introduction

This document is a description of and reference for the liquid5 games animation scripting language.

The purpose of the scripting language is to allow easy manipulation and animation of 3 dimensional objects in a graphics-engine independent context. The features of the scripting language allow for fps-independent (real-time rendered) as well as fps-dependent engines (or renderers).

reference

comments for source code documentation are handled in a C++ style and are preceded by " // "

blocks are handled in a C/C++ style:
  the start of a block is marked by { 
  the end of a block is marked by } 

statements are handled in a C/C++ style and are terminated by " ; "

breaks & exits from functions are handled in a C/C++ style using the "return" or "return value"
statements and are terminated by " ; "

scope is handled similar to C/C++: constants and variables that are defined globally can be accessed from anywhere in the script, while constants and variables that are defined locally in a block can only be accessed from inside that block.
**[liquid5] scripting language reference**

**constants:**

as the name suggests, constants are identifiers that are associated with a value which will remain unaltered during program execution. By convention constant identifiers in this scripting language are defined using capital letters only.

the following is a list of pre-defined constants, known to the virtual machine:

- **PI** - \( \pi \) - any explanation needed? X0000
- **BLACK** - colour value for black X0001
- **WHITE** - colour value for white X0002
- **RED** - colour value for red X0003
- **GREEN** - colour value for green X0004
- **BLUE** - colour value for blue X0005
- **VECI** - unity vector in positive x direction X0006
- **VECJ** - unity vector in positive y direction X0007
- **VECK** - unity vector in positive z direction X0008
- **VECN** - null vector X0009
- **TRUE** - boolean value 1 X000A
- **FALSE** - boolean value 0 X000B
- **NULL** - undefined, void X000C

more may be added during future revisions of this language...

Constants can be defined using the **constant** statement

```
constant identifier = value;
```

where constant is the reserved keyword, identifier is the name (by convention in capital letters) referring to the constant and value being the not changeable value contained within the constant.

**note:** the terminator ‘;’ is not part of the statement itself, but necessary to mark its end.

Constant definitions must be made at the top of the script if they are to be global, or at the top of a block, if they are to be valid within that block only. User-defined constants must be of a **scalar** type. Objects, fields and lists are not allowed. Although it is not an illegal operation to overload a pre-defined constant, it will generate a warning and should be avoided to guarantee correct functionality.
**engine state variables:**

these are variables that can be accessed for reading but not assigned a value. The values in these variables are updated automatically by the program running the virtual machine.

By convention engine state variable identifiers in this scripting language are defined using capital letters only.

- FPS - Frames Per Second - any explanation needed? X0000
- TIMECODE - time passed since the start of the script in 1/50s of a second X0001
- TIMER - time passed since the last reset of the timer in 1/50s of a second X0002
- SYSTIME - system time - field of four scalars: hours, minutes, seconds and 1/50s of a second X0003
- PROGTIME - time passed since program startup - field of four scalars: hours, minutes, seconds and 1/50s of a second X0004
- PALCODE - time passed since the start of the script in 1/25th of a second X0005
- NTSCCODE - time passed since the start of the script in 1/30s of a second X0006
- SVERSION - version number of the virtual machine (may come in handy) X0007
- EVERSION - version number of the game engine (may come in handy) X0008
- ERROR - last error code generated by the engine (for recovery) X0009
- WARNING - last warning generated by the engine (for handling) X000A
- FRAME - the current frame being rendered (may come in handy) X000B
- CAMERA - the index of the currently used camera X000C
- CAMERAS - the number of defined cameras X000D
- POLYS - the current number of displayed polygons X000E
- VERTS - the current number of displayed vertices X000F
- SCRW - the current screen width X0010
- SCRH - the current screen height X0011
- SCRD - the current screen colour depth X0012
- RENDERER - a string containing the name of the gfx api and its version number X0013
- OBJECTS - a list of all the objects that are loaded and set to be visible X0014

more may be added during future revisions...

at present there are no plans to allow for the use of user-defined engine state variables.
error codes & warnings:
the codes themselves may change in subsequent versions due to reordering...
The following are the error codes which can be generated by the compiler

