ArtRage Script Programmers Guide.

Introduction:

The ArtRage scripting engine incorporates a rich C-like interpreted language. The language allows local and global variable definitions, function definitions that can take parameters and return values, as well as flow-control statements such as 'for/next' loops, 'while' loops, 'if/then/else'. It can be used to create complex behaviours for ArtRage tools, or entire parametric painted scenes.

This is not a comprehensive guide to programming. The reader is assumed to have a basic knowledge of programming, using variables, declaring and calling functions.

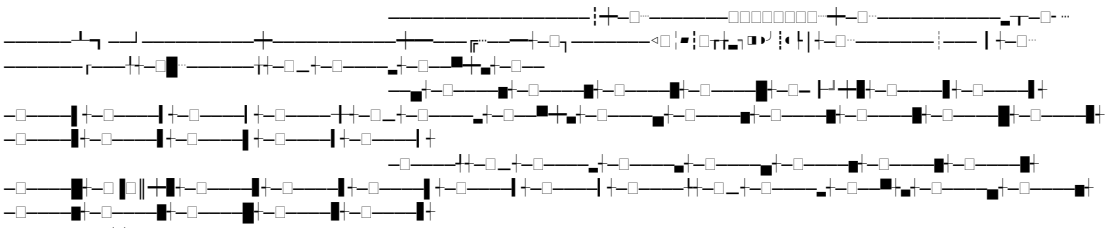
The script file

Script files are simple Unicode text files. They can be edited with Notepad or TextEdit. They are 'Human readable'. The files are read by the script parser line-by-line.

'Language' vs 'Recorded'

When a script is recorded a Unicode(UTF16) text file is produced with XML-style block formatting. The file is 'human readable', with blocks of information surrounded with <block> </block> statements to help the script parser identify the contents of the blocks. For example, if you record a simple script in ArtRage then open the resulting script file with Notepad or TextEdit you will see something similar to this:

```
//==============================================================================
//==============================================================================
//                              ArtRage Script File.
//==============================================================================
//==============================================================================


//==============================================================================
// Version Block - Script version and ArtRage version:
//==============================================================================

<Version>
    ArtRage Version: ArtRage 3 Studio Pro
    ArtRage Build: 3.0.8
    Professional Edition: Yes
    Script Version: 1
</Version>


//==============================================================================
// Header block - Info about the painting/person who generated this script:
//==============================================================================

<Header>
    // === Project data
    Painting Name: "Untitled"
    Painting Width: 1280
    Painting Height: 1024
    Painting DPI: 72
    // === Author data
    Author Name: "Andy"
    Script Name: "Test"
    Comment: ""
    Script Type: ""
    Script Feature Flags: 0x000000005
</Header>

//==============================================================================
// ArtRage project features. Sets the startup state of the script:
//==============================================================================

<StartupFeatures>
    Script Startup Features: {
```



```
    }    // End of Script startup feature binary data.
</StartupFeatures>

//============================================================================
// Script data follows:
//============================================================================

<Events>
<StrokeEvent>
        <StrokeHeader>
                <EventPt>    Wait: 0.000s Loc: (665, 336)  Pr: 1    Ti: 1    Ro: 0    Rv: NO   Iv:
NO      </EventPt>
                <Recorded>       Yes      </Recorded>
                <Smooth> Count:   3
                        Loc: (665, 333)  Pr: 0    Ti: 1    Ro: 0
                        Loc: (665, 334)  Pr: 0    Ti: 1    Ro: 0
                        Loc: (665, 335)  Pr: 0    Ti: 1    Ro: 0
                </Smooth>
                <PrevA>  Loc: (-158.671, 253.411)  Pr: 0.243437     Ti: 1    Ro: 0    </PrevA>
                <PrevB>  Loc: (-142.082, 236.822)  Pr: 0.391875     Ti: 1    Ro: 0    </PrevB>
                <OldHd>  Loc: (665, 334)  Pr: 0    Ti: 1    Ro: 0    Dr: (-0.967093, -0.254422)
        Hd: (0.254422, -0.967093) </OldHd>
                <NewHd>  Loc: (665, 333)  Pr: 0    Ti: 1    Ro: 0    Dr: (-0.886641, -0.462459)
        Hd: (0.462459, -0.886641) </NewHd>
        </StrokeHeader>
        Wait: 0.000s     Loc: (666, 338)  Pr: 1    Ti: 1    Ro: 0    Rv: NO   Iv: NO
```

Lines starting with '//' are comment lines. They are ignored by the script engine. See the Comments section.

The block starting with <Version> identifies the ArtRage and script language versions to the ArtRage script parsing engine. The block starting with <Header> gives information about the script and author. And so on. Don't be intimidated by the '<StartupFeatures>' block – it is there to set ArtRage into a known state before the script plays back. It saves the person recording the script having to manually record all the settings of all the tools, and all the paper and colour settings etc., to get the script on playback to look the same as the recorded result. Recorded blocks are surrounded by <block> </block> to identify them and their contents to the script parser. The <Events> block is the exception – it doesn't need to be closed as the block is closed by falling out the end of the script text file and is always the last block.

The script programming language elements don't need to be surrounded by block identifiers. Anything the script parser doesn't recognize as part of its own recording it assumes is part of the written programming. In general the C-style function definitions should appear before the <Events> block of the script, and the main body of the script should be inside the <Events> block. For example:

```
real VectorLength(real x, real y)
{
        return sqrt(x * x + y * y)
}


<Events>

real rWidth = 200
real rHeight = 150
real rDiagonal = VectorLength(rWidth, rHeight)
```

Recorded scripts are recorded and played back in a linear sequence. The script starts, event 1 happens, then event 2, then event 3 and so on until the script ends. With programmed scripts flow-

control can change the order of events. Events can be put into functions which can called many times with different settings.

The language

The ArtRage scripting language is based on a subset of the C language. A programmer with C language experience should immediately be able to start producing complex scripts once they're aware of some limitations of the ArtRage scripting language.

Comments

Comments can be used in script files to make them easier for humans to read and understand. Comments are entirely ignored by the script engine.

Lines starting with '//' are comment lines. Comments can appear on a line after a scripted command – everything after the '//' is ignored by the script engine and is solely for the benefit of humans. These are C-style comments and not legal XML. C-style block comments /* … */ can also be used to comment out larger blocks of script files so they're ignored by the script engine. Note that currently you cannot use block comments to comment out part of a line.

For example 'n = 3 + 4 /* + 5 */' wouldn't work in the script file (even though it is legal in a C-style program).

Examples of comments:

```
// This is a comment on a line. The whole line is ignored
int n = 3;        // n is given the value of three.
/* n = 4;
n = 5
n = 6 */          // All three of these lines are commented out and ignored.
```

Use comments often. Comments help anyone reading your script to understand what the script does and why you wrote it that way. Have pity on your future self who will need to understand and debug the script in six months time – give yourself plenty of help with useful comments, written when you actually understood what you were thinking when you wrote the script.

Statements

In general the script is evaluated line-by-line. You can have more than one statement per text line so long as those statements are separated by a semi-colon ';'. Unlike the C language a line with only a single statement doesn't enforce the ';' rule. For example:

```
int n = 3                 // Only statement on the line, so a semi-colon isn't enforced
int x; n = x + n;         // Semicolon required after the 'x' but is optional after the 'n'
```

Variables

Four types of variables are defined for the scripting language: int, real, flag, and string. Each are described here

- int variable stores a signed whole number of 32 bits. It is equivalent to the C-language 'long int' type. Use it for numbers where you don't want or care about decimal places, or where you want to do 'bitwise' operations. (See Operators). If you assign a real number to an int variable the decimal component is dropped off.

- real variable stores a single-precision floating-point number. It is equivalent to the C-language 'float' type. Use it where you care about accuracy to decimal places. If you mix real numbers with int variables in an operation the ints are promoted to real numbers.
- flag variable stores a true/false or yes/no value. They are useful for choosing behaviours or different paths to follow in the script. They accept these values for 'true': True, Yes, 1. They accept these values for 'false': False, No, 0. The character case of True, False, Yes and No is not important – YES is the same as Yes, TRUE is the same as True.
- string stores a sequence of Unicode characters. Use it for storing messages, names – anything with text. 'string' variables have some special features:
    - When you assign text to a string surround it with quote marks "". For example: string s = "A piece of string"
    - You can use certain 'escape' characters to add special characters to the string. \r inserts a 'carriage return', \n inserts a 'newline', \t inserts a 'tab', \" inserts a quotation mark, \\ inserts a '\' character. \% also inserts a '%' character but isn't generally needed unless you want two or more %'s next to each other - %% string s = "A piece of string\n\r on two lines.\tWith a tab character. "
    - You can insert variable values directly into a string of text by preceding the variable name with %%. For example: int n = 3; string s = "On the count of %%n"; string t = "%%s, jump! "; // t = "On the count of 3, jump! "
    - Strings can also be assigned the value of other variables – they are converted to string equivalents. For example: int n = 3; string s = n; // s = "3" flag f = Yes; s = f; // s = "true" (flags always evaluate to the strings 'true' or 'false')
    - You can concatenate strings with the '+' and '+=' operators. For example: string s = "A piece "; string t = s + "of"; t += " string. " // t = "A piece of string. " s += " of " + 3.1415926; // s = "A piece of 3.1415926"
    - Strings can be compared with the '==' and '!=' operators to see if they match. For example: string s = "Fred"; if (s == sName && s != "Harry") …
    - Strings have functions which work directly on their contents (See String manipulation functions)

To declare a variable the syntax is very similar to the C language:

```
int n = 3;
string sMyString = "Hello there";
real rNum;
flag fValid = Yes
int n2 = n * 2;
```

The above example demonstrates how you can assign values at the same time as you declare the variables. Variable names must start with an alphabetic character but can contain any mix of alpha-numeric characters. The variable names are case sensitive: int n is a different variable to int N.

When variables are declared the assigned values can be complex statements, as in the example above with n2.

If variable names are preceded with 'global_' they are put into a global namespace, which means they can be accessed from any function or from another script called from the first script. If you try to declare a global variable more than once, both instances will refer to the same variable. This means script fragment files called from the main script can affect global variables.

```
int global_n = 3;
global_n = 4 + 2;
…
int global_n = 5;  // Will not cause an error – global_n takes the value 5 even if this
                   // is in a different function, or script fragment file. It's the same variable.
```

Numeric values can be expressed as simple numbers: 1, 6, -5 or decimal numbers: 1.2, -4.897, or exponent floating points: 1e12, -2e3, 4e-7, or as hexadecimal (base 16): 0x03e, 0x045, -0x0fa10

Wherever you see a numeric value or string-literal value in a recorded ArtRage script you can replace that value with one of your variables, or with a complex expression. For example:

```
Painting Name: sName
…
<EventPt> Loc: (x, y + 100)        Pr: rPres        Ti: 1    Ro: 0    Rv: NO   Iv: NO   </EventPt>
```

Dynamic Arrays

In addition to the standard variable types the ArtRage scripting language supports dynamic arrays of the standard variables. The four types of dynamic arrays are realarray, intarray, stringarray and flagarray – one type of array for each of the standard variable types.

Dynamic arrays are collections of variables of the same type. You can set the size of dynamic arrays, add elements to the arrays, access the individual elements of the array using square brackets '[]'. You can copy arrays using the equals operator '=' and join arrays together with the '+' and '+=' operators.

To declare a dynamic array you can use syntax similar to the standard variables:

```
intarray aNum
stringarray aNames
```

You can initialize an array to set its size and the values of its elements by using curly braces '{}'. The example below creates an array with four elements, with each element valued at 2, 4, 6, and 8 respectively.

```
intarray aNum = { 2, 4, 6, 8 }
```

The ArtRage scripting language has functions built in to help work with dynamic array variable types (See Array functions).

Here are some more examples of using dynamic arrays and accessing elements in the arrays.

```
stringarray sWorkDays = { "Mon", "Tue", "Wed", "Thu", "Fri" }
sWorkDays[0] = "Blah"; // Monday is now Blah.
stringarray sWeekendDays = { "Sat", "Sun" }
stringarray sWeekDays = sWorkDays + sWeekendDays
// sWeekdays is { "Blah", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun" }
intarray nNum;
nNum = { 2, 4, 6, 8, 10 }
nNum[1] = nNum[3] + nNum[4];  // nNum is now { 2, 18, 6, 8, 10 }
nNum.SetSize(2);  // nNum is now { 2, 18 }
```

Operators

Most of the C-language operators are supported and generally work the same way as their C counterparts. Operators generally take two values (operands) and return a result. In some cases the types of operands can be mixed – you can add an int type variable to a real type variable which will return a real type result. The order of evaluation of operations is given by the evaluation priority (see Table 1: Operators) . Operators with lower evaluation priority numbers are evaluated before

higher evaluation priority numbers. Operators with equal priorities are evaluated left-to-right as they're encountered. The contents of braces '(' and ')' is evaluated first, and braces can be nested.

Some examples of operator priority and braces:

- x = 3 - 2 + 4; Operator '+' and '-' both have a priority of 6, so x evaluates to 5
- x = 3 - (2 + 4); The braces mean the 4 + 2 is evaluated first so x evaluates to 3 – 6, which is -3
- x = 3 + 4 * 5; The '*' (multiplication) operator is higher priority (priority 5) so is evaluated before the addition (priority 6). x = 3 + 20; x = 23
- x = (3 + 4) * (2 + (6 * 7)); Evaluates to: x = (7) * (2 + (42)), x = 7 * 44, x = 308

Table 1: Operators

Operator	Evaluation priority	Description	Examples
=	16	Assignment operator. Set a variable to a value.	int n = 3 int r r = n
Math operators – basic mathematic operations			
+	6	Addition	R = 3 + n
-	6	Subtraction	R = 3 – n
*	5	Multiplication	R = 3 * n
/	5	Division	R = 3 / n

%	5	Modulus – remainder of division. When used with type real it is equivalent to fmod	R = n % 3; // R is the remainder of n / 3
			S = 9.1 % 2.5; // Result is 1.4

Bitwise operators – used with type int variables to affect individual binary bits			
&	10	Bitwise 'AND' operation. Set result bit to '1' if bit in both operands is '1'	N = 3 & 2; // Result is 2 (binary 11 & 10 == 10)
|	12	Bitwise 'OR' operation. Set result bit to '1' if bit in either operands is '1'	N = 3 | 2; // Result is 3 (binary 11 | 10 == 11)
^	11	Bitwise 'XOR' (exclusive or). Set result to '1' so long as bit in only one of the operands has a '1'. For example: 0^0 = 0, 1^0 = 1, 0^1 = 1, 1^1 = 0	N = 3 ^ 2; // Result is 1 (binary 11 ^ 10 == 01)
<<	7	Binary shift left. Shifts bits higher by one bit. Equivalent to an integer multiply by 2	N = 3 << 2 // result is 12 (binary 1100 – zeros are added to the right)
>>	7	Binary shift right. Shifts bits lower by one bit. Equivalent to integer division by 2	N = 3 >> 1; // result is 1 (binary 01 – bits fall off right)
Boolean (flag) operators.			

