
        kGREETING = 'Hello';
VAR
        myMessage : STRING;

BEGIN
        myMessage:='VectorScript';

        Message(kGREETING,myMessage);
        Wait(5);
        SysBeep;
        ClrMessage;
END;
Run(FirstExample);
```

The Grammatik of VectorScript

space

```
PROCEDURE FirstExample;
CONST
kGREETING = ‘Hello’;
VAR
myMessage : STRING;
BEGIN
myMessage:=’VectorScript’;
Message(kGREETING,myMessage);
Wait(5);
SysBeep;
ClrMessage;
END;
Run(FirstExample);
```

The Grammatik of VectorScript

space

PROCEDURE FirstExample;CONST kGREETING = ‘Hello’;VAR myMessage : STRING;BEGIN myMessage:=’VectorScript’;
Message(kGREETING,myMessage);Wait(5);SysBeep;ClrMessage;END;Run(FirstExample);

The Grammatik of VectorScript

Comments

• Comments in VectorScript are used **to place descriptive text** within script code. They are most often used to **document script code** for your reference and for others who may work on your scripts. The Vector Script compiler ignores comments.

• The general syntax for a single VectorScript comment is:

```
{This is a comment}
```

• To comment out a block of the VectorScript code the syntax is:

```
(* my comment:
write what ever you wanted,
even VectorScriptcode or {other comments!}
 *)
```

Identifiers

• Identifiers in VectorScript are symbols which are used to refer to something else: **constants**, **variables**, **data types**, **procedure** or **function names**, and other similar items. The rules for writing VectorScript identifiers are :

- The first character must be a letter or an underscore.
- Subsequent characters may be a character, digit, or underscore.
- Identifiers may not contain spaces, tabs, or other characters.

Value Identifiers

num	color_32bit	totalLumberUsed
SUM	_dummy	A_very_fine_identifier

Invalid Identifiers

52pickup	three+two	SUB TOTAL

Reserved Words

ALLOCATE	AND	ARRAY	BEGIN
BOOLEAN	CASE	CHAR	CONST
DIV	DO	DOWNTO	DYNARRAY
ELSE	END	FALSE	FOR
FUNCTION	GOTO	HANDLE	IF
INTEGER	LABEL	LONGINT	MOD
NIL	NOT	OF	OR
OTHERWISE	PI	PROCEDURE	REAL
REPEAT	STRING	STRUCTURE	THEN
TO	TRUE	TYPE	UNTIL
USES	VAR	VECTOR	WHILE

FILE	*FORWARD*	*IMPLEMENTATION*	*INHERITED*
INTERFACE	*INTRINSIC*	*OBJECT*	*OVERRIDE*
PACKED	*PROGRAM*	*SET*	*UNIT*
USES	*WITH*		

Variables, and Constants

Variables

• The **VAR block** in the VectorScript is the only location where variables can be declared;

• The purpose of the **VAR block** is to define storage requirements, not to define data.

```
VAR
        myMessage : STRING;
```

• The general syntax for a variable declaration is:

<identifier>(,<identifier>,...) : <data type>;

```
jobName:STRING;
i,j,k:INTEGER;
```

Variables, and Constants

Constants

• The **CONST block** in the VectorScript is the only location where constants can be declared;

• Constants, unlike variables, do not require an explicit data type..

```
CONST
        kGREETING : ´Hello´;
```

• The general syntax for a variable declaration is:

<identifier> = <value>;

LOCAL_GREETING_FRENCH = ´Bonjour ´;

Data Types

Fundamental Data Types

- Numeric
- Text
- Other

Data Types

Numeric

- **INTEGER**
  -32767 to 32767
- **LONGINT**
  -2.147.183.647 to 2.147.183.647
- **REAL**
  1.9 x 10e-4951 to 1.1 x 10e4932

Data Types

Text

- **STRING**
  - up to 255 characters
  - ASCII character
- **CHAR**
  - a single ASCII character

Data Types

..Other

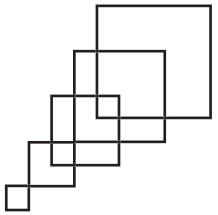
- **BOOLEAN**
  - TRUE or FALSE
- **HANDLE**
  - to store a **reference** to other VectorWorks data in memory.
- **VECTOR**
  - A VectorScript VECTOR consists of three **component** REAL values which can also be treated as a single unit value.
- **POINT**
  - to store the coordinates of a 2D point. It is a compound data type consisting of two **component** REAL values: **x** and **y**.
- **POINT3D**
  - to store the coordinates of a point in 3D space. It is a compound data type consisting of three **component** REAL values: **x**, **y** and **z**.
- **RGBColor**
  - The RGBCOLOR data type can store a color as three components: **red**, **green**,and **blue**. Each component is a LONGINT value.
- **NIL**

Repetition Statements

The FOR Statement

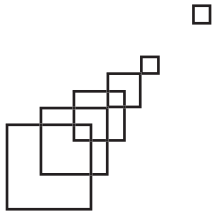
- **FOR ... TO... DO**