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>undefined</td>
</tr>
<tr>
<td>002</td>
<td>undefined</td>
</tr>
<tr>
<td>004</td>
<td>wrong identifier</td>
</tr>
<tr>
<td>006</td>
<td>unknown identifier</td>
</tr>
<tr>
<td>008</td>
<td>scalar value expected</td>
</tr>
<tr>
<td>010</td>
<td>; expected</td>
</tr>
<tr>
<td>012</td>
<td>, expected</td>
</tr>
<tr>
<td>014</td>
<td>comparative operator expected</td>
</tr>
<tr>
<td>016</td>
<td>undefined</td>
</tr>
<tr>
<td>018</td>
<td>undefined</td>
</tr>
<tr>
<td>020</td>
<td>( expected</td>
</tr>
<tr>
<td>022</td>
<td>{ expected</td>
</tr>
<tr>
<td>024</td>
<td>[ expected</td>
</tr>
<tr>
<td>026</td>
<td>undefined</td>
</tr>
<tr>
<td>028</td>
<td>undefined</td>
</tr>
<tr>
<td>030</td>
<td>$ expected</td>
</tr>
<tr>
<td>032</td>
<td>% expected</td>
</tr>
<tr>
<td>034</td>
<td>do expected</td>
</tr>
<tr>
<td>036</td>
<td>of or = expected</td>
</tr>
<tr>
<td>038</td>
<td>while expected</td>
</tr>
<tr>
<td>040</td>
<td>string too long</td>
</tr>
<tr>
<td>042</td>
<td>undefined</td>
</tr>
<tr>
<td>044</td>
<td>undefined</td>
</tr>
<tr>
<td>046</td>
<td>too few trigger handlers</td>
</tr>
<tr>
<td>048</td>
<td>undefined</td>
</tr>
<tr>
<td>001</td>
<td>undefined</td>
</tr>
<tr>
<td>003</td>
<td>wrong type</td>
</tr>
<tr>
<td>005</td>
<td>identifier expected</td>
</tr>
<tr>
<td>007</td>
<td>number expected</td>
</tr>
<tr>
<td>009</td>
<td>string expected</td>
</tr>
<tr>
<td>011</td>
<td>: expected</td>
</tr>
<tr>
<td>013</td>
<td>= expected</td>
</tr>
<tr>
<td>015</td>
<td>assignment operator expected</td>
</tr>
<tr>
<td>017</td>
<td>undefined</td>
</tr>
<tr>
<td>019</td>
<td>undefined</td>
</tr>
<tr>
<td>021</td>
<td>) expected</td>
</tr>
<tr>
<td>023</td>
<td>} expected</td>
</tr>
<tr>
<td>025</td>
<td>] expected</td>
</tr>
<tr>
<td>027</td>
<td>undefined</td>
</tr>
<tr>
<td>029</td>
<td>LVal instead of RVal</td>
</tr>
<tr>
<td>031</td>
<td>£ expected</td>
</tr>
<tr>
<td>033</td>
<td>@ expected</td>
</tr>
<tr>
<td>035</td>
<td>of expected</td>
</tr>
<tr>
<td>037</td>
<td>for expected</td>
</tr>
<tr>
<td>039</td>
<td>until expected</td>
</tr>
<tr>
<td>041</td>
<td>undefined</td>
</tr>
<tr>
<td>043</td>
<td>undefined</td>
</tr>
<tr>
<td>045</td>
<td>trigger handler expected</td>
</tr>
<tr>
<td>047</td>
<td>main function expected</td>
</tr>
<tr>
<td>049</td>
<td>undefined</td>
</tr>
</tbody>
</table>
The following are the error codes which can be generated by the virtual machine. Some of these errors should never happen though. If they happened this would suggest a serious fault in the implementation of the virtual machine.

**error codes**

0   no error so far  
001  file not found  
002  could not load object  
003  could not copy object  
004  could not reference object  
005  constant undefined  
006  engine state variable undefined  
007  stack overflow  
008  trigger undefined  
009  infinite recursion  
010  command unsupported  
011  library unknown  
012  library file not found  
013  library loading failed  
014  ...
The following are the warnings which can be generated by the virtual machine. Some of these should never happen though. If they happened this would suggest a serious fault in the implementation of the virtual machine.