==	9	Logical equivalency ("is equal to"). Returns a flag value of 'True' if operands are the same value.	int n = 3;
			if (n == 3) s = n; // True, so 's = n' executes
			flag f = n == 3; // f takes the value 'True'

!=	9	Logical inequality ("is not equal to"). Returns a flag value of 'True' if operands are different values.	int n = 3; if (n != 3) s = n; // False, so 's = n' doesn't execute

>	8	Greater than. Returns 'True' if the left-hand operand is numerically higher than the right-hand operand.	int n = 3;
			if (n > 2) s = n; // True
			flag f = n > 3; // f takes the value 'False'. (3 isn't greater than 3)

<	8	Less than. Returns 'True' if the left-hand operand is numerically lower than the right-hand operand.	int n = 3; if (n < 2) s = n; // False

>=	8	Greater-than-or-equal-to. Returns 'true' if the left-hand operand is the same as or numerically greater than the right-hand operand	int n = 3;
			if (n>=3) s = n; // True.
			flag f = n >= 2; // f = True

<=	8	Less-than-or-equal-to. Returns 'true' if the left-hand operand is the same as or numerically lower than the right-hand operand	int n = 3;
			if (n <= 3) s = n; // True
			flag f = n <= 2.9; // f = false

&&	13	Logical 'AND'. Returns 'true' if both the left-hand and right-hand operands evaluate to 'true'	int n = 3; int s = 4
			if (n < s && n > 2) // True
			flag f = n == 3 && s == n; // False

||	14	Logical 'OR'. Returns 'true' if either the left- or right-hand operands evaluate to 'true'	int n = 3; int s = 4 if (n < s || n > 2) // True

			flag f = n == 3 || s == n; // true
Unary operators.			

-	3	Negative numeric value	int n = -3;
			int s = -n;
			int t = -(s * n)

!	3	Logical 'NOT' operator. Makes 'True' results into 'False'	int n = 3;
			if (!(n > 4)) // true – 3 is not greater than 4
			flag f = True; flag d = !f; // d = False

~	3	Bitwise invert. In integer values, underlying '1' bits become '0' and '0' bits become '1'	int n = 3; // 0x03 int s = ~n; // s = 0x0fffffc
Compound assignment operators – perform an operation on a variable and assign the result to the same variable.			

+=	16	Mathematical Addition with assign.	int n = 3;
			n += 5; // n now equals 8
			n += 3 * n; // n now equals 32

-=	16	Mathematical subtraction with assign.	int n = 3;
			n -= 5; // n now equals -2
			n -= 3 * n; // n now equals -8

*=	16	Mathematical multiplication with assign.	int n = 3;
			n *= 5; // n now equals 15
			n *= 3 * n; // n now equals 225

/=	16	Mathematical division with assign	int n = 3; n \= 5; // n now equals zero. Integer divisions are truncated!
%=	16	Mathematical modulus with assign. Remainder of division, and assign.	int n = 3; n %= 5; // n now equals 3 – remainder of division of 3 / 5
&=	16	Bitwise 'AND' with assign. Binary bit values are compared.	int n = 3; n &= 5; // n now equals 1
|=	16	Bitwise 'OR' with assign. Binary bit values are compared.	int n = 3; n |= 5; // n now equals 7
^=	16	Bitwise 'XOR' with assign. Binary bit values are compared.	int n = 3; n ^= 5; // n now equals 6
<<=	16	Bitwise shift left with assign.	int n = 3; n <<= 5; // n now equals 96 (binary 1100000)
>>=	16	Bitwise shift right with assign	int n = 3; n >>= 5; // n now equals zero because all the bits fell off the right
Post increment/decrement. (Note, these aren't valid in compound statements)			
++	2	Increment operand by 1	int n = 3; n++; // n now equals 4
--	2	Decrement operand by 1	int n = 3; n--; // n now equals 2
Array operators. These are the only operators that work on an entire dynamic array			
=	16	Assignment (copy) operator. Copy the entire contents of the array	intarray a intarray b = { 2, 3, 4} a = b;
+	6	Add the entire contents of one array onto the end of another.	intarray a = {1, 2} intarray b = a + a // result b = {1, 2, 1, 2}

+=	16	Add the entire contents of one array onto the array, and assign the result to the first array.	intarray a = {1, 2}
			intarray b = {3, 4}
			a += b;
			// result: a = {1, 2, 3, 4 }

File Variables.

The ArtRage scripting language has a special type of variable to help with reading and writing to files. The variable is the 'file' variable and is declared similarly to the other types of variables:

```
file f
file filDocFile
```

However unlike the other variable types you cannot assign a value and you cannot use operators with file variables. To set file variables, to open files, to read and write to files you need to call the functions on the file variable. (See File Variable functions)

To call a function using a file variable you use a dot '.' after the name of the variable then the function name. For example:

```
file filDoc
flag fSuccess = filDoc.Open("C:/Banana.txt")
if (fSuccess) {
   int nLen = filDoc.Length()
   MessageBox("The length of Banana.txt is %%nLen bytes")
   filDoc.Close()
}
```

The above code would open a file called Banana.txt on the C: drive if it existed, and would bring up a message box with the number of bytes in the file.

Note: When specifying file paths '\' and '/' can both be used. However because '\' is the special character, you need to use '\\' wherever you would want to use a single '\' in a path name. So filDoc.Open("C:\\Banana.txt") has exactly the same effect as filDoc.Open("C:/Banana.txt")

You cannot create a function of type file (See Function definitions). Trying to declare a function of type 'file' will cause a script error.

When you pass a 'file' variable as a parameter to one of your own functions the file variable isn't copied. Instead a reference to the same variable is passed to the function and any file functions you call affect the original file. This differs from other variable types where a copy is passed to a user function. For example:

```
void CheckScore(int nScore, file filTable) {
  filTable.SetPos(0)
  nScore *= 10;       // Improve my score
  int nOld = filTable.ReadInt32()
  if (nOld < nScore) {
     filTable.SetPos(0)
     filTable.WriteInt32(nScore)
  }
}

file filScoreTable
filScoreTable.Open("C:/score.bin")
int nMyScore = 15
CheckScore(nMyScore, filScoreTable)
```

In the above example passing filScoreTable to the user-defined function CheckScore allows the function to make changes to the file score.bin however the variable nMyScore wasn't affected when we multiplied nScore by 10 inside the function. nScore was a unique and independent copy of nMyScore, but filTable was a reference to (and affects) the original filScoreTable.

Flow control.

One of the biggest advantages of a programmed script over a recorded script is that the program can affect the order in which a script executes. With a recorded script events occur sequentially – they are played back in same order in which they were recorded. 'A' is followed by 'B' which is followed by 'C' because that's the order they were recorded. With a programmed script 'A' may execute, then 'B' may execute 20 times, then 'A' may execute again with different settings.

Blocks

A script can use curly braces '{' and '}' to define a block of code. For every open brace '{' you must have a close brace '}', and blocks can be nested – blocks inside blocks.

Blocks are used to define the extent of the chunk of code executed after a flow-control statement. This is explained more fully in the definitions of the flow control statements below.

Variables declared inside a block are local to that block. For example:

```
int nHeight = 100;        // Total height of all things.
int nWidth = 200;         // Total width.
{
   int nHeight = 20;      // Height of one thing.  This is a completely different 'nHeight'
                          // which hides the existence of the other 'nHeight'
   int nArea = nWidth * nHeight;  // nArea = 200 x 20 = 4000.
}
int nA2 = nArea;          // Error!  nArea was only valid in the scope of the block.
int nH2 = nHeight;        // nH2 = 100.  This refers to the first nHeight before the block.
                          // The nHeight that was inside the block has vanished - gone forever.
```

If/then/else

A script can make decisions about whether to execute statement block by comparing some values. The syntax of the 'if' statement is:

if (*expression*) *statement*

where '*expression'* is replaced with something that evaluates to 'true' or 'false', and '*statement*' is the single script statement or block (surrounded by curly braces) that executes if the expression is 'true'. If the value is false the code block after the 'if' statement is completely ignored. For example:

```
int n = 3;
if (n >= 3) {
         string sMessage = "The value of n is %%n";
         MessageBox(sMessage);
}
```

In the example above the block of code after the 'if' statement would be executed – the message box would be displayed. If n were equal to 2, then the expression would be 'false', and the entire block with the string and message box would be ignored.

The '*statement'* part of the 'if' statement can be a single statement or a whole block.

```
if (n >= 3) MessageBox("The value of n is %%n");    // Equivalent to code above.
```

If the expression is 'true' you may want to execute one block of code, and if the expression is 'false' execute a different block of code. That is where 'else' is used:

if (*expression*) *statement1* else *statement2*

If '*expression'* evaluates to 'true' then '*statement1'* would be executed and '*statement2*' would be completely ignored. But if '*expression'* evaluated to 'false' then '*statement1'*would be completely ignored and '*statement2*' would be executed instead. For example:

```
int n = 3;
if (n == 1)
{
   MessageBox("n is very small")
}
else MessageBox("n is not so small")
```

The ‘not so small’ messagebox would be displayed. Note how the example mixes block and single statement versions of if/else.

Several comparisons can be done and the first one which evaluates to ‘true’ will execute its block with the others ignored. This is the ‘else if’ part of the ‘if’ statement.

```
int n = 3;
if (n == 1) MessageBox("n is very small");
else if (n == 2) MessageBox("n is middling");
else if (n == 3) MessageBox("n is not so small");
else if (n < 10) {
  n *= 3;
  MessageBox("n is now much bigger")
}
else n = 0;
```

In the example above the ‘not so small’ message box will be displayed and all other code ignored. The (n < 10) code is also true, but only the **first** expression which returns ‘true’ gets its code executed. If n had been 4, then the “much bigger” block would have been executed. If n had been 10 then the last ‘else’ statement would have been hit.

The ‘*expression’* part of the ‘if’ statements can be very complex. You can have complex statements, get values from functions, use the logical operators.

```
if (sDay == "Thursday" && ((Cost(nCheese) > (10 * sqrt(gdp))) || (Weather() != fWet))) {
   flag fGo = YesNoBox("Do I really want to go outside? ");
   if (fGo)  …
```

For/next

Often a script will need to repeat a process many times. A ‘for’ statement allows a block of script code to execute a set number of times. Here is the syntax:

for (*start-expr*; *test-expr*; *increment-expr*) *statement*

When the script engine hits a ‘for’ statement it first evaluates ‘*start-expr’* – typically this is used to set the base value of the loop counter. Then it evaluates the ‘*test-expr’* – if this evaluates to ‘false’ the for loop ends but if it’s true the script engine executes ‘*statement’. ‘statement’* could be a single statement or it could be a block of code surrounded by curly braces ‘{‘ and ‘}’. After the code in ‘*statement’* has executed the script engine executes ‘*increment-expr’*. Typically Then it tests the ‘*test-expr’* again and if it is ‘true’ executes the ‘*statement’* code again. This repeats until ‘*test-expr’* becomes false. Then the script will carry on at the next statement past the end of the ‘*statement’* block.

The above can be a bit difficult to visualize until you understand how a ‘for’ statement is typically used:

```
int n ;
for (n = 0; n < 4; n++) {
   MessageBox("We're on loop number %%n");
}
```

When the script engine hits the ‘for’ statement it sets n to zero. Then it checks if n < 4, which is true. So it executes the code block to print the message. Then it executes the n++ statement. Then it evaluates ‘n < 4’ which is still true. So the code block executes. Eventually n will increment to 4. At that point evaluating ‘n < 4’ becomes ‘false’ so the ‘for’ loop stops. The ‘for’ loop executed four times, with n stepping through the numbers from 0 to 3 (inclusive) inside the loop.

The above example is the typical way 'for' loops are used – stepping positively through a set number of loops. However it's not the only way. The *'start-expr', 'test-expr'* and *'increment-expr'* are nothing special – they can really be any expression. You could start with n = 100, test that it is greater than zero, and decrement n. 'for (n = 100; n > 0; n--)' Or you could evaluate a function, or compare strings, or something else.

While loop

If the script needs to process a block of code until some condition is met rather than just a set number of times you can use a 'while' loop. The syntax is very similar to the 'if' statement:

while (*expression*) *statement*

So long as '*expression*' evaluates to 'true' the code in '*statement*' is repeated. When the script engine hits the 'while' statement it evaluates the '*expression*'. If that evaluates to 'true' the '*statement*' code is executed. At the end of the statement block the script loops back to the '*expression*' and evaluates it again. If it is still true '*statement*' is executed again. And so on until '*expression*' becomes false. It is important that something inside the '*statement*' block changes something to make '*expression*' eventually become false or the 'while' loop would never end.

```
flag fHappy = true;
int nCount = 0
while (fHappy) {
  fHappy = YesNoBox("At count %%nCount are you still happy? ");
  nCount++;
}
```

In the above example the while loop starts with fHappy evaluating to 'true'. The code block pops up a message box asking if you are still happy. If the user selects 'yes' fHappy is still true and the loop executes again. Eventually they will select 'no', fHappy takes on the value 'false' and the loop will end when the script next checks the 'while' expression. It's important to note that the loop doesn't stop at the point where fHappy was set to true, it stops when the 'while' loop evaluates fHappy. This means even though the user selected 'no', nCount was still incremented before the script engine looped back up to the 'while' statement to evaluate 'fHappy' and end the loop.

Break/Continue

Sometimes when you're writing a 'for/next' loop or 'while' loop you need to drop out immediately. Or you may get half way through the code block inside the loop and decide you don't need to execute the rest but you'd like to move onto the next iteration of the loop.

To exit a 'for/next' loop or 'while' loop immediately you can use the 'break' statement.

```
flag fHappy = true;
int nCount = 0
while (fHappy) {
  fHappy = YesNoBox("At count %%nCount are you still happy? ");
  if (!fHappy) break;
  nCount++;
}
```

In the example above the 'while' loop will exit immediately at the 'break' statement when fHappy becomes false. The nCount++ wont happen if fHappy becomes false.

To immediately move onto the next iteration of a loop use the 'continue' statement. When the script engine hits a 'continue' inside a block of code for a 'for' or 'while' loop it immediately returns to the start of the loop. In the case of a 'for' loop this means it will then increment the loop counter and re-

evaluate the end condition. In a 'while' loop it will test the end condition. Then it may execute the '*statement*' block or not depending on the condition.

```
for (int n = 0; n < 1000; n++) {
  if (n % 2) continue;
  if (n % 3) continue;
  if (n % 5) continue;
  MessageBox("The first number divisible by 2, 3 and 5 is %%n");
  break;
}
```

In the above example the script will loop through 1000 numbers looking for the first number divisible by 2, 3, and 5 with no remainder. If it never finds one the loop would end after 1000 tests. If it finds a number that isn't divisible by 2 the 'continue' statement makes the loop execute again. If it finds a number divisible by 2 it tests for divisibility by 3. If that fails the 'continue' will go back to the loop and move on to the next n. Eventually (when n == 30) the message box will be displayed. After that the 'break' statement will cause the loop to fall out.

If there are nested 'for/next' or 'while' loops the 'break' and 'continue' statements will affect only the innermost loop block in which they appear. For example:

```
flag fHappy = true;
while (fHappy) {
  for (int n = 100; n >= 1; n--) {
    if (n == 42) continue;         // I don't like 42. Ignore it
    fHappy = YesNoBox("%%n loops to go.  Still happy?")
    if (!fHappy) break;
  }
  if (fHappy) MessageBox("Yay! Still happy! Let's loop again!");
}
```

In the example above the 'continue' statement is inside the 'for' loop, so it causes the script to evaluate the next 'for'. The 'break' statement is also inside the 'for' block so it causes the script to fall out of the 'for' block, and onto the statement following the 'for' block – the MessageBox.

Exit

If you want the script to end at a certain point you can use the 'exit' directive. The script will end immediately as though it had finished normally.

```
if (YesNoBox("Shall we give up?) == yes)) exit
MessageBox("So you want to carry on I see.")
```

In the example above if the user chose to give up the script would have ended immediately and the second message would never be seen.

Function calls

Use your own functions to define a block of script code to which you can give a useful name. You can pass parameters to functions; the function can act on those parameters and return a value as a result. Functions need to be defined somewhere in the script prior to their being called so the script engine knows about the function name. As the script engine steps through the script it may find a function definition. At that point the only action the script engine takes is to note of the name of the function and where it is in the script file. Then it completely ignores the function and steps past it. Later in the script when a function is called the script engine looks up its list of function names, finds the one with the matching name, jumps to the point in the script file where it found that function and then executes the statements inside the function block. When the function exits the script will return to the point just after where the function call was made.

Function definitions

Define functions somewhere in the script earlier than where they will be used - typically before the <events> block of the script. Function definitions cannot appear inside other function definitions. A function definition must be at the start of a new line in the script. The script statements which define the function must be inside a block of curly braces '{' and '}'. Here's the syntax for function definitions:

return-type function-name() { statements … }

or

return-type function-name(param1-type param1-name, …) { statements … }

which by example might look like this:

```
void CubeRoot(real rNum) {
  // Some code here
}
```

The 'return-type' of the function is what is handed back to the script after the function call and it can be one of five types:

- void: No value is returned from the function.
- int: An integer value is returned from the function.
- real: A real number is returned from the function.
- flag: A 'true/false' or 'yes/no' or '1/0' value is returned by the function
- string: A string value is returned.

A function **must** return a value if it is defined to return a value (void is the only type that is defined not to return a value), and the return value must be of the same type as the function definition.

The '*function-name'* is the name of the function. It must start with an alphabetic character, but can be any combination of alpha-numeric characters. Function names are case-sensitive. You should give functions meaningful names relating to what they do to make your scripts easier to read.

You can optionally pass parameters to the function. Inside braces '(' and ')' after the function name you can define the '*param-type'* and '*param-name'* for any number of parameters passed to the function. Separate each parameter definition with a comma. '*param-type'* is one of the four variable types (see Variables) and '*param-name'* is the name of a new variable.

Then the body of the function is defined by the curly braces block '{}'.

Here is an example of a function definition:

```
real Hypotenuse(real rSideA, real rSideB) {
  real rLength = sqrt(rSideA * rSideA + rSideB * rSideB);
  return rLength;
}
```

In the example above the function is of type 'real', which means it is going to evaluate to a real number result which it will give back to the point in the script where the function was called. The name of the function is 'Hypotenuse'. The function takes two parameters – a real number variable named 'rSideA' and a real number variable named 'rSideB'. The passed-in parameters act like

variable definitions – they create named variables of their types which the function can use inside its code block. After the parameter list come the body of the function – the block of script code which is executed when you call the function. Inside the curly braces we can define new variables, call other function, use flow control or any other script language statements.

If the function has a type, we **must** use a 'return' statement to exit the function with a value of the same type as the defined type of the function. The only function type which doesn't require a return value is a function type 'void'. Void functions can either end when the script gets to the enclosing close curly brace '}' or when the script encounters a 'return' statement without a returned value. For example:

```
void ResetGlobals()
{
  global_name = "";
  global_height = 0
}


void CheckName(string sName) {
  if (sName == "Andy") {
    MessageBox("Hi %%sName");
    return;
  }
  MessageBox("I don't know you, %%sName");
  return;
}
```

The 'ResetGlobals' function doesn't take any parameters, performs some task, then hits the end of the code block at which point the script continues from the point where the function was called.

The 'CheckNames' function takes one string parameter. It performs some task which causes it to exit with a 'return' early in the function. Or it doesn't exit early if sName isn't 'Andy', it carries on until it hits the later 'return' which causes it to exit at that point.

'return' can be anywhere inside a function block. If the function has a defined type, the 'return' needs to pass back a value of the same type. If the function has a defined type, there **must** be a return statement with the same type to exit the function – there will be a script error if a function with a defined type is allowed to execute to the enclosing curly brace '}'. If the function is of type 'void', the 'return' statements are optional.

Calling functions.

Let me stress again that when the script engine is working line-by-line through a script, when it gets to a function definition it doesn't execute the function. The only thing the script engine does is to take a note of the function name and note where it currently is in the script. Then it completely ignores the function, steps past the function code block and carries on. It may find another function definition which it also completely ignores aside from noting its name and location. At some point the script gets to a code statement which isn't inside a function definition. If that statement makes a reference to the name of a function which the script engine met earlier it saves its current position in the script file, looks up the function name in its list and finds the location in the script file of the function definition. Then it starts executing the script from the code inside the function block. When the function returns it unwinds back to the point in the script where the call was made and carries on. This is the function call. Inside a function block you can also call other functions – so long as the script engine has previously found a function definition with the same name as your function

call it will save its current place, move to the function definition and carry on inside that new function. Here is an example:

```
real HypSquared(real rSideA, real rSideB)
{
  return NumSquared(rSideA) + NumSquared(rSideB);   // Returns a hypotenuse squared: a * a + b * b
}

real NumSquared(real rNum) {
  return rNum * rNum;     // Returns a number squared: r * r
}

real Hypotenuse(real rSideA, real rSideB)
{
  real rLenSq = HypSquared(rSideA, rSideB);   // Get the square of the hypotenuse
  real rLength = sqrt(rLenSq);                // Hypotenuse is square root of a * a + b * b
  return rLength                              // Pass that value back to the script.
}

real a = 3;
real b = 4;
real c = Hypotenuse(a, b);
```

The script engine starts chewing through the script file line by line. It gets to the function definition for 'HypSquared'. All it does it take a note of the name and the location in the script file. Then it ignores that whole block and carries on. It hits the 'NumSquared' definition. Again it just notes the name and the location, and ignores the definition and carries on. Then it hits the 'Hypotenuse' function. Again it notes it, and ignores it. Eventually it hits the first code not inside a function definition. 'real a = 3;' So it creates a new variable of type 'real' called 'a' and gives it a value of '3'. Then it creates the real 'b' with '4'. Then it creates real 'c' and tries to give it a value, but meets function name instead. It finds two values inside the braces '(a, b)' so it puts those values onto a stack – a temporary holding area. It looks up the function name in its function list and finds it has a matching name at a location around half-way through the script. The engine takes a note of its current point in the script, and then it moves its current location in the script to the start of the definition of the 'Hypotenuse' function. At that new point in the script, after the definition of the name of the 'Hypotenuse' function, it finds two parameter definitions inside braces '(real rSideA, real rSideB)'. So it creates two new real-number variables named 'rSideA' and 'rSideB'. It finds it has a stack with two real-number values, so it assigns those values to the new variables. rSideA = 3, rSideB = 4. Then it executes the code inside the body of the function 'Hypotenuse'. The first line in the function says to create a real-number variable named 'rLenSq' with a value of – ah, another function. The script engine notes its current point in the file, evaluates the parameter list for the function call and puts the values on the stack, looks up the name of the called function 'HypSquared' finding it earlier in the script file, moves the script execution to that point, creates two new real variables (even though they have the same name as the earlier rSideA and rSideB these are two entirely new variables), gives them the values on the stack and starts executing the code inside the 'HypSquared' function. And that code says to return a value which is – ah, more functions. Location noted, parameters evaluated, code execution point moved to 'NumSquared', with the value of rSideA on the stack. And the code in 'NumSquared' returns a real-number value of rNum * rNum. rNum at that point was 3, so the return value is 9. The script unwinds back to where 'NumSquared' was called uses the value 9. Then it carries on and sees it needs to add – yet another function call. So NumSquared is evaluated again, this time with the value 4 and returns 16. The code execution inside the 'HypSquared' function adds the result of those function calls together and returns a real-number result of 25. The script unwinds again back into the 'Hypotenuse' function with the 25 value and assigns it to rLenSq. Still inside the 'Hypotenuse' code block, real rLength is assigned the result of another function call sqrt, passing it the value 25. Sqrt is one of the internal math functions

described here (Math functions), and returns the square root of a value. In this case sqrt(25) = 5, so rLength gets the value 5. The next line in the 'Hypotenuse' function is 'return rLength'. The script unwinds the last step taking the value of 5 and assigning it to the real-number variable 'c'. Then it carries on to the next statement. And so on.

Important things to note about that sequence:

- The code inside the function definitions was ignored by the script engine when it met the function definitions. Just the function's name and location were noted
- When the function call was made the parameters to pass to the function were evaluated and their result put on the stack which was then given to the function. **The parameters in the function call and the parameter definitions in the function definition have to match in type and quantity.** If the function has a parameter list expecting a real number and a string, you **must** call the function with a real number and a string as parameters. If the function is expecting three real numbers, you must call the function with three real number parameters.
- The parameters defined for the function call and any variables defined inside the function have scope local to the function – they vanish when the function returns. They fall out of 'scope' and they are deleted.
- Parameters inside function definitions and variable definitions inside function blocks are completely new variables even if they have the same name as variables outside the function.
- The type of the value returned from functions **must** match the type of the function declaration. 'void' functions are the exception.
- The definition of the function and its parameters **must** be on one line of script text. Function definitions cannot be split across two or more lines of text.

In the above example C programmers might have noticed that the definition of the function 'NumSquared' appears later in the script than the first time it is called. In the 'HypSquared' function, 'NumSquared' is called, but the definition for 'NumSquared' appears after the definition for 'HypSquared'. In the C language this would cause an error. However there is no error in the ArtRage scripting language because the script engine met the definition of 'NumSquared' before it got to code which called 'HypSquared' (which then called 'NumSquared'). Therefore the definitions of functions don't need to appear in the script earlier than they are referenced inside other functions so long as the script engine has passed over them prior to their being called.

Function recursion

A function can call itself. This is called recursion. It can be useful if you need to evaluate an iterative problem where the results of each iteration depend on the previous iteration.

For example if I wanted a function to draw concentric circles where each circle is half the radius of the previous I could write something like this:

```
void Circle(real x, real y, real rRadius) {
  // Code to draw circle here…
  …
  if (rRadius > 10) Circle(x, y, rRadius / 2);
}

Circle(300, 300, 500);
```

In the example above the circle function is called the first time with a radius value of 500. The Circle function draws a circle of radius 500 (that particular script code is left to your imagination), at the location 300, 300. rRadius is greater than 10 so Circle calls itself again with rRadius / 2. This iteration of Circle draws a circle of radius 250 at the location 300, 300. rRadius is greater than 10 so Circle is called again with rRadius of 125. And so on. The sixth iteration of Circle calling itself has a rRadius smaller than 10, so it doesn't call Circle again – the function falls out of its closing block. The script returns to the previous iteration of Circle. The previous iteration was processing the 'if' statement, which has now been completed so it too falls out the end of the function, back to the previous iteration. And so on - the recursed iterations of the Circle function unwind and the script returns to the point just after the Circle(300, 300, 500) call was originally made.

The ArtRage function call stack is not very large. If you recurse to more than around 50 levels you will get a 'stack overflow' error. If you are recursing to that level there is probably a better way of doing what you're trying to do anyway.

The 'Wait:' directive.

The 'Wait:' directive has some important implications for function calls.

When ArtRage records a script many of the scripted events are preceded with 'Wait: 0.018s' (for example). The 'Wait' directive tells the script engine to momentarily pause playback until that number of seconds has passed. This lets the script events play back at the same rate they were recorded. If the artist recording the script paused for a few seconds before making a paint stroke the script playback would pause for a few seconds before playing that stroke back again. If the option to play scripts back at full speed is selected all 'Wait:' directives are considered to be zero.

For very long waits the 'Wait:' directive can be expressed in minutes and seconds in the format 'Wait: m:s'. For example 'Wait: 2:23.5s' would pause the script for one minute and twenty-three-and-a-half seconds. If you want to pause for a variable amount of time, using a variable, you can surround the time component of the directive with braces '(' and ')'. For example 'Wait: (rTime)' where rTime is a real-number value of seconds.

The way the ArtRage script engine actually deals with the 'Wait:' directive internally has implications for functions. When the script engine finds a 'Wait:' directive it saves the state of the engine and enters an idle state. Once the time elapsed since the 'Wait:' directive was read is equal to the time in the 'Wait:' directive the script restores its state and processes the next statement of the script. It wakes up and carries on, so to speak.

When a function is defined to return a value, the point where that function is called may be in an incredibly complex situation. For example consider this:

```
real Hypot(real a, real b) {
  return sqrt(a * a + b * b);
}

flag fLong = Hypot(x, cos(t + 4 * n)) + sqrt(rLenSq) * 10 > 50;
```

It's a terrible statement, I know. Valid, but terrible. The way the value assigned to flag fLong is evaluated is very complex. There are functions inside parameter lists, operators that need to be evaluated in a certain order, and comparisons of the result with variable type conversions. The

ArtRage engine is unable to save the state of this (it's a recursive nightmare…) if it were to encounter a 'Wait:' directive inside the Hypot function. Fortunately there isn't one.

If, however, your function contained scraps of recorded ArtRage script it will probably have lines with 'Wait:' directives.

When a function with a defined type contains a 'Wait:' directive the wait is completely ignored. The script ignores the delay and carries on immediately processing the next statement after the 'Wait:' directive. Example of using a 'Wait:' directive in a function:

```
void SetToolSize(real rSize) {  // Sets the size of the currently selected tool.
  int nID = 0x0B2D05E64;
  Wait: 0.018s
  EvType: Command CommandID: SetToolProperty ParamType: ToolProp Value: { nID (Size), rSize }
}

SetToolSize(0.5);
```

The above is valid. I was lazy and cut'n'pasted the command line from a recorded script, so the 'Wait:' directive was included (but isn't actually required). The function can safely be called anywhere. However the following example wont respect the 'Wait:' directive:

```
void PaintFlower(real x, real y) {
  SetToolSize(0.5);  // As above.
  … // code to paint flower in the Monet style – very pretty.
}

flag RateMyFlower() {
  PaintFlower(200, 100);
  return YesNoBox("Do you like my flower?");
}

flag fLovely = RateMyFlower();
```

The SetToolSize function is valid as a void function with a 'Wait:' directive. The PaintFlower function is valid as a void function that calls a function that contains a 'Wait:' directive. However the RateMyFlower function returns a value, and it calls functions that contain 'Wait:' directives. This will cause the script engine to ignore all the 'Wait:' directives inside PaintFlower and SetToolSize – the flowers would be painted at the fastest speed possible. Therefore it is very important to be aware of where your wait directives are being used if you want them to actually wait.

Built-in functions

The ArtRage script engine supports many built-in functions. These are pre-defined functions which you can call, for some common tasks. Most of the C-Language mathematical functions are built in. Also the ArtRage script engine has functions to return and change information about strings. The script engine has functions for getting and setting parameters relating to the painting file and painting. And it has some general-purpose functions for interacting with the user as the script is being run.

Math functions

Most of the standard C-Language mathematical functions are included and built-in to the scripting language.

Table 2: Built-in math functions

Function Name	Function declaration	Description	Example

sin	real sin(real r)	Calculates and returns the sin of angle r (in radians)	real t = sin(3.1415926);
cos	real cos(real r)	Calculates and returns the cosine of angle r (in radians)	real t = cos(2 * pi);
tan	real tan(real r)	Calculates and returns the tangent of angle r (in radians)	real t = tan(rDir);
asin	real asin(real r)	Returns the arcsine of r. Return value is in the range of 0 – π radians	real r = asin(t);
acos	real acos(real r)	Returns the arccosine of r. Return value is in the range of 0 – π radians.	real rAngle = acos(t) / pi * 360
atan	real atan(real r)	Returns the arctangent of r. Return value is in the range of -π /2 – π /2 radians	real r = atan(t);
atan2	real atan2(real y, real x)	Returns the arctangent of y/x (if x equals 0, atan2 returns π/2 if y is positive, -π/2 if y is negative, or 0 if y is 0.)	real r = atan(rHeight, rWidth);
sqrt	real sqrt(real r)	Returns the square root of r. If r is negative sqrt will return an invalid number.	real rLen = sqrt(a * a + b * b);
abs	int abs(int n)	Return the absolute (positive) value of the argument. This is equivalent to the C-Language 'abs' function.	int n = abs(x); if (abs(xa - xb) > 20) fMoved = yes;
fabs	real abs(real r)	Return the absolute (positive) value of the argument. This is equivalent to the C-Language 'fabs' function.	real r = fabs(x); real rRoot = sqrt(fabs(dx));
exp	real exp(real r)	Return exponent of r. That is e to the power r, where e is the base of the natural logarithm.	real r = exp(rV);
log	real log(real r)	Return the natural logarithm (base e) of r	real r = log(rVal);
log10	real log10(real r)	Return the base 10 logarithm of r	real r = log10(rVal);
pow	real pos(real x, real y)	Returns x raised to the power of y	real rCubed = pow(r, 3);
mod	int mod(int x, int y)	Returns the remainder of the integer division of x / y.	int n = mod(a, b) if (mod(n) == 0) fDivN = yes
fmod	real fmod(real x, real y)	Returns the remainder of the floating point division of x / y.	real r = fmod(ry, ry) real rDecimal = fmod(rLen, 1);

String manipulation functions

The ArtRage script language includes built-in functions to work with text strings. The functions are called on the string variable with a dot '.' then the function name. For example: n = s.Length() sets the integer variable named 'n' to the length of the string variable named 's'.

Table 3: String functions

Function Name	Function declaration	Description	Example

Length	int Length()	Returns the number of characters in string. If this an empty string, Length returns 0	string s = "ArtRage"
			int n = s.Length(); // n = 7
			int nLen = (sFirst + sLast).Length();

Left	string Left(int n)	Return the leftmost n characters from string. If n is greater than the length of string, all of string is returned.	string s = "ArtRage"
			string a = s.Left(3); // a = "Art"
			string sInitial = sName.Left(1);

Right	string Right(int n)	Return the rightmost n characters from string. If n is greater than the length of string all of string is returned.	string s = "ArtRage" string sRag =s.Right(4); // sRag = "Rage"
Mid	string Mid(int nStart, int nLen)	Return the string from string which starts at the character index nStart and has length nLen characters. Index is zero-based.	string s = "ArtRage" s = s.Mid(2, 4); // s = "tRag"
ToUpper	void ToUpper()	Sets string to be all uppercase	string sShout ="hello." sShout.ToUpper()
ToLower	void ToLower()	Returns the all lowercase version of s	string sReply = "YES?" sReply.ToLower();
Trim	void Trim()	Removes whitespace from the start and end of the string. Whitespace includes spaces, tabs, and carriage returns.	string sClean = sInput.Left(4) sClean.Trim()
ClipLeft	void ClipLeft(int n)	Removes leftmost n characters from string	string s = "ArtRage" s.ClipLeft(3); // s = "Rage"
ClipRight	void ClipRight(int n)	Removes rightmost n characters from string	string s = "ArtRage" s.ClipRight(4); // s = "Art"
Find	int Find(string sFind)	If the string sFind exists as a substring of	string s = "ArtRage"

		string, return the index of the first character sFind inside string. If not found returns -1. The search is case-sensitive	int n = s.Find("Rage"); // n = 3
			n = s.Find("rage"); // n = -1
			s.ToUpper(); n = s.Find("RAGE"); // n = 3

SetHex	void SetHex(int n)	Sets the contents of the string to a Hexadecimal (base 16) representation of the integer value 'n'. The hex string is prefixed with '0x0'. The hex string represents either an 8-bit, 16-bit, or 32-bit number, depending on the magnitude of 'n'	string s; s.SetHex(31); // s = 0x01f
			n = 257
			s.SetHex(n * 2); // s = 0x00202
			s.SetHex(-1); // s = 0x0ffffffff

Array functions

These functions work on the dynamic array variables (realarray, intarray, stringarray and flagarray). They are called on the dynamic array variable with a dot '.' then the function name. For example: n = a.GetSize() sets the integer variable named 'n' to the number of elements in the dynamic array variable named 'a'.

Table 4: Dynamic Array functions

Function Name	Function declaration	Description	Example
SetSize	void SetSize(int n)	Set the number of elements in the array. Either expands or shrinks the array to match the requested size.	stringarray aMonth aMonth.SetSize(12) aMonth.SetSize(nMarsMonths)
GetSize	int GetSize()	Return a count of the number of elements in the array.	n = aMonth.GetSize() if (aDays.GetSize() == 7)…
Add	void Add(<type> n)	Add an element to the end of the array. 'n' must be a type of value the array type supports. For example with an intarray 'n' should evaluate to an int.	aMonth.Add("January") aAge.Add(nToday – nBirthday) aPrime.Add(7)
Add	void Add(<array type> a)	Add an entire array to the end of this array. Equivalent to the '+=' operator for arrays. Array 'a' **must** be the same array type as this. For example with an intarray 'a' **must** also be an intarray	stringarray sDays = { "Mon", "Tue" } stringarray sNights = {"Wed", "Thu" } sDays.Add(sNights) // Result is sDays = {"Mon", "Tue", "Wed", "Thu"}
Copy	void Copy(<array type> a)	Replace our entire contents with the contents of array 'a'. Equivalent to the '=' operator for arrays. Array 'a' **must** be the same array type as this. For example with an intarray 'a' **must** also be an intarray	intarray aNum = { 1, 2, 3, 4 } intarray aPrime = { 1, 2, 3, 5 } aNum.Copy(aPrime)
InsertAt	void InsertAt(int nPos, <type> n)	Insert an element into the array at index nPos (zero-based index). 'n' must be a type of value the array type supports. For example with an intarray 'n' should evaluate to an int.	intarray aNum = { 10, 20, 30 } aNum.InsertAt(1, 15) // Result is aNum = { 10, 15, 20, 30 }
InsertAt	void InsertAt(int nPos, <array type> a)	Insert an entire array element into the array at index nPos (zero-based index). Array 'a' **must** be the same array type as this. For example with an intarray 'a' **must** also be an intarray	stringarray s = {"a", "e", "f" } stringarray s2 = { "b", "c"} s.InsertAt(1, s2) // Result is s = {"a", "b", "c", "d", "e", "f"}

RemoveAt	void RemoveAt(int nPos)	Remove one of the elements from the array at the index (zero based) 'nPos'	stringarray d = {"Mon", "Tue", "Wed"}
			d.RemoveAt(0); // I don't like Mondays
			// Result is d = {"Tue", "Wed"}

RemoveAt	void RemoveAt(int nPos, int nCount)	Remove nCount number of elements from the array at the index (zero based) 'nPos'	stringarray d = {"Mon", "Tue", "Wed"}
			d.RemoveAt(1, 2); // Two days off
			// Result is d = {"Mon"}

Move	void Move(int nSrc, int nDst)	Move an element in the array from position 'nSrc' to the position 'nDst' (both zero-based).	intarray n = { 2, 4, 6, 8 } n.Move(1, 0) // Result is n = {4, 2, 6, 8}

File Variable functions

This group of functions work on the ArtRage script variable type 'file'. These functions are called on the file variable with a dot '.' then the function name. For example: n = fDoc.Length() sets the integer variable named 'n' to the length (in bytes) of the file described by the file variable called 'fDoc'.

Table 5: File Variable functions

Function Name	Function declaration	Description	Example

Open	flag Open(string s)
	flag Open(string s, flag fReadOnly)
		Open the file at the path given by 's' for read and write access (or optionally read-only). Returns 'true' if the file could be opened or 'false' if opening the file failed. You need to open a file before you can read or write data to it.	file filDoc;
			flag f = filDoc.Open("C:/banana.txt"); // open for read and write access.
			f = filDoc.Open("D:/News.doc", true); // Opened for read-only

Close	flag Close()	Close the file. Closing a file ensures all data is completely written to the file and allows other file variables to access the file. It isn't mandatory to close a file you have opened – when the file variable falls out of scope the file is automatically closed.	filDoc.Close();

LoadDialog	flag LoadDialog()	Brings up a file browser to allow the user to choose a file to open for reading. If the file exists and can be opened for read-only access the file is opened and 'true' is returned. If the file cant be opened or the user cancels, 'false' is returned.	flag fSuccess = filDoc.LoadDialog();
			if (fSuccess) … // do something with the file
			else { … // file couldn't be opened.

SaveDialog	flag SaveDialog(string sName)	Brings up a file browser to allow the user to choose a file to save to. sName gives the default name, but that can be changed by the user. If the file can be opened for read-write access, it is opened and 'true' is returned. If the user cancels or the open fails, 'false' is returned.	flag fYes = filDoc.SaveDialog("score.txt") if (fYes) {… // You can write and read this file else {… // Failed. Do something else.

Name	string Name()	Return the filename of the file. This is set when the file is opened (either with Open() or with the file dialogs)	s = filDoc.Name()
			filDoc.LoadDialog();
			if (filDoc.Name() == "fish.txt")…

FullPath	string FullPath()	Returns the full path including the name of the file. This is set when the file is opened (either with Open() or with the file dialogs)	s = filDoc.FullPath();
			filDoc.SaveDialog("Scores.txt");
			if (filDoc.FullPath() == sOld) …

Length	int Length()	Returns the length (in bytes) of the opened file. If the file is not open or valid, -1 is returned.	int nBytes = filDoc.Length() if (filDoc.Length() < 4) { .. // short or invalid file?

SetLength	flag SetLength(int n)	Set the length of the file to the number of bytes given in 'n'. If 'n' is shorter than the current file length, the file is shortend. If 'n' is longer, the file is lengthened and the contents are invalid until overwritten.	file filScore;
			filScore.SaveDialog("Scores.txt");
			filScore.SetLength(1000);

Pos	int Pos()	Get the byte position in the opened file where the next read or write will occur. If the file isn't open or is invalid, -1 is returned.	int n = filDoc.Pos(); // Get the current file position
			if (filDoc.Pos() > filDoc.Length() - 4) { … // near the end of the file.

SetPos	flag SetPos(int n)	Set the byte position in the opened file where the next read or write will occur. If the position is a valid location in the file, and the SetPos success, return 'true'. Returns 'false' on failure.	filDoc.SetPos(0); // At the start.
			filDoc.SetPos(filDoc.Length()); // at the end
			filDoc.SetPos(filDoc.GetPos() + 4); // Skip a 32-bit integer.

ReadUint8 ReadInt8	int ReadUint8() int ReadInt8()	Read and return a single byte (8 bits) from the opened file. On success the file current location (Pos()) is incremented by one byte. ReadUint8() returns the value as an unsigned inteter in the range 0 to 255. ReadInt8() returns the value as a signed integer in the range -127 to +127.	int n = filDoc.ReadUint8(); // 0 – 255 int m = filDoc.ReadInt8(); // -127 to +127
ReadUint16 ReadInt16	int ReadUint16() int ReadInt16()	Read and return a double byte word (16 bits) from the opened file. On success the file	int n = filDoc.ReadUint16(); // 0 – 65535 int m = filDoc.ReadInt16(); // -32767 to

		current location (Pos()) is incremented by two bytes. ReadUint16() returns the value as an unsigned inteter in the range 0 to 65535 ReadInt16() returns the value as a signed integer in the range -32767 to +32767.	+32767

ReadUint32
ReadInt32
	int ReadUint32()
	int ReadInt32()
		Read and return a 4- byte word (32 bits) from the opened file. On success the file current location (Pos()) is incremented by four bytes. Both ReadUint32() and ReadInt32() return the value as a signed integer in the range -2^31 to +2^31 because that's the largest value an int variable can contain.	int n = filDoc.ReadUint32();
			int m = filDoc.ReadInt32();
			// Both return values in the range -2,147,483,647 to + 2,147,483,647

ReadString	string ReadString()	Read and return a Unicode UTF-16 string (two bytes per character) from the file, up to the next carriage-return + line-feed. (0x000d, 0x000a). If the string wasn't written with a carriage-return + linefeed pair, the returned string will be invalid. The file pointer is incremented to the first byte past the cr/lf pair.	sName = filDoc.ReadString() sName.TrimRight(); // get rid of cr/lf
ReadAsciiString	stringReadAsciiString()	Read an Ascii string (one byte per character) from the file, up to the next carriage-return + linefeed. (0x0d, 0x0a). The string is then converted to UTF-16 and returned.	sDate = filDoc.ReadAsciiString();

WriteUint8
WriteInt8
	flag WriteUint8(int n)
	flag WriteInt8(int n)
		Write a single byte (8 bits) to the opened file. On success the file current location (Pos()) is incremented by one byte, and 'true' is returned. If the value couldn't be written, 'false' is returned. The effect of WriteUint8() and WriteInt8() is the same – both write the least-significant 8 bits of 'n' to the file.	int n = 200
			flag f = filDoc.WriteUint8(n); // 0x0C8 written to file
			f = filDoc.WriteInt8(n);// 0x0C8 also written to file.
			filDoc.WriteUint8(-1000); // 0x018 written to file. (-1000 = 0x0fffffc18)

WriteUint16
WriteInt16
	flag WriteUint16(int n)
	flag WriteInt16(int n)
		Write two bytes (16 bits) to the opened file. On success the file current location (Pos()) is incremented by two bytes, and 'true' is returned. If the value couldn't be written, 'false' is returned. The effect of WriteUint16() and WriteInt16() is the same – both write the least-significant 16 bits of 'n' to the file.	int n = 2000
			flag f = filDoc.WriteUint16(n); // 0x07D0 written to file
			f = filDoc.WriteInt16(n);// 0x07D0 also written to file.
			filDoc.WriteUint16(-1000); // 0x0FC18 written to file. (-1000 = 0x0fffffc18)

WriteUint32
WriteInt32
	flag WriteUint32(int n)
	flag WriteInt32(int n)
		Write four bytes (32 bits) to the opened file. On success the file current location (Pos()) is incremented by four bytes, and 'true' is returned. If the value couldn't be written, 'false' is returned. The effect of WriteUint32() and WriteInt32() is the same – both write all four bytes of 'n' to the file.	int n = 0x0ff2050ff
			flag f = filDoc.WriteUint16(n); // 0x0ff2050ff written to file
			f = filDoc.WriteInt16(n);// 0x0ff2050ff also written to file.
			filDoc.WriteUint16(-1000); // 0x0FFFFFC18 written to file. (-1000 = 0x0fffffc18)

WriteString	flag WriteString(string s)	Write the text string in 's' to the open file as a UTF-16 (two bytes per character) string. Then write carriage-return + linefeed (0x000d, 0x000a) characters to the file. On success the file pointer is incremented past the cr/lf pair and 'true' is returned. If the string couldn't be written 'false' is returned. Simply put: WriteString(s) writes the string that ReadString() will read.	flag f = filDoc.Write("Ha!"); // "Ha!/r/n" is actually written to file: 0x00048, 0x00061, 0x00021, 0x0000D, 0x0000A Bytes: 48 00 61 00 21 00 0D 00 0A 00
WriteAsciiString	flag WriteAsciiString (string s)	Write the text string in 's' to the open file as an Ascii (one byte per character) string. Then write carriage-return + linefeed (0x0d, 0x0a) characters to the file. On success the file pointer is incremented past the cr/lf pair and 'true' is returned. If the string couldn't be written 'false' is returned. Simply put: WriteAsciiString(s) writes the string that ReadAsciiStrig() will read.	flag f = filDoc.Write("Ha!"); // "Ha!/r/n" is actually written to file: 0x048, 0x061, 0x021, 0x00D, 0x00A Bytes: 48 61 21 0D 0A

ArtRage system functions.

This group of functions gets and sets information from ArtRage and from the painting. There are some sub-groups of functions relating to colour, mouse/keyboard input, transformation spaces, and miscellaneous.

Paint colour functions.

These functions affect the currently selected paint colour in ArtRage. You can get and set the values in RGB space or HLS space.

Table 6: ArtRage paint colour functions

Function Name	Function declaration	Description	Example
ColourH	real ColourH()	Returns the Hue value of the currently selected paint colour. Range 0 - 1	real rHue = ColourH();
ColourL	real ColourL()	Returns the Luminance value of the currently selected paint colour. Range 0 – 1	real rLum = ColourL();
ColourS	real ColourS()	Returns the Saturation value of the currently selected paint colour. Range 0 - 1	real rSat = ColourS();
ColourR	real ColourR()	Returns the Red channel of the currently selected paint colour. Range 0 – 1	real rRed = ColourR();
ColourG	real ColourG()	Returns the Green channel of the currently selected paint colour. Range 0 – 1	real rGreen = ColourG();
ColourB	real ColourB()	Returns the Blue channel of the currently selected paint colour. Range 0 – 1	real rBlue = ColourB();
ColourMetal	real ColourMetal()	Returns the Metallic channel of the currently selected paint colour. Range 0 – 1	real rMetal = ColourMetal();

SetColourHLS	void SetColourHLS(real H, real L, real S)	Sets the current paint colour with the HLS values (in range 0 – 1). Note this will also affect the RGB values.	SetColourHLS(0.3, 0.5, 1.0); // A loud colour
			SetColourHLS(cos(rAng), ColourL(), 0);

SetColourRGB	void SetColourRGB(real R, real G, real B)	Sets the current paint colour with the RGB channel values (in range 0 – 1). Note this will also affect the HLS values.	SetColourRGB(1, 0, 1); // Magenta!
			SetColourRGB(ColourR() / 2, ColourG(), ColourB()); // Fade the red a bit.

SetColourMetal	void SetColourMetal(real m)	Sets the metallic channel of the current paint colour. m is in range 0 – 1	SetColourMetal(0); // No metallic
			SetColourMetal(0.3); // Pearlescent

Layer Property functions.

These functions get information about layers in the current ArtRage painting.

Table 7: Painting Layer Property functions

Function Name	Function declaration	Description	Example
CurrentLayerIndex	int CurrentLayerIndex()	Get the index of the current selected layer.	int nLayer = CurrentLayerIndex();
LayerCount	int LayerCount	Get a count of the layers in the painting	int nLayerCount = LayerCount();
LayerName	string LayerName(int n)	Get the name of the layer ‘n’ in the painting.	string s = LayerName(0);
LayerOpacity	real LayerOpacity(int n)	Get the opacity (0 to 1) of the layer ‘n’ in the painting	real rOpac = LayerOpacity(1)
LayerVisible	flag LayerVisible(int n)	Get visibility flag for layer ‘n’ in the painting	flag fVisible = LayerVisible(0) if (!LayerVisible(1)) {…
LayerBlendMode	int LayerBlendMode(int n)	Get the blend mode for the layer ‘n’ in the painting.	int nMode = LayerBlendMode(1);
LayerBumpBlendMode	int LayerBumpBlendMode(int n)	Get the bump blending mode for the layer ‘n’ in the painting.	int m = LayerBumpBlendMode(2)
LayerType	int LayerType(int n)	Get the layer type for layer ‘n’. Layer types are: 0 = Paint, 1 = Group Open, 2 = Group Closed, 3 = Group End marker, 4 = Text layer, 5 = Sticker layer	int nType = LayerType(n); if (LayerType(n) != 0) { // Not paint layer
LayerPreserveTrans	flag LayerPreserveTrans(int n)	Get the transparency preserve flag for the layer ‘n’.	flag fLock = LayerPreserveTrans(1) if (LayerPreserveTrans(n)) {…

Mouse/Keyboard functions.

These functions get values from the mouse/stylus/graphics tablet input device, or from the keyboard. The Wait- functions can stop script execution until the user does a mouse action.

Note: These functions aren't currently supported by the ArtRage UI – currently scripts run 'modally', and don't allow user interaction. These functions will be enabled in a future release of ArtRage.

Table 8: Mouse/keyboard functions

Function Name	Function declaration	Description	Example
SampleMouse	void SampleMouse()	Causes the script to sample the mouse/graphics tablet. The sampled values are stored for access by other functions	SampleMouse();
MouseX	real MouseX()	Get the sampled mouse X coordinate in canvas pixel coordinates.	real x = MouseX();
MouseY	real MouseY()	Get the sampled mouse Y coordinate in canvas pixel coordinates.	real y = MouseY();
MousePressure	real MousePressure()	Get the sampled pressure value from the graphics tablet or stylus input device. Range 0 – 1	real rPressure = MousePressure(); if (MousePressure() > 0.9) fHeavy = Yes;
MouseTilt	real MouseTilt()	Get the sampled tilt value from the graphics tablet or stylus input device. Range (0 = horizontal, 1 = vertical)	real rTilt = MouseTilt();
MouseOrientation	real MouseOrientation()	Get the sampled rotation value from the graphics tablet or stylus input device. Range 0 – 1. If Tilt is 1, this value is meaningless.	real rRotation = MouseOrientation();
MouseInverted	flag MouseInverted()	Get the sampled stylus device's inverted state. True if the stylus eraser end is being used (at the time of the SampleMouse())	flag fErase = MouseInverted(); if (MouseInverted()) fDeadMouse = yes;
WaitResumeAtMouse	void WaitResumeAtMouse()	Stops script execution until the user clicks the mouse in the canvas, then offset strokes to be at mouse point. It also updates SampleMouse values.	WaitResumeAtMouse(); // Stop for click – next played-back stroke will be at mouse.

WaitSampleMouse	void WaitSampleMouse()	Stops script execution until the user moves or clicks the mouse. It will update the SampleMouse values and carry on.	WaitSampleMouse(); // Stop until click or mouse move.
			if (fDone) WaitSampleMouse();

MouseEventType	int MouseEventType()	Sampled mouse/stylus event type: 0 = MouseStart, 1 = MouseContinue, 2 = MouseEnd, -1 = Unknown	int nEvent = MouseEventType(); if (MouseEventType() == 2) fUp = true;
MouseButtonIsDown	flag MouseButtonIsDown()	Returns true if the mouse button is down. This is the state of the button when MouseButtonIsDown() is called, not the stored mouse values.	flag fDown = MouseButtonIsDown(); while (MouseButtonIsDown()) {…
KeyShiftIsDown	flag KeyShiftIsDown()	Return true if either shift key on the keyboard is down.	flag fConstrain = KeyShiftIsDown(); if (KeyShiftIsDown()) {…
KeyCtrlIsDown	flag KeyCtrlIsDown()	Return true if either of the Ctrl keys on the keyboard is down. (Command key on Mac keyboards)	flag fMove = KeyCtrlIsDown(); while (KeyCtrlIsDown()) {…
KeyAltIsDown	flag KeyAltIsDown()	Return true if either Alt key on the keyboard is down. (Option key on Mac keyboards)	flag fRot = KeyAltIsDown();

Transformation space functions.

Transformations allow the script engine to playback a recorded script in a different position, or at a different size and aspect, or at a different angle. Transformations can be pushed onto a stack, changed, then popped off the stack to restore the transformations prior to change. The transformation stack also stores the currently selected paint colours (see Paint colour functions.)

Table 9: Transformation functions

Function Name	Function declaration	Description	Example

TransScaleX	real TransScaleX()	Returns the current transformation scaling in the horizontal direction. The amount stroke positions are squashed or stretched.	real rScaleX = TransScaleX();

TransScaleY	real TransScaleY()	Returns the current transformation scaling in the vertical direction.	real rScaleY = TransScaleY();
			if (TransScaleY() > TransScaleX() * 2)
			fWideScreen = true;

TransCentreX	real TransCentreX()	Returns the X value of centre of rotation for scale and rotation transformations. Result is in canvas pixel coordinates. Defaults to half canvas width.	real rXCentre = TransCentreX();
TransCentreY	real TransCentreY()	Returns the Y value of centre of rotation for scale and rotation transformations. Result is in canvas pixel coordinates. Defaults to half canvas height.	real rYCentre = TransCentreY();
TransOffsetX	real TransOffsetX()	Returns thehorizontal amount strokes will be offset (in canvas pixel coordinates) from their recorded position	real rOffX = TransOffsetX();
TransOffsetY	real TransOffsetY()	Returns the vertical amount strokes will be offset (in canvas pixel coordinates) from their recorded position	real rOffX = TransOffsetX();
TransRotation	real TransRotation()	Returns the angle in radians (0 - 2 π) strokes will be rotated around the centre of rotation.	real rAng = TransRotation(); if (TransRotation() == rPi) fUpsideDown = yes;

SetTransScale	void SetTransScale(real x, real y)	Set the amount of horizontal and vertical scaling applied to paint strokes as they are played back. The scale also affects the tool size values (average scale of X and Y)	SetTransScale(2, 1); // Wide
			SetTransScale(1, 1); // Normal
			SetTransScale(TransScaleX() * 2, TransScaleY * 2); // Twice as big as it was

SetTransCentre	void SetTransCentre(real x, real y)	Set the centre point in canvas pixel coordinates for scale and rotation operations.	SetTransCentre(500, 500);
SetTransOffset	void Set TransOffset(real x, real y)	Set the amount strokes will be offset from their recorded position when they're played back. X and y are in canvas pixel coordinates.	SetTransOffset(100, 0); SetTransOffset(TransOffsetX() + 20, TransOffsetY());

SetTransRotation	void SetTransRotation(real r)	Set the angle (in radians) to rotate strokes on playback. Rotation happens around the TransCentre point.	SetTransRotation(0); // Not rotated
			SetTransRotation(pi); // Upside down.

PushState	void PushState()	Push the current transformations and colour values onto the stack.	PushState();
PopState	void PopState()	Pop the transformations and colour values off the stack. They are restored to how they were before the PushState() occurred.	PopState();

Message Functions.

To display a message to the person running the script use one of these functions.

Table 10: Message functions

Function Name	Function declaration	Description	Example
MessageBox	void MessageBox(string s)	Display a message box with the string as text of the message box. The script halts until the user presses 'ok' in the message box.	MessageBox("Here we go! "); MessageBox("%%x wide/n/r%%y high");
YesNoBox	flag YesNoBox(string s)	Display a message box with the string as text and a 'yes' button and a 'no' button. The script halts until the user presses a button. Returns a true value for 'yes' and a false value for 'no'	flag fContinue = YesNoBox("Want to carry on? "); if (YesNoBox("Is the number greater than %%guess? ") == yes) {…
MessageTip	void MessageTip(string s)	The script pops up a text message in a balloon which gradually fades out. The script isn't halted.	MessageTip("This is a squiggly line"); MessageTip("Notice I select the %%sName tool here");

PlaySoundFile	void PlaySoundFile(string s)	Load and play a file in .WAV format with the filename 's'. Script continues executing while the sound file plays.	PlaySoundFile("bang.wav"); // wake up!
			PlaySoundFile("OdeToJoy.wav"); // This is a fantastic painting to watch.

InputBox	void InputBox(string s)	Halts execution and brings up a modal dialog with the string s as text. Where text in s is preceded by $$ an input box to set that variable is created in the dialog.	string sName = "your name";
			InputBox("Enter name: $$sName");
			// sName is set to the entered value.
			InputBox("Size: ($$x, $$y) ");

NoteBox	void NoteBox(string s, int nWidth, int nHeight, real x, real y,int nEdge, real rDist, int nID)	Display a message in a box sized Width x Height pixels, pointing to location X, Y in the canvas. The 'tail' of the NoteBox will try to be on the edge (0 = Top, 1 = Left, 2 = Bottom, 3 = Right) at	NoteBox("Left ear", 100, 30, 512, 384,1, 0.5, -1) NoteBox("Watch this\n\r stroke here", 200, 80, 300, 300, 2, 0, 10);

		rDist (0 – 1) along edge. If nID = -1, the box will be modal(stops playback) with an OK button. If nID >= 0, box will be floating and can be dismissed with DismissNoteBox below	

DismissNoteBox	void DismissNoteBox(int n)	Dismiss a non-modal NoteBox. If n = -1 then all non-modal noteboxes will be dismissed. If n >= 0 only the notebox with that ID will be dismissed	DismissNoteBox(-1); // Too many noteboxes
			DismissNoteBox(10); // Stroke finished.

CanvasHighlight	void CanvasHighlight(real x, real y, real r, int nRGB)	Create a highlighted area of the canvas at the cords x, y with a radius r. nRGB is a hex colour value 0x0AARRGGBB	CanvasHighlight(500, 500, 100, 0x0ff000000); // Look at this interesting bit!
DismissCanvasHighlight	void DismissCanvasHighlight()	Dismiss the canvas highlight from the canvas	DismissCanvasHighlight(); // Not interesting anymore
Assert	void Assert(flag f)	This is more of a debugging tool (possibly in the future scripts will have a debugging mode to help developers). If f evaluates to false a message box with a warning and script information is displayed.	Assert(x < PaintingWidth()); Assert(sName == "Andy");

Miscellaneous functions.

A few more functions which are jolly useful but don't fit into a particular category.

Table 11: Miscellaneous functions

Function Name	Function declaration	Description	Example
PaintingWidth	int PaintingWidth()	Return the width of the current painting, in pixels	int nWidth = PaintingWidth();
PaintingHeight	int PaintingHeight()	Return the height of the current painting, in pixels	int nHeight = PaintingHeight();

PaintingDPI	int PaintingDPI()	Return the Dots-Per-Inch setting of the current painting	int nDPI = PaintingDPI();
			real rWidth = PaintingWidth();
			real rSizeInches = rWidth / PaintingDPI();

CurrentToolID	int CurrentToolID()	Returns the integer ID of the currently selected tool in ArtRage. Equivalent to the value used by the CID_ToolSelect recorded script event to change tool	int nTool = CurrentToolID(); if (CurrentToolID() == 4901) fPencil = yes;
CurrentToolSetting	real CurrentToolSetting(int nID)	Return the setting with id nID from the current tool. nID is equivalent to the property ID used by SetToolProperty in recorded scripts.	real rToolSize = CurrentToolSetting(0x0B2D05E64);

Random	real Random()
	real Random(real r)
	real Random(real a, real b)
		Random() returns a value between 0 and 1.
		Random(r) returns a value between 0 and r
		Random(a, b) returns an value between a and b
			real x = Random(PaintingWidth());
			real rAge = Random(21, 31);
			SetColourHLS(Random(), 0.5, 1); // Fun!

Noise	real Noise(real x, real y)	Smooth 2D Perlin noise. The period is about 1, so divide your x and y values by 10 to see noise	real rVal = Noise(x / 10, y / 10);
Cloud	real Cloud(real x, real y)	Smooth cloud noise. The period is about 1, so divide your x and y values by 10 to see cloud	real rVal = Cloud(x / 10, y / 10)
Randomize	void Randomize()	Restart the random number generator from a random position.	Randomize(); // They'll never find me now.
ScriptFile	void ScriptFile(string s)	Branches script execution into the file with the name in the string s. Useful for build scraps of scripts. Global variables can be used to pass values around script scraps.	ScriptFile("flower.arscript"); SetTransOffset(100, 100); ScriptFile("AnotherFlower.arscript");

File Functions.

These functions work on whole files. When specifying file paths '\' and '/' can both be used. However because '\' is the special character, you need to use '\\' wherever you would want to use a single '\' in a path name. So FileDelete("C:\\Banana.txt") has exactly the same effect as FileDelete("C:/Banana.txt")

Table 12: File functions

Function Name	Function declaration	Description	Example
FileExists	flag FileExists(string sPath)	Returns 'true' if a file exists at the pathname given by sPath. Returns 'false' if there is no file.	flag f = FileExists("C:/Banana.txt")
FileDelete	flag FileDelete(string sPath)	Delete the file at the path given by sPath. If the deletion is successful (or the file doesn't exist) return 'true', otherwise return 'false' if the file couldn't be deleted.	flag f = FileDelete("C:/Banana.txt");
FileCopy	flag FileCopy(string sSrc, string sDst)	Make a copy of the file given by sScr, and save it with the name and path given by sDst. Returns 'true' if the copy was successful, or 'false' if the copy failed. Copy keeps the original file, and makes a new file.	flag f = FileCopy("C:/Banana.txt", D:/Banana.txt")
FileMove FileRename	flag FileMove(string sSrc, string sDst) flag FileRename(string sSrc, string sDst)	FileMove and FileRename do exactly the same. Move the file given by sSrc to the location and with the name given by sDst. Use whichever one makes the most sense in the context of your script.	flag f = FileMove("c:/Banana.txt", "D:/banana.txt"); FileRename(refA.FullPath(), refB.FullPath())

Putting it together.

You could write an ArtRage script file entirely from scratch, declaring functions and writing code to do something. In general you probably want to paint something, using ArtRage tools, options and settings to give a painted result. Rather than trying to write everything from scratch it's easiest to record most of what you want done using the script recording functions then edit the resulting script file with Notepad or TextEdit.

Here are some sample scripts – partially recorded and partially coded. If you want to run these scripts copy the script into Notepad or TextEdit. Then save the text file as a Unicode file. **It is very important to save as a Unicode text file.** The ArtRage script engine will only work with UTF16 Unicode text files.

100 Flowers.

In the sample listing (Listing 1: 100 Flowers) I recorded a script to paint a single flower. Recorded scripts are quite verbose – most of the text at the start is to put ArtRage into a 'known' state. Then I opened the script with NotePad (on Windows) and made some slight changes to the script file. The first change was to surround the entire block of recorded paint strokes with a function definition:

```
//============================================================================
// Script data follows:
//============================================================================
<Events>
void PaintOneFlower() {
        …
}
```

Following the closing bracket of the function definition I added a block of ArtRage script code to call the 'PaintOneFlower' function 100 times:

```
Randomize()       // Different each time.
for (int n = 1; n <= 100; n++) {
    MessageTip("Flower number %%n")

    // Offset the flower from where it was originally painted at Loc: (525, 672)
    real rOffsetX = Random(-525 + 100, PaintingWidth() - 525 - 100)  // Left/Right half width
```

```
        real rOffsetY = 4 * n - 672 + PaintingHeight() / 2          // Move down slightly as we go
        SetTransOffset(rOffsetX, rOffsetY)                          // Set global transform
        PaintOneFlower()                                            // Paint the next flower.
    }
```

The (525, 672) is the first location the script recorded when I painted the flower – look for the first '<EventPt>' block. In the sample above I give the flower a random X placement between 100 and (Painting Width() – 100), and a slowly increasing Y placement from half the height of the painting downwards.

Then I added a couple of statements to randomly change the hue of the petals and centre of the flower. Because this was called just after the playback of the scripted colour change, the colour of the petals and the colour of the centre of the flower are based on the recorded colours, but with just the Hue component of the colour selected randomly.

```
SetColourHLS(Random(), ColourL(), ColourS());
```

And those are the only changes I needed to make to the recorded script of one painted flower to make it into 100 randomly coloured flowers painted in random positions.

Five circles.

The sample listing (Listing 2: Five Circles) is less of a recorded script and more of a script written from ArtRage Script language statements. The script skips all the usual recorded heading and preamble. This means whatever is played back with be working with the tools and settings the user has chosen before playback. If the user selects the crayon tool then runs this script, five circles will be drawn in crayon. If the user selects the watercolour brush and metallic green paint then runs the script, five circles will be painted with the watercolour brush in metallic green.

In the listing there is a function definition: 'void Circle(real rCX, real rCY, real rRad)' this function paints a circle centred on rCY, rCY with a radius rRad pixels. It paints the circle by using ArtRage scripted commands for a <StrokeEvent> block. Inside the Circle function these lines set the location of the stroke event:

```
<StrokeHeader>
    <EventPt>    Loc: (x, y)      Pr: 1    Ti: 1    Ro: 0    Rv: NO   Iv: NO   </EventPt>
    <Recorded>   No       </Recorded>
</StrokeHeader>
```

The <StrokeHeader> block sets up the stroke location, and stroke smoothing variables (which we don't care about generally). It also has the '<Recorded> No </Recorded>' statement which lets the playback engine know these points didn't come from a recording. With recorded strokes the playback engine needs to know to apply some smoothing to the stroke. If the flag is 'No' the playback engine assumes the strokes are written to be played back at the exact positions specified.

Then the circle is drawn with Loc: points of varying pressure around the circumference of the circle.

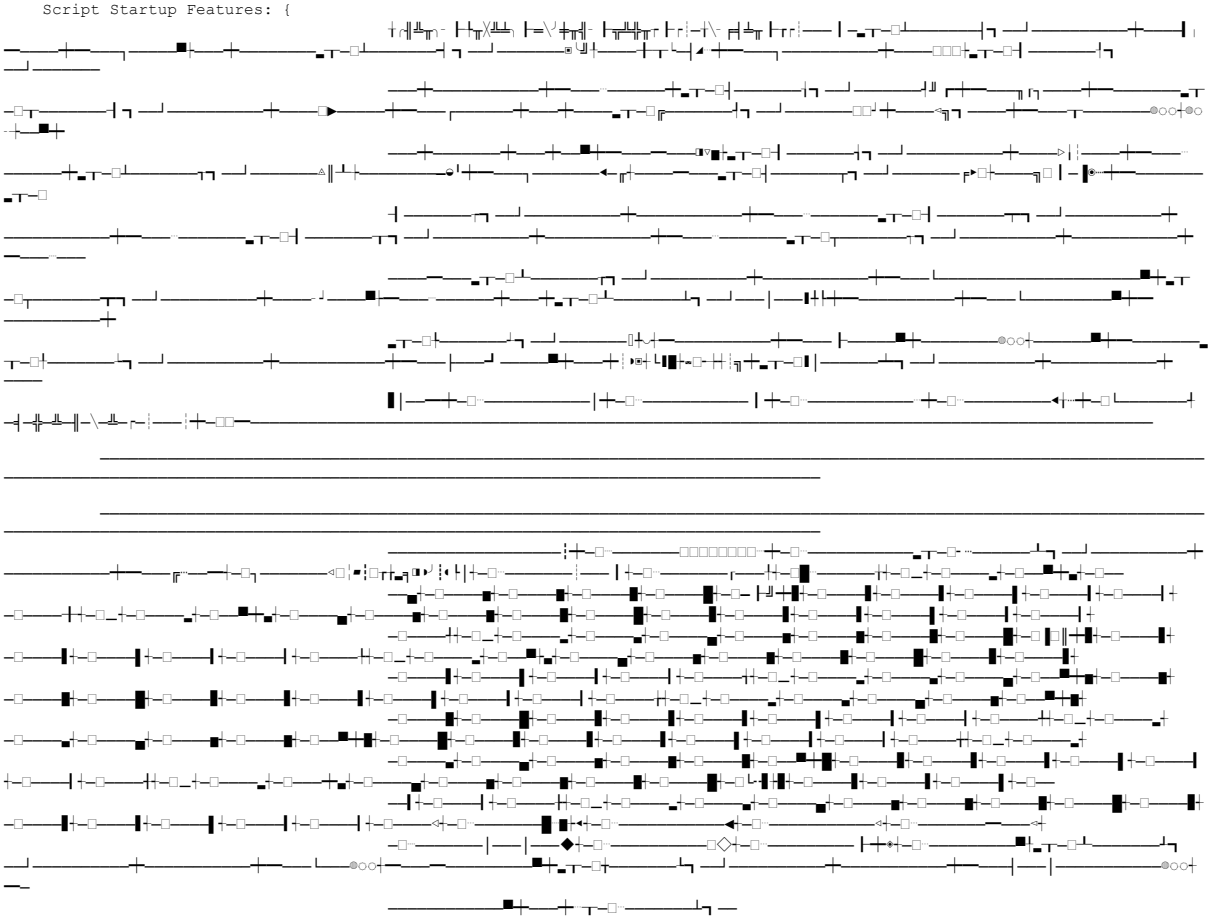
When the script playback hits the </StrokeEvent> end of block it completes the stroke.

Sample ArtRage Script Code Listings

Here are a couple of sample listings of ArtRage script files. If you want to try them out, open Notepad or TextEdit. Copy the listing text and paste into the text editor. Save the text file as Unicode. Then you can load the script in ArtRage.

Listing 1: 100 Flowers

```
//============================================================================
//============================================================================
//                            ArtRage Script File.
//============================================================================
//============================================================================


//============================================================================
// Version Block - Script version and ArtRage version:
//============================================================================

<Version>
    ArtRage Version: ArtRage 3 Studio Pro
    ArtRage Build: 3.0.8
    Professional Edition: Yes
    Script Version: 1
</Version>


//============================================================================
// Header block - Info about the painting/person who generated this script:
//============================================================================

<Header>
    // === Project data
    Painting Name: "Untitled"
    Painting Width: 1280
    Painting Height: 1024
    Painting DPI: 72
    // === Author data
    Author Name: "Andy Bearsley"
    Script Name: "100 Flowers"
    Comment: "How to paint 100 flowers"
    Script Type: ""
    Script Feature Flags: 0x000000001
</Header>


//============================================================================
// ArtRage project features. Sets the startup state of the script:
//============================================================================

<StartupFeatures>
    Script Startup Features: {
```



```
    }    // End of Script startup feature binary data.
</StartupFeatures>


//============================================================================
// Script data follows:
```

```
//============================================================================

<Events>

void PaintOneFlower() {                 // Added this line in a text editor.

Wait: 0.000s        EvType: Command     CommandID: SetForeColour      ParamType: Pixel    Value: { 0x0FFB9A5E3 }
SetColourHLS(Random(), ColourL(), ColourS());               // Added this line in a text editor.
<StrokeEvent>
          <StrokeHeader>
                    <EventPt> Wait: 1.200s        Loc: (525, 672)     Pr: 0.164384        Ti: 0.633333        Ro: 0.333333
          Rv: NO    Iv: NO    </EventPt>
                    <Recorded>          Yes       </Recorded>
                    <Smooth>  Count:    3
                              Loc: (525, 672)     Pr: 0     Ti: 1     Ro: 0
                              Loc: (524.499, 671.501)       Pr: 0     Ti: 1     Ro: 0
                              Loc: (523.997, 671.003)       Pr: 0     Ti: 1     Ro: 0
                    </Smooth>
                    <PrevA>   Loc: (-158.671, 253.411)      Pr: 0.243437        Ti: 1     Ro: 0     </PrevA>
                    <PrevB>   Loc: (-142.082, 236.822)      Pr: 0.391875        Ti: 1     Ro: 0     </PrevB>
                    <OldHd>   Loc: (524, 672)     Pr: 0     Ti: 0.633333        Ro: 0.333333        Dr: (0.108273, 0.994121)
          Hd: (-0.994121, 0.108273)     </OldHd>
                    <NewHd>   Loc: (525, 672)     Pr: 0     Ti: 0.644444        Ro: 0.330556        Dr: (0.204429, 0.978881)
          Hd: (-0.978881, 0.204429)     </NewHd>
          </StrokeHeader>
          Wait: 0.000s    Loc: (524, 671)    Pr: 0.41683    Ti: 0.644444        Ro: 0.341667        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (523, 670)    Pr: 0.48728    Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (522, 669)    Pr: 0.503914   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (521, 665)    Pr: 0.561644   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (520, 663)    Pr: 0.585127   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.017s    Loc: (518, 658)    Pr: 0.610568   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (517, 655)    Pr: 0.632094   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (517, 653)    Pr: 0.634051   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (517, 649)    Pr: 0.635029   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (517, 646)    Pr: 0.637965   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (517, 643)    Pr: 0.6409     Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.016s    Loc: (517, 637)    Pr: 0.66047    Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (518, 634)    Pr: 0.678082   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (518, 630)    Pr: 0.683953   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (518, 625)    Pr: 0.68591    Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (519, 623)    Pr: 0.68591    Ti: 0.622222        Ro: 0.341667        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (520, 619)    Pr: 0.686888   Ti: 0.622222        Ro: 0.341667        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (522, 616)    Pr: 0.68591    Ti: 0.622222        Ro: 0.341667        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (524, 613)    Pr: 0.68591    Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (526, 611)    Pr: 0.68591    Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (527, 610)    Pr: 0.68591    Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (529, 610)    Pr: 0.679061   Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (531, 611)    Pr: 0.679061   Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (533, 613)    Pr: 0.680039   Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (535, 615)    Pr: 0.680039   Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (536, 619)    Pr: 0.678082   Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (536, 621)    Pr: 0.680039   Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (537, 624)    Pr: 0.682975   Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (537, 626)    Pr: 0.684932   Ti: 0.622222        Ro: 0.341667        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (537, 629)    Pr: 0.688845   Ti: 0.622222        Ro: 0.341667        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (536, 630)    Pr: 0.691781   Ti: 0.622222        Ro: 0.341667        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (535, 633)    Pr: 0.692759   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (534, 636)    Pr: 0.693738   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (533, 637)    Pr: 0.692759   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (532, 639)    Pr: 0.692759   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (531, 641)    Pr: 0.692759   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (531, 644)    Pr: 0.689824   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (531, 648)    Pr: 0.682975   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (531, 651)    Pr: 0.669276   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (531, 653)    Pr: 0.643836   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (531, 657)    Pr: 0.445205   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.016s    Loc: (532, 658)    Pr: 0          Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
</StrokeEvent>
<StrokeEvent>
          <StrokeHeader>
                    <EventPt> Wait: 0.638s    Loc: (533, 673)    Pr: 0.167319   Ti: 0.633333        Ro: 0.338889        Rv:
NO        Iv: NO    </EventPt>
                    <Recorded>          Yes       </Recorded>
                    <Smooth>  Count:    3
                              Loc: (533, 673)     Pr: 0     Ti: 1     Ro: 0
                              Loc: (532.687, 672.689)     Pr: 0       Ti: 1     Ro: 0
                              Loc: (532.375, 672.378)     Pr: 0       Ti: 1     Ro: 0
                    </Smooth>
                    <PrevA>   Loc: (531, 657)     Pr: 0.445205        Ti: 0.622222        Ro: 0.347222        </PrevA>
                    <PrevB>   Loc: (531, 653)     Pr: 0.643836        Ti: 0.622222        Ro: 0.347222        </PrevB>
                    <OldHd>   Loc: (532, 673)     Pr: 0     Ti: 0.633333        Ro: 0.333333        Dr: (-0.974455, 0.224581)
          Hd: (-0.224581, -0.974455)    </OldHd>
                    <NewHd>   Loc: (533, 673)     Pr: 0     Ti: 0.633333        Ro: 0.333333        Dr: (-0.94756, 0.319578)
          Hd: (-0.319578, -0.94756)     </NewHd>
          </StrokeHeader>
          Wait: 0.000s    Loc: (533, 671)    Pr: 0.368885   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (534, 670)    Pr: 0.412916   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (535, 668)    Pr: 0.456947   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (539, 666)    Pr: 0.508806   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
          Wait: 0.019s    Loc: (541, 664)    Pr: 0.547945   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (542, 661)    Pr: 0.567515   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
          Wait: 0.016s    Loc: (547, 657)    Pr: 0.592955   Ti: 0.622222        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (548, 656)    Pr: 0.599804   Ti: 0.622222        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.010s    Loc: (553, 653)    Pr: 0.607632   Ti: 0.622222        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (554, 652)    Pr: 0.618395   Ti: 0.622222        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (558, 649)    Pr: 0.632094   Ti: 0.622222        Ro: 0.341667        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (559, 648)    Pr: 0.644814   Ti: 0.622222        Ro: 0.341667        Rv: NO    Iv: NO
          Wait: 0.014s    Loc: (564, 645)    Pr: 0.651663   Ti: 0.622222        Ro: 0.341667        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (566, 644)    Pr: 0.651663   Ti: 0.622222        Ro: 0.341667        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (569, 642)    Pr: 0.654599   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (572, 641)    Pr: 0.652642   Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
          Wait: 0.006s    Loc: (577, 639)    Pr: 0.65362    Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
```

```
        Wait: 0.018s    Loc: (580, 639)    Pr: 0.65362    Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (582, 638)    Pr: 0.65362    Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (584, 638)    Pr: 0.65362    Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (586, 638)    Pr: 0.652642   Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (588, 638)    Pr: 0.651663   Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (590, 639)    Pr: 0.650685   Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (591, 641)    Pr: 0.648728   Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (592, 642)    Pr: 0.649706   Ti: 0.633333        Ro: 0.352778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (592, 645)    Pr: 0.657534   Ti: 0.633333        Ro: 0.352778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (592, 647)    Pr: 0.66047    Ti: 0.633333        Ro: 0.352778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (591, 649)    Pr: 0.662427   Ti: 0.633333        Ro: 0.352778        Rv: NO    Iv: NO
        Wait: 0.036s    Loc: (589, 651)    Pr: 0.657534   Ti: 0.633333        Ro: 0.352778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (587, 653)    Pr: 0.654599   Ti: 0.633333        Ro: 0.352778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (585, 655)    Pr: 0.65362    Ti: 0.633333        Ro: 0.352778        Rv: NO    Iv: NO
        Wait: 0.036s    Loc: (582, 656)    Pr: 0.65362    Ti: 0.633333        Ro: 0.352778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (581, 657)    Pr: 0.654599   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (578, 659)    Pr: 0.649706   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (575, 660)    Pr: 0.638943   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (574, 661)    Pr: 0.636008   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (570, 661)    Pr: 0.626223   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (567, 662)    Pr: 0.624266   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (563, 663)    Pr: 0.624266   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
        Wait: 0.013s    Loc: (558, 665)    Pr: 0.622309   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (556, 665)    Pr: 0.610568   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (552, 668)    Pr: 0.600783   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (551, 669)    Pr: 0.546967   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
        Wait: 0.014s    Loc: (550, 672)    Pr: 0          Ti: 0.622222        Ro: 0.341667        Rv: NO    Iv: NO
</StrokeEvent>
<StrokeEvent>
        <StrokeHeader>
                <EventPt> Wait: 0.579s    Loc: (534, 675)    Pr: 0.143836   Ti: 0.622222        Ro: 0.336111        Rv:
NO      Iv: NO    </EventPt>
                <Recorded>          Yes       </Recorded>
                <Smooth>  Count:    3
                        Loc: (534, 675)    Pr: 0      Ti: 1     Ro: 0
                        Loc: (534.469, 674.548)    Pr: 0        Ti: 1     Ro: 0
                        Loc: (534.937, 674.096)    Pr: 0        Ti: 1     Ro: 0
                </Smooth>
                <PrevA>   Loc: (551, 669)    Pr: 0.546967         Ti: 0.633333        Ro: 0.344444        </PrevA>
                <PrevB>   Loc: (552, 668)    Pr: 0.600783         Ti: 0.633333        Ro: 0.344444        </PrevB>
                <OldHd>   Loc: (534, 674)    Pr: 0      Ti: 0.622222        Ro: 0.336111        Dr: (-0.897004, -
0.442023) Hd: (0.442023, -0.897004)     </OldHd>
                <NewHd>   Loc: (534, 675)    Pr: 0      Ti: 0.622222        Ro: 0.336111        Dr: (-0.951187, -
0.308615) Hd: (0.308615, -0.951187)     </NewHd>
        </StrokeHeader>
        Wait: 0.000s    Loc: (536, 675)    Pr: 0.531311   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (538, 675)    Pr: 0.543053   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
        Wait: 0.017s    Loc: (543, 677)    Pr: 0.561644   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (546, 678)    Pr: 0.582192   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (548, 678)    Pr: 0.597847   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (551, 680)    Pr: 0.618395   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (554, 680)    Pr: 0.637965   Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (557, 681)    Pr: 0.658513   Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
        Wait: 0.006s    Loc: (563, 682)    Pr: 0.669276   Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (566, 683)    Pr: 0.682975   Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (569, 684)    Pr: 0.694716   Ti: 0.644444        Ro: 0.341667        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (571, 685)    Pr: 0.699609   Ti: 0.644444        Ro: 0.341667        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (573, 687)    Pr: 0.710372   Ti: 0.644444        Ro: 0.336111        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (574, 689)    Pr: 0.712329   Ti: 0.644444        Ro: 0.336111        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (575, 691)    Pr: 0.714286   Ti: 0.644444        Ro: 0.333333        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (578, 695)    Pr: 0.715264   Ti: 0.644444        Ro: 0.333333        Rv: NO    Iv: NO
        Wait: 0.036s    Loc: (579, 698)    Pr: 0.717221   Ti: 0.655556        Ro: 0.327778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (579, 700)    Pr: 0.719178   Ti: 0.644444        Ro: 0.325 Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (579, 702)    Pr: 0.721135   Ti: 0.655556        Ro: 0.327778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (578, 704)    Pr: 0.720157   Ti: 0.655556        Ro: 0.327778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (577, 706)    Pr: 0.717221   Ti: 0.655556        Ro: 0.325 Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (574, 708)    Pr: 0.716243   Ti: 0.655556        Ro: 0.325 Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (572, 709)    Pr: 0.713307   Ti: 0.655556        Ro: 0.327778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (570, 709)    Pr: 0.712329   Ti: 0.655556        Ro: 0.327778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (567, 709)    Pr: 0.708415   Ti: 0.655556        Ro: 0.327778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (564, 709)    Pr: 0.707436   Ti: 0.655556        Ro: 0.327778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (560, 708)    Pr: 0.701566   Ti: 0.666667        Ro: 0.330556        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (558, 707)    Pr: 0.696673   Ti: 0.666667        Ro: 0.330556        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (556, 705)    Pr: 0.697652   Ti: 0.666667        Ro: 0.330556        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (555, 704)    Pr: 0.696673   Ti: 0.666667        Ro: 0.330556        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (554, 703)    Pr: 0.693738   Ti: 0.666667        Ro: 0.330556        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (553, 702)    Pr: 0.681996   Ti: 0.666667        Ro: 0.330556        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (552, 701)    Pr: 0.672211   Ti: 0.666667        Ro: 0.330556        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (551, 698)    Pr: 0.67319    Ti: 0.655556        Ro: 0.327778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (550, 695)    Pr: 0.675147   Ti: 0.655556        Ro: 0.327778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (549, 693)    Pr: 0.675147   Ti: 0.655556        Ro: 0.327778        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (548, 690)    Pr: 0.683953   Ti: 0.644444        Ro: 0.325 Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (545, 687)    Pr: 0.686888   Ti: 0.644444        Ro: 0.325 Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (543, 684)    Pr: 0.688845   Ti: 0.644444        Ro: 0.325 Rv: NO    Iv: NO
        Wait: 0.014s    Loc: (537, 681)    Pr: 0.68591    Ti: 0.644444        Ro: 0.330556        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (535, 680)    Pr: 0.646771   Ti: 0.644444        Ro: 0.330556        Rv: NO    Iv: NO
        Wait: 0.018s    Loc: (530, 680)    Pr: 0.417808   Ti: 0.622222        Ro: 0.330556        Rv: NO    Iv: NO
        Wait: 0.005s    Loc: (527, 680)    Pr: 0          Ti: 0.622222        Ro: 0.330556        Rv: NO    Iv: NO
</StrokeEvent>
<StrokeEvent>
        <StrokeHeader>
                <EventPt> Wait: 0.563s    Loc: (526, 684)    Pr: 0.186888   Ti: 0.622222        Ro: 0.330556        Rv:
NO      Iv: NO    </EventPt>
                <Recorded>          Yes       </Recorded>
                <Smooth>  Count:    3
                        Loc: (526, 684)    Pr: 0      Ti: 1     Ro: 0
                        Loc: (526.174, 684.842)    Pr: 0        Ti: 1     Ro: 0
                        Loc: (526.349, 685.683)    Pr: 0        Ti: 1     Ro: 0
                </Smooth>
                <PrevA>   Loc: (530, 680)    Pr: 0.417808         Ti: 0.622222        Ro: 0.330556        </PrevA>
                <PrevB>   Loc: (535, 680)    Pr: 0.646771         Ti: 0.644444        Ro: 0.330556        </PrevB>
```

```
                    <OldHd>   Loc: (526, 685)    Pr: 0      Ti: 0.622222        Ro: 0.336111        Dr: (-0.524954, -0.85113)
          Hd: (0.85113, -0.524954)      </OldHd>
                    <NewHd>   Loc: (526, 684)    Pr: 0      Ti: 0.622222        Ro: 0.330556        Dr: (-0.472142, -
0.881522) Hd: (0.881522, -0.472142)     </NewHd>
          </StrokeHeader>
          Wait: 0.000s    Loc: (525, 687)    Pr: 0.528376   Ti: 0.622222        Ro: 0.330556        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (525, 691)    Pr: 0.546967   Ti: 0.622222        Ro: 0.330556        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (524, 692)    Pr: 0.55773    Ti: 0.622222        Ro: 0.330556        Rv: NO    Iv: NO
          Wait: 0.009s    Loc: (523, 697)    Pr: 0.568493   Ti: 0.633333        Ro: 0.327778        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (522, 698)    Pr: 0.586106   Ti: 0.633333        Ro: 0.327778        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (521, 702)    Pr: 0.604697   Ti: 0.633333        Ro: 0.322222        Rv: NO    Iv: NO
          Wait: 0.016s    Loc: (520, 707)    Pr: 0.631115   Ti: 0.633333        Ro: 0.322222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (519, 708)    Pr: 0.644814   Ti: 0.633333        Ro: 0.322222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (517, 712)    Pr: 0.656556   Ti: 0.633333        Ro: 0.322222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (516, 715)    Pr: 0.681018   Ti: 0.644444        Ro: 0.319444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (513, 718)    Pr: 0.69863    Ti: 0.644444        Ro: 0.319444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (511, 720)    Pr: 0.706458   Ti: 0.644444        Ro: 0.319444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (510, 721)    Pr: 0.707436   Ti: 0.644444        Ro: 0.319444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (507, 722)    Pr: 0.708415   Ti: 0.644444        Ro: 0.319444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (505, 722)    Pr: 0.712329   Ti: 0.644444        Ro: 0.319444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (501, 723)    Pr: 0.713307   Ti: 0.644444        Ro: 0.319444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (498, 722)    Pr: 0.710372   Ti: 0.644444        Ro: 0.319444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (496, 720)    Pr: 0.71135    Ti: 0.644444        Ro: 0.319444        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (495, 717)    Pr: 0.687867   Ti: 0.644444        Ro: 0.319444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (495, 715)    Pr: 0.668297   Ti: 0.644444        Ro: 0.319444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (496, 713)    Pr: 0.666341   Ti: 0.644444        Ro: 0.319444        Rv: NO    Iv: NO
          Wait: 0.014s    Loc: (499, 707)    Pr: 0.667319   Ti: 0.644444        Ro: 0.319444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (501, 704)    Pr: 0.667319   Ti: 0.633333        Ro: 0.322222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (503, 702)    Pr: 0.668297   Ti: 0.633333        Ro: 0.322222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (506, 700)    Pr: 0.661448   Ti: 0.633333        Ro: 0.322222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (508, 698)    Pr: 0.65362    Ti: 0.633333        Ro: 0.322222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (510, 696)    Pr: 0.650685   Ti: 0.633333        Ro: 0.322222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (513, 694)    Pr: 0.649706   Ti: 0.633333        Ro: 0.322222        Rv: NO    Iv: NO
          Wait: 0.012s    Loc: (517, 689)    Pr: 0.649706   Ti: 0.633333        Ro: 0.322222        Rv: NO    Iv: NO
          Wait: 0.032s    Loc: (519, 684)    Pr: 0.652642   Ti: 0.633333        Ro: 0.322222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (520, 682)    Pr: 0.652642   Ti: 0.633333        Ro: 0.322222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (520, 680)    Pr: 0.659491   Ti: 0.633333        Ro: 0.327778        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (520, 678)    Pr: 0.666341   Ti: 0.633333        Ro: 0.327778        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (519, 677)    Pr: 0.126223   Ti: 0.622222        Ro: 0.330556        Rv: NO    Iv: NO
          Wait: 0.003s    Loc: (519, 677)    Pr: 0          Ti: 0.622222        Ro: 0.330556        Rv: NO    Iv: NO
</StrokeEvent>
<StrokeEvent>
          <StrokeHeader>
                    <EventPt> Wait: 0.427s    Loc: (518, 679)    Pr: 0.189824   Ti: 0.622222        Ro: 0.330556        Rv:
NO        Iv: NO    </EventPt>
                    <Recorded>          Yes       </Recorded>
                    <Smooth>  Count:    3
                              Loc: (518, 679)    Pr: 0      Ti: 1     Ro: 0
                              Loc: (518.406, 679.832)    Pr: 0        Ti: 1     Ro: 0
                              Loc: (518.813, 680.663)    Pr: 0        Ti: 1     Ro: 0
                    </Smooth>
                    <PrevA>   Loc: (519, 677)    Pr: 0.126223         Ti: 0.622222        Ro: 0.330556        </PrevA>
                    <PrevB>   Loc: (520, 678)    Pr: 0.666341         Ti: 0.633333        Ro: 0.327778        </PrevB>
                    <OldHd>   Loc: (518, 680)    Pr: 0      Ti: 0.633333        Ro: 0.327778        Dr: (-0.423475, -
0.905908) Hd: (0.905908, -0.423475)     </OldHd>
                    <NewHd>   Loc: (518, 679)    Pr: 0      Ti: 0.633333        Ro: 0.327778        Dr: (-0.426405, -
0.904532) Hd: (0.904532, -0.426405)     </NewHd>
          </StrokeHeader>
          Wait: 0.000s    Loc: (516, 679)    Pr: 0.370841   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (515, 680)    Pr: 0.422701   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (512, 682)    Pr: 0.54501    Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (511, 683)    Pr: 0.585127   Ti: 0.622222        Ro: 0.336111        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (509, 684)    Pr: 0.609589   Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.017s    Loc: (504, 686)    Pr: 0.642857   Ti: 0.633333        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.028s    Loc: (498, 688)    Pr: 0.679061   Ti: 0.633333        Ro: 0.333333        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (497, 689)    Pr: 0.697652   Ti: 0.633333        Ro: 0.333333        Rv: NO    Iv: NO
          Wait: 0.014s    Loc: (492, 690)    Pr: 0.712329   Ti: 0.633333        Ro: 0.333333        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (490, 690)    Pr: 0.720157   Ti: 0.644444        Ro: 0.330556        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (487, 690)    Pr: 0.734834   Ti: 0.644444        Ro: 0.330556        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (484, 690)    Pr: 0.752446   Ti: 0.644444        Ro: 0.330556        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (481, 690)    Pr: 0.754403   Ti: 0.644444        Ro: 0.333333        Rv: NO    Iv: NO
          Wait: 0.016s    Loc: (476, 688)    Pr: 0.757339   Ti: 0.644444        Ro: 0.333333        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (474, 686)    Pr: 0.75636    Ti: 0.644444        Ro: 0.333333        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (473, 684)    Pr: 0.741683   Ti: 0.644444        Ro: 0.336111        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (473, 681)    Pr: 0.74364    Ti: 0.644444        Ro: 0.336111        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (472, 678)    Pr: 0.742661   Ti: 0.644444        Ro: 0.341667        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (472, 675)    Pr: 0.74364    Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (472, 672)    Pr: 0.72407    Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (473, 671)    Pr: 0.721135   Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (477, 668)    Pr: 0.720157   Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (480, 667)    Pr: 0.719178   Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
          Wait: 0.024s    Loc: (486, 667)    Pr: 0.717221   Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (488, 668)    Pr: 0.717221   Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
          Wait: 0.014s    Loc: (495, 669)    Pr: 0.7182     Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (499, 670)    Pr: 0.7182     Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (504, 672)    Pr: 0.717221   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (506, 675)    Pr: 0.717221   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (508, 677)    Pr: 0.719178   Ti: 0.622222        Ro: 0.347222        Rv: NO    Iv: NO
          Wait: 0.016s    Loc: (515, 679)    Pr: 0.716243   Ti: 0.611111        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (518, 679)    Pr: 0.701566   Ti: 0.622222        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.014s    Loc: (528, 678)    Pr: 0.606654   Ti: 0.622222        Ro: 0.338889        Rv: NO    Iv: NO
          Wait: 0.017s    Loc: (531, 678)    Pr: 0          Ti: 0.622222        Ro: 0.338889        Rv: NO    Iv: NO
</StrokeEvent>
Wait: 0.039s        EvType: Command     CommandID: SetForeColour      ParamType: Pixel    Value: { 0x0FFC9E918 }
SetColourHLS(Random(), ColourL(), ColourS());               // Added this line in a text editor.
<StrokeEvent>
          <StrokeHeader>
                    <EventPt> Wait: 1.265s    Loc: (536, 667)    Pr: 0.212329   Ti: 0.633333        Ro: 0.344444        Rv:
NO        Iv: NO    </EventPt>
                    <Recorded>          Yes       </Recorded>
                    <Smooth>  Count:    3
                              Loc: (536, 667)    Pr: 0      Ti: 1     Ro: 0
```

```
                            Loc: (536.953, 667.047)    Pr: 0        Ti: 1     Ro: 0
                            Loc: (537.906, 667.094)    Pr: 0        Ti: 1     Ro: 0
                    </Smooth>
                    <PrevA>   Loc: (530, 751)    Pr: 0.251468         Ti: 0.622222        Ro: 0.336111        </PrevA>
                    <PrevB>   Loc: (523, 751)    Pr: 0.391389         Ti: 0.622222        Ro: 0.336111        </PrevB>
                    <OldHd>   Loc: (537, 667)    Pr: 0      Ti: 0.644444        Ro: 0.344444        Dr: (-0.0397409, -
0.99921)  Hd: (0.99921, -0.0397409)     </OldHd>
                    <NewHd>   Loc: (536, 667)    Pr: 0      Ti: 0.644444        Ro: 0.341667        Dr: (-0.26517, -0.964202)
          Hd: (0.964202, -0.26517)      </NewHd>
          </StrokeHeader>
          Wait: 0.000s    Loc: (535, 666)    Pr: 0.409002   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (533, 666)    Pr: 0.470646   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (532, 667)    Pr: 0.534247   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (530, 668)    Pr: 0.545988   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (528, 671)    Pr: 0.550881   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (527, 674)    Pr: 0.551859   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (527, 677)    Pr: 0.550881   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.019s    Loc: (529, 680)    Pr: 0.54501    Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (531, 681)    Pr: 0.509785   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (533, 681)    Pr: 0.479452   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.010s    Loc: (538, 680)    Pr: 0.484344   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.028s    Loc: (543, 677)    Pr: 0.488258   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (544, 676)    Pr: 0.490215   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (544, 674)    Pr: 0.539139   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (544, 672)    Pr: 0.554795   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (544, 670)    Pr: 0.564579   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.036s    Loc: (540, 667)    Pr: 0.624266   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.018s    Loc: (537, 667)    Pr: 0.631115   Ti: 0.633333        Ro: 0.344444        Rv: NO    Iv: NO
          Wait: 0.016s    Loc: (532, 670)    Pr: 0.556751   Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
          Wait: 0.013s    Loc: (532, 670)    Pr: 0          Ti: 0.633333        Ro: 0.35  Rv: NO    Iv: NO
</StrokeEvent>

}         // End of 'PaintOneFlower()' function.  Added this line in a text editor.

// Added the following code block with a text editor
Randomize()
for (int n = 1; n <= 100; n++) {
    MessageTip("Flower number %%n")

    // Offset the flower from where it was originally painted at Loc: (525, 672)
    real rOffsetX = Random(-525 + 100, PaintingWidth() - 525 - 100)   // Randomly move flower left/right half the painting.
    real rOffsetY = 4 * n - 672 + PaintingHeight() / 2                // Move flower down slightly as we go.
    SetTransOffset(rOffsetX, rOffsetY)
    PaintOneFlower()
}
```

Listing 2: Five Circles

```
// Function to draw a circle with the current tool and colour, and varying pressure.
void Circle(real rCX, real rCY, real rRad) {
    <StrokeEvent>

        // Prepare for the stroke.
        real x = rCX
        real y = rCY + rRad
        real pi = 3.14159265

        // Do the stroke header - Sets 'event location'
        <StrokeHeader>
            <EventPt>         Loc: (x, y)         Pr: 1     Ti: 1     Ro: 0     Rv: NO    Iv: NO    </EventPt>
            <Recorded>        No        </Recorded>
        </StrokeHeader>

        // Do the circle
        Loc: (x, y) Pr: 1     Ti: 1     Ro: 0     Rv: NO    Iv: NO         // MouseDown point

        // Body of circle.
        for (real s = 0; s <= pi * 2.05; s += pi / 100) {   // Slightly more than a circle, so paint end overlaps start
            x = rCX + sin(s) * rRad
            y = rCY + cos(s) * rRad
            real rPres = cos(s * 2) / 2 + 0.5
            Loc: (x, y)       Pr: rPres Ti: 1     Ro: 0     Rv: NO    Iv: NO
        }
        Loc: (x, y) Pr: 1     Ti: 1     Ro: 0     Rv: NO    Iv: NO        // MouseUp point.

    </StrokeEvent>
}

<Events>

// Program actually starts here.
int nCircleCount = 5
real rMaxRad = 500

real rRadStep = rMaxRad / nCircleCount

// Draw five circles, from smallest to biggest.
for (int n = 1; n <= nCircleCount; n++) {
          real rRad = rRadStep * n
          MessageTip("Circle number %%n at radius %%rRad")
          Circle(640, 512, rRad)
          Wait: 1.0s          // Admire our circle for a second.
}

</Events>
```

TABLE OF CONTENTS.

TABLE OF FIGURES.