```
PROCEDURE LOOP;
VAR
        i : INTEGER;
BEGIN
        FOR i:=1 TO 5 DO rect(i,i,i*2,i*2);
END;
Run(LOOP);
```



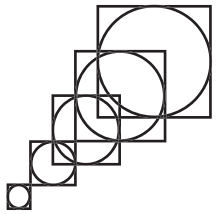
- **FOR ... DOWNTO... DO**

```
BEGIN
        FOR i:=1 DOWNTO -5 DO rect(i,i,i*2,i*2);
END;
```



- **FOR ... TO... DO BEGIN**

```
BEGIN
        FOR i:=1 TO 5 DO  BEGIN
                rect(i,i,i*2,i*2);
                oval(i,i,i*2,i*2);
                SysBeep;
        END;
END;
```



Repetition Statements

The WHILE Statement

- **WHILE ... DO**

```
PROCEDURE WhileLoop;
VAR
        h : HANDLE;
BEGIN
        h:= FActLayer;
        WHILE (h <> NIL) DO BEGIN
                SetSelect(h);
                h:=NextObj(h);
        END;
END;
Run(WhileLoop);
```

Repetition Statements

The REPEAT Statement

• **REPEAT ... UNTIL (...)**

```
PROCEDURE RepeatLoop;
VAR
        h : HANDLE;
BEGIN
        h:= FActLayer;
        REPEAT
                SetSelect(h);
                h:=NextObj(h);
        UNTIL (h=NIL);
END;
Run(RepeatLoop);
```

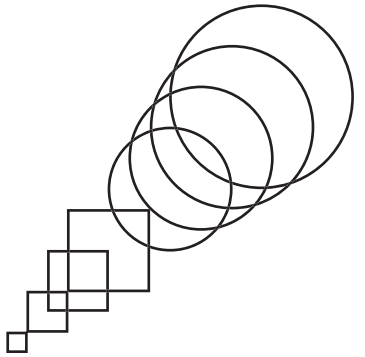
• Unlike the **WHILE** statement, however, the **REPEAT** statement evaluates the control expression after executing its controlled statement. This means that the controlled statement will **always execute at least once.**

Conditional Statements

The IF Statement

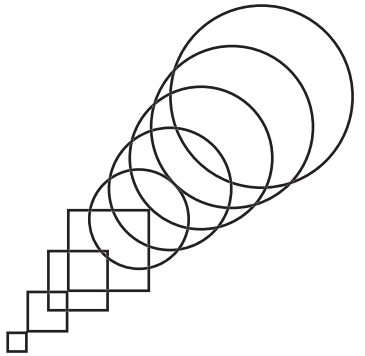
- **IF ... THEN ...**

```
PROCEDURE theIf;
VAR
        i : INTEGER;
BEGIN
        FOR i:=0 TO 9 DO BEGIN
                IF (i<5) THEN Rect(i,i,i*2,i*2);
                IF (i>5) THEN Oval(i,i,i*2,i*2);
        END;
END;
RUN(theIf);
```



- **IF ... THEN ... ELSE..**

```
PROCEDURE theIf;
VAR
        i : INTEGER;
BEGIN
        FOR i:=0 TO 9 DO BEGIN
                IF (i<5) THEN Rect(i,i,i*2,i*2)
                ELSE Oval(i,i,i*2,i*2);
        END;
END;
RUN(theIf);
```

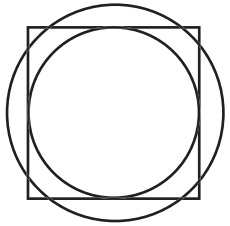


Conditional Statements

The CASE Statement

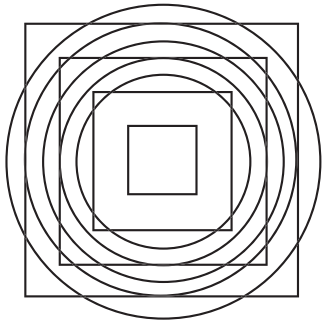
- **CASE ... OF ... END;**

```
PROCEDURE theIf;
VAR
        i : INTEGER;
BEGIN
        FOR i:=0 TO 9 DO BEGIN
                CASE i OF
                        2: Rect(-i,-i,i,i);
                        4,5: Oval(-i/2,-i/2,i/2,i/2);
                END;
        END;
END;
RUN(theIf);
```



- **CASE ... OF ... [OTHERWISE] ... END;**