## warnings

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no warning so far</td>
</tr>
<tr>
<td>001</td>
<td>no camera defined</td>
</tr>
<tr>
<td>002</td>
<td>no camera active</td>
</tr>
<tr>
<td>003</td>
<td>no object loaded</td>
</tr>
<tr>
<td>004</td>
<td>invalid user input</td>
</tr>
<tr>
<td>005</td>
<td>...</td>
</tr>
<tr>
<td>020</td>
<td>infinite loop</td>
</tr>
<tr>
<td>021</td>
<td>...</td>
</tr>
<tr>
<td>030</td>
<td>morphing objects have different number of vertices</td>
</tr>
<tr>
<td>031</td>
<td>...</td>
</tr>
</tbody>
</table>
variable datatypes:

**scalar** - identifiers representing any number or character [string]
when declared, a scalar has the value NULL by default

    scalars are always preceded by the $ symbol

```
| scalar $x;
| scalar $x=1;
| scalar $x=0.5;
| scalar $x="Hello World";
| scalar $y, $z;
```

**object** - identifiers representing any graphical or geometrical data
(including particle emitters, textures etc.)

when declared, an object has the value NULL by default

    objects are always preceded by the £ symbol

```
| object £x;
| object £x="l5mesh.l5o";
| object £y, £z;
`````
extended datatypes:

**field** - covering arrays, enumerated datatypes and records/structures

if not explicitly declared otherwise, fields are of the datatype scalar by default when declared

enumeration of field elements start with the scalar value 0 (zero)

individual field elements can be accessed with the [] operator

fields are always preceded by the % symbol

```
field %x={first, second, third};
field %x of 3 scalar;
```

**list** - a dynamic array

lists are always preceded by the @ symbol

```
list @x;
list @x, @y;
```
**implicit abstract datatypes:**

**vector** - a field of always three scalars - handled like fields

the scalars are enumerated as \( x \), \( y \) and \( z \)

vectors cannot be pre-initialised

---

```
| vector %v;
| vector %v,%w;
```

**colour** - a field of always four scalars - handled like fields

the scalars are enumerated as \( r \), \( g \), \( b \) and \( a \) with \( a \) containing the value 1 by default

colours cannot be pre-initialised

---

```
| colour %c;
| colour %c,%d;
```

**string** - a scalar which is always handled as a character string

---

```
| string $x;
| string $x,$y;
| string $x="Hello World";
| string $x="Hello",$y="World";
```
[liquid5] scripting language reference

**camera** - an object which is only associated with object coordinates
but not with any other object geometry (mesh).
by default one camera is always defined.
the identifier for this default camera is `viewport`
except for `viewport` no camera can be pre-initialised

```
-----------
| camera $x;
| camera $x,$y;
|
-----------
```

**boolean** - a scalar which can only hold the values 0 (zero) and 1 (one)
by default booleans are always pre-initialised with the value 0.

```
-----------
| boolean $x;
| boolean $x,$y;
|
-----------
```

**group** - a virtual object which represents a number of other, already existing objects
a group allows more than 1 object to be collectively processed as 1
if possible an object should not be part of more than 1 group
if possible it should be avoided to create groups containing groups

```
-----------
| group $x=(object1,object2,object3);
|
-----------
```
operators:

+      -  addition
-      -  subtraction
*      -  multiplication
/      -  division
^      -  power
\     -  mod (modulo)
=      -  value assignment
=      -  comparison - equals
#      -  comparison - not equals
<      -  comparison - smaller than
<=     -  comparison - smaller than or equals
>      -  comparison - larger than
>=     -  comparison - larger than or equals
|     -  logical OR
&     -  logical AND
!     -  logical negation
-      -  arithmetic negation
~      -  sizeof/length
++     -  increment
--     -  decrement
+=     -  assign and increment by rval
-=     -  assign and decrement by rval
*+=    -  assign and multiply with rval
/=/    -  assign and divide by rval
.     -  child object access
[ ]    -  field element access
[liquid5] scripting language reference

libraries:
user defined libraries will not be implemented in the current version of the scripting language, however in later versions it will be possible to include libraries using the use statement:

use ascii_filename;

note: the terminator ';' is not part of the statement itself, but necessary to mark its end.

reserved words:

<table>
<thead>
<tr>
<th>boolean</th>
<th>camera</th>
<th>colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>constant</td>
<td>do</td>
</tr>
<tr>
<td>else</td>
<td>elsif</td>
<td>field</td>
</tr>
<tr>
<td>for</td>
<td>foreach</td>
<td>from</td>
</tr>
<tr>
<td>fromto</td>
<td>function</td>
<td>group</td>
</tr>
<tr>
<td>if</td>
<td>list</td>
<td>object</td>
</tr>
<tr>
<td>of</td>
<td>on</td>
<td>procedure</td>
</tr>
<tr>
<td>repeat</td>
<td>return</td>
<td>scalar</td>
</tr>
<tr>
<td>select</td>
<td>statement</td>
<td>string</td>
</tr>
<tr>
<td>this</td>
<td>trigger</td>
<td>until</td>
</tr>
<tr>
<td>use</td>
<td>vector</td>
<td>while</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**reserved identifiers:**

<table>
<thead>
<tr>
<th>abs</th>
<th>active</th>
<th>arot</th>
</tr>
</thead>
<tbody>
<tr>
<td>atan</td>
<td>atrns</td>
<td>close</td>
</tr>
<tr>
<td>console</td>
<td>cont</td>
<td>copy</td>
</tr>
<tr>
<td>cos</td>
<td>cross</td>
<td>deg</td>
</tr>
<tr>
<td>delete</td>
<td>dot</td>
<td>exec</td>
</tr>
<tr>
<td>follow</td>
<td>getaim</td>
<td>getgroup</td>
</tr>
<tr>
<td>getpos</td>
<td>gettop</td>
<td>getxpos</td>
</tr>
<tr>
<td>getypos</td>
<td>getzpos</td>
<td>hide</td>
</tr>
<tr>
<td>hidden</td>
<td>instance</td>
<td>isalpha</td>
</tr>
<tr>
<td>isbool</td>
<td>iscam</td>
<td>isgroup</td>
</tr>
<tr>
<td>ismorph</td>
<td>isnum</td>
<td>isreal</td>
</tr>
<tr>
<td>length</td>
<td>loadObj</td>
<td>lookat</td>
</tr>
<tr>
<td>max</td>
<td>min</td>
<td>morph</td>
</tr>
<tr>
<td>normal</td>
<td>pause</td>
<td>prterr</td>
</tr>
<tr>
<td>prtwn</td>
<td>rad</td>
<td>rand</td>
</tr>
<tr>
<td>ratan</td>
<td>rcos</td>
<td>read</td>
</tr>
<tr>
<td>regroup</td>
<td>rotgx</td>
<td>rotgy</td>
</tr>
<tr>
<td>rotgz</td>
<td>rotx</td>
<td>roty</td>
</tr>
<tr>
<td>rotz</td>
<td>rsin</td>
<td>rtan</td>
</tr>
<tr>
<td>setaim</td>
<td>setpos</td>
<td>settop</td>
</tr>
<tr>
<td>show</td>
<td>shutdown</td>
<td>sin</td>
</tr>
<tr>
<td>sqrt</td>
<td>tan</td>
<td>throw</td>
</tr>
<tr>
<td>toggle</td>
<td>trace</td>
<td>trnsx</td>
</tr>
<tr>
<td>trnsgy</td>
<td>trnsgz</td>
<td>trnsx</td>
</tr>
<tr>
<td>trnsy</td>
<td>trnsz</td>
<td>ungroup</td>
</tr>
<tr>
<td>unparent</td>
<td>viewport</td>
<td>visible</td>
</tr>
<tr>
<td>wait</td>
<td>write</td>
<td>zoom</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
control structures:

conditional statements:

simple if statement:

```plaintext
if (condition)
{
...
}
```

simple if statement with else branch:

```plaintext
if (condition)
{
...
}
else
{
...
}
```

complex if statement with alternative condition:

```plaintext
if (condition)
{
...
}
elseif (condition)
{
...
}
```
complex if statement with alternative condition and else branch:

```plaintext
if (condition) {
  ...
} elseif (condition) {
  ...
} else {
  ...
}
```

multi-coditional select statement:

```plaintext
select identifier from {
  scalar : ...;
  ...
}
```

loops:

head-controlled while loop:

```plaintext
while (condition) {
  ...
}
```
foot-controlled **while** loop:

```plaintext
do
{
...
}**while**(condition);
```

**note**: the terminator `;` is not part of the statement itself, but necessary to mark its end.

foot-controlled **repeat** loop:

```plaintext
repeat
{
...
}**until**(condition);
```

**note**: the terminator `;` is not part of the statement itself, but necessary to mark its end.

head-controlled **for** loop:

```plaintext
**for**(expression; condition; expression)
{
...
}
```

where the first expression should be used for initialising the loop index

and the second expression should be used to manipulate the loop index

**note** - foreach (with **this**) and fromto loops are reserved for future expansion

```plaintext
foreach(list|field)
```

```plaintext
fromto(start_timecode; end_timecode)
```
triggers:

A 'trigger' is an event occurring in the main engine.
The general syntax for triggers (event handlers) is:

```plaintext
on trigger do
{
  ...
}
```

where **on** and **do** are reserved keywords and
**trigger** is the user-defined or pre-defined event

At the end of a trigger call program flow automatically resumes from the statement before which the event occurred. The trigger block is exited automatically.

However an earlier exit from the trigger (event handler) body can be achieved using the *return* statement.

```plaintext
return;
```

**note:** the terminator `;` is not part of the statement itself, but necessary to mark its end.

All pre-defined functions can be called from within a trigger without restrictions

The *only* user-defined functions that may be called from within a trigger are

**statements** (due to the architecture of the virtual machine).

The triggers which are pre-defined in the virtual machine are:

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>for checking if the user has made any input (disabled if console is active)</td>
<td>X0000</td>
</tr>
<tr>
<td>keypress</td>
<td>for checking if the user has used the keyboard (disabled if console is active)</td>
<td>X0001</td>
</tr>
<tr>
<td>mouse</td>
<td>for checking if the user has used the mouse (disabled if console is active)</td>
<td>X0002</td>
</tr>
<tr>
<td>error</td>
<td>useful for error recovery - will handle all errors or specific errors (if specified)</td>
<td>X0003</td>
</tr>
<tr>
<td>error(error_number)</td>
<td></td>
<td>X0004</td>
</tr>
</tbody>
</table>
exception(exception_number) for exception handling - will handle all exceptions or specific exceptions

warning(warning_number) will handle all warnings or specific warnings (if specified).

collision(object) for collision detection in general, collisions with a specified object,
collision(object,object) or between two specified objects

timer(scalar) for checking if a specified point in time has been reached by the engine

terminate this is a special trigger. it is triggered by game engine shutdown and doesn't start an event handler but instead signals the virtual machine to clear all active events and end all wait states.

triggers can be defined using the trigger statement:

trigger(identifier,condition);

where trigger is the reserved keyword,
identifier is the name referring to the event handler and condition is the condition which will trigger the event

note: the terminator ';' is not part of the statement itself, but necessary to mark its end.

note: user-defined triggers cannot have parameters.

Trigger definitions must be made at the top of the script. The scope will always be the whole script. Every user-defined trigger must have a trigger handler. It is an illegal operation to overload a pre-defined trigger.
functions:

Commands for the virtual machine in the liquid5 animation scripting language are distinguished as being a member of one of the following 3 categories:

1. functions
2. procedures
3. statements

functions have a return value and can have parameters;
procedures have parameters, but no return value;
statements have neither parameters nor a return value

Although it is not an illegal operation to overload a pre-defined constant, it will generate a warning and should be avoided to guarantee correct functionality.

Function parameters are always passed by reference, i.e. that any changes which are applied to a parameter variable inside a function, will automatically also be applied to the variable which is passed into the function as a parameter.

The only way to prevent this from happening is to set the const(ant) qualifier for the parameter or by explicitly calling the function with a literal instead of a variable as parameter, which implies the const qualifier. This qualifier must explicitly be set in the function definition.

functions can be defined using the function statement

\[
\text{function} \hspace{1em} \text{identifier}(\text{parameterlist}) \text{ of datatype}
\]

where function is the reserved keyword,
identifier is the name referring to the function,
parameterlist is the list of parameters (see below) and
datatype is the function return type

\textbf{note:} the declaration has to be immediately followed by the function definition.
procedures can be defined using the procedure statement

```plaintext
procedure identifier(parameterlist)
```

where procedure is the reserved keyword, identifier is the name referring to the procedure and parameterlist is the list of parameters (see below)

**note:** the declaration has to be immediately followed by the procedure definition.

statements can be defined using the statement statement

```plaintext
statement identifier
```

where statement is the reserved keyword and identifier is the name referring to the statement

**note:** the declaration has to be immediately followed by the statement definition.

Parameterlists are a list of parameters, separated by commas (',') with which the function/procedure can be called. Each parameter consists of the parameter's datatype and the identifier by which it will be referred to from within the function/procedure and the optional keyword `const` to mark a parameter which may not be changed:

```plaintext
[const] datatype identifier[, ...]
```

At the end of a function/procedure/statement program flow automatically resumes after the function call. The function/procedure/statement is exited automatically. However an earlier exit from the function/procedure/statement body can be achieved using the return statement.

The last statement in a function should be a return statement followed by the return value of the function which should be an identifier of the same datatype as the datatype defined as the return datatype for the function.

If the return statement or the return datatype are missing at the end of a function, `NULL` will be returned.
return;

or

return identifier;

**note:** the terminator ';' is not part of the statement itself, but necessary to mark its end.
pre-defined function groups:

mathematical/geometrical functions:

scalar = rad(scalar) - converting a value from degrees to radians X0000
scalar = deg(scalar) - converting a value from radians to degrees X0001
scalar = cos(scalar) - calculate the cosine of an angle (degrees) X0002
scalar = sin(scalar) - calculate the sine of an angle (degrees) X0003
scalar = tan(scalar) - calculate the tangent of an angle (degrees) X0004
scalar = atan(scalar) - calculate the inverse tangent of an angle (degrees) X0005
scalar = rcos(scalar) - calculate the cosine of an angle (radians) X0006
scalar = rsin(scalar) - calculate the sine of an angle (radians) X0007
scalar = rtan(scalar) - calculate the tangent of an angle (radians) X0008
scalar = ratan(scalar) - calculate the inverse tangent of an angle (radians) X0009
scalar = sqrt(scalar) - returns the square root of the parameter X000A

numerical functions:

scalar = rand(scalar x) - returns a random number between 0 and x-1 X000B
scalar = min(scalar,scalar) - returns the smaller of the two parameters X000C
scalar = max(scalar,scalar) - returns the larger of the two parameters X000D
scalar = abs(scalar) - returns the absolute value of the parameter X000E

vector functions:

scalar = length(vector) - returns the length of the vector X000F
vector = normal(vector) - returns the unit vector of the parameter vector X0010
scalar = dot(vector,vector) - returns the dot product of the 2 parameter vectors X0011
vector = cross(vector,vector) - returns the cross product of the 2 parameter vectors X0012
object-management functions:

object = `loadObj`(string) - loads an object from the file specified by the string X0013

object = `copy`(object) - creates a copy of the specified object X0014

object = `instance`(object) - creates an instance of the specified object X0015

object = `morph`(object,object,scalar) - creates an object which is a morph of the first specified object by `scalar` percent into the second specified object X0016

object=`unparent`(child_object,object) - disconnects a child object from its parent and returns the child object X0017

boolean = `ungroup`(object,group) - removes an object from a group. returns `FALSE` if the object was not member of the group X0018

`regroup`(object,group) - adds an object to an existing group X0019

group = `getgroup`(object) - if the object is part of a group, the group identity is returned, otherwise `NULL` is returned if the object is part of more than one group, a group consisting of these groups is returned. X001A

`delete`(object) - removes the specified object from the scene X001B

`show`(object) - shows the specified object X001C

`hide`(object) - hides the specified object X001D

`toggle`(object) - shows / hides the specified object X001E

boolean = `hidden`(object) - checks if the specified object is known to the virtual machine X001F

boolean = `visible`(object) - checks if the specified object is positioned within the boundaries of the viewing frustum X0020

boolean = `ismorph`(object) - checks if the specified object is the result of morphing two other objects X0021

object placement and positioning functions:

`setpos`(object,scalar,scalar,scalar) - sets object position

`setpos`(object,vector) - sets object position

`setaim`(object,scalar,scalar,scalar) - sets object heading/direction (local z-axis) mainly useful for camera orientation

`setaim`(object,vector) - sets object heading/direction (local z-axis) mainly useful for camera orientation

`settop`(object,scalar,scalar,scalar) - sets object top direction
settop(object, vector)  
   define where "UP" (local y-axis) points to for an object (normal vector to heading)  
   mainly useful for camera orientation

**object sensory functions:**

**vector** = getpos(object)  
   - get object position

scalar = getxpos(object)  
   - get object global x-coordinate

scalar = getypos(object)  
   - get object global y-coordinate

scalar = getzpos(object)  
   - get object global z-coordinate

vector = getaim(object)  
   - get object heading/direction (local z-axis)

vector = gettop(object)  
   - get object local z-axis (normal vector to heading)

**object transformation functions:**

rotx(object, scalar)  
   - rotate an object scalar degrees about its local x-axis

roty(object, scalar)  
   - rotate an object scalar degrees about its local y-axis

rotz(object, scalar)  
   - rotate an object scalar degrees about its local z-axis

rotgx(