```
PROCEDURE theIf;
VAR
        i : INTEGER;
BEGIN
        FOR i:=0 TO 9 DO BEGIN
                CASE i OF
                        0..4: Rect(-i,-i,i,i);
                        OTHERWISE Oval(-i/2,-i/2,i/2,i/2);
                END;
        END;
END;
RUN(theIf);
```



Arrays in VectorScript

• An **array** in VectorScript is a collection of data values referenced by a **single identifier**. Arrays allow **large amounts of data** to be **stored and manipulated** during script execution.

• VectorScript arrays are **indexed.**

• VectorScript provides support for two types of arrays: **static arrays** (ARRAY), and **dynamic arrays** (DYNARRAY).

- Static Array
- Dynamic Array

Arrays in VectorScript

Static Array

• Static arrays (ARRAY) are declared using the **same method as** used for **variables**

• Static arrays come in **one-** and **two-dimensional** varieties.
The general syntax for one-dimensional static arrays is:

```
<identifier> : ARRAY [ m..n ] OF <data type>;
e.g. myArray : Array[0..23] OF INTEGER;
```

• To retrieve a value from an element of a one-dimensional array, the bracket notation has to be used, e.g.

```
j := values[3];
values[23] := 15.5;
total := price[i] + tax;
```

Arrays in VectorScript

Static Array

- example Script:

```
PROCEDURE ExampleArray;
VAR
        s:STRING;
        i:INTEGER;
        words:ARRAY[1..10] OF STRING;
BEGIN
        words[1]:=’VectorScript ‘;
        words[2]:=’is ‘;
        words[3]:=’a ‘;
        words[4]:=’fine ‘;
        words[5]:=’language.’;

        FOR i:=1 TO 5 DO s:=Concat(s,words[i]);
                Message(s);
        END;
END;
Run(ExampleArray);
```

Arrays in VectorScript

Static Array

• Two-dimensional static arrays **extend the syntax of a one-dimensional** array by adding an additional array index to the declaration:

<identifier> : ARRAY [m..n,r..s] OF <data type>;

• In the declaration for the two-dimensional array, **the first index** value defines the **number of "rows"** in the array, while **the second index** defines the **number of "columns."**

• Accessing an element in a two-dimensional array is not very different from a one-dimensional array:

```
j := values[3,5];
values[23,1] := 15.5;
total := price[i,j] + tax;
```

• If we think of the two-dimensional array in terms of **rows** and **columns**, we would use **two index values to indicate the row and column position** of the array element to be indexed.

Arrays in VectorScript

Dynamic Array

• Dynamic arrays (DYNARRAY) in VectorScript are **similar to static arrays**, with the notable exception of how they are dimensioned, or sized.

• While static arrays are explicitly sized when they are declared in the VAR block of your script, **the size of a dynamic array is declared during the actual execution of a script.**

• Dynamic arrays **can also be resized** at any point **during script execution** to suit your data storage requirements.

• Dynamic arrays can also be specified as **one- or two-dimensional**. The general syntax for dynamic arrays are:

• one-dimensional:
<identifier> : DYNARRAY [] OF <data type>;

• two-dimensional:
<identifier> : DYNARRAY [,] OF <data type>;

Arrays in VectorScript

Dynamic Array

• **To dimension** a dynamic array, VectorScript uses the **ALLOCATE** keyword (along with a reference to the array):

```
ALLOCATE int_values[1..5];
```

• Extended String Support with CHAR Arrays

  • VectorScript also supports a specialized set of functionality when using arrays of the CHAR data type.

  • Arrays of type CHAR can be used in place of the STRING data type in certain operations within VectorScript.

• for more Details to manipulating STRINGS and CHAR data type you can check the manual

Arrays in VectorScript

Dynamic Array

- example Script:

```
PROCEDURE Example_DynArray;
VAR
        i,j,numtxt : INTEGER;
        h : HANDLE;
        textStore: DYNARRAY[] OF STRING;
BEGIN
        numtxt:=Count(((T=Text) & (SEL=TRUE)));
        j:=1;
        ALLOCATE textStore[1..numtxt];
        h:=FSActLayer;
        WHILE (h <> NIL) DO BEGIN
                IF (GetType(h) = 10) THEN BEGIN
                        textStore[j]:=GetText(h);
                        j:=j+1;
                END;
                h:=NextSObj(h);
        END;
        ALLOCATE textStore[1..numtxt+2];
        TextOrigin(2,2);
        CreateText('New text 1');
        numtxt:=numtxt+1;
        textStore[numtxt]:=GetText(LNewObj);
        TextOrigin(2,4);
        CreateText('New text 2');
        numtxt:=numtxt+1;
        textStore[numtxt]:=GetText(LNewObj);
        FOR i:=1 TO numtxt DO BEGIN
                Message('Array element ',i,' contains ', textStore[i]);
                Wait(1);
        END;
END;
Run(Example_DynArray);
```

Structures

• A structure in VectorScript is a **collection of one or more variables** which are grouped together under a single identifier for convenient handling.

• Structures help to **organize complex data into groupings** that may be treated as a single "unit" instead of separate entities.

• The general syntax for a structure declaration is:

<structure name> = STRUCTURE
<identifier>[,<identifier>,…] : <data type>;
<identifier>[,<identifier>,…] : <data type>;

• Members within a structure may be referred to directly using the **. (structure member) operator**.

<structure name>.<member name>

• This format, also known as "**dot notation**," gives you direct access to the value within the specified member.

Structures

- Example:

```
PROCEDURE Example_structure;

TYPE
        HANSPETER = STRUCTURE
                vorname, nachname : STRING;
        END;

        ADRESSE = STRUCTURE
                strasse : STRING;
                hausnummer: INTEGER;
                stadt : STRING;
        END;

VAR
        person1: HANSPETER;
BEGIN
        person1.vorname:= 'Uschi';
        person1.nachname:='Biedermann';

        Message(person1.vorname, ' ' ,person1.nachname);
END;
Run(Example_structur);
```

User Defined Functions

- User-Defined **Procedures.**
- User-Defined **Functions**.

User Defined Functions

User-Defined Procedures

• with user defined functions, you can **break large script** tasks **into smaller ones**.

• Another term for user-defined functions is **subroutines** which, as the name implies, are pieces of script code which perform tasks within the main script.

• User-defined procedure subroutines are **the most common type of subroutine**.

• User-defined procedures are **declared after** the definition (**CONST, TYPE**, and **VAR**) blocks of a script, but before the script body.

• Just like a script, subroutines may have any of the **standard VectorScript definition blocks** (LABEL, CONST, TYPE, or VAR) as well as a script body.

• The general syntax for user-defined procedures is:

PROCEDURE <procedure identifier>[(<parameter list>)]
e.g. PROCEDURE SumOfSquares(limit:INTEGER;VAR result:INTEGER);

User Defined Functions

User-Defined Procedures

• Following the subroutine identifier is the **parameter list** for the subroutine. This optional list defines a method of **moving data in and out** of the subroutine.

• While it is possible to refer to values in the enclosing program blocks directly, doing so **would eliminate the ability** to easily use the subroutine in other code, which is one of the **major advantages** of using subroutines.

• The parameter list declares **a set of identifiers** (and their associated data types) that will be used **to pass data to and from the subroutine**.

• The **VAR keyword** indicates an identifier that will be **used to pass data** out of the subroutine to the calling code.

• **Identifiers** in the parameter list **can be treated as variables** and used within the subroutine script code.

• By calling the subroutine, **the order and types of the variable identifiers must exactly match those in the declaration.**

User Defined Functions

User-Defined Procedures

- example:

```
Procedure testSubroutine;

VAR
        myPosX,myPosY: REAL;
        myRadius, myDiameter            :INTEGER;

{SUBROUTINES}
Procedure CircelByPointAndRadius(rad, pointX, pointY : REAL; VAR diameter:INTEGER);
BEGIN
        diameter:=2*rad;
        oval(pointX-rad, pointY-rad, pointX+rad, pointY+rad);
END;
{End of declaration subroutine}

BEGIN;
        myPosX:= 23.5;
        myPosY:=myPosX;
        myRadius:= 50;
        CircelByPointAndRadius(myRadius, myPosX, myPosY, myDiameter);
END;
Run(testSubroutine);
```

User Defined Functions

User-Defined Functions

• User-defined functions incorporate **all the features of user-defined procedures**, but they have one additional feature which makes them extremely useful when writing scripts: **an associated value**.

• User-defined functions, unlike procedures, can **pass data out** of the subroutine through a return value, which associates the value with the subroutine identifier.

• User-defined function declarations have one additional requirement: **a return value type** after the parameter list. This data type indicates what type of data will be passed through the return value mechanism and will be associated with the identifier.

• The general syntax for user-defined functions is:

FUNCTION <procedure identifier>[(<parameter list>)]:<return value type>

User Defined Functions

User-Defined Functions

- example:

```
PROCEDURE SubrExample2;
VAR
        n,sum:INTEGER;

FUNCTION SumOfSquares(limit:INTEGER):INTEGER;
        BEGIN
                SumOfSquares:= limit*(limit+1)*(2*limit+1)/6;
        END;

BEGIN
        n:=IntDialog('Enter the limit value','0');
        {sum of squares for the first n integers}
        sum:= SumOfSquares(n);
        Message('The sum of squares is: ',sum);
END;
Run(SubrExample2);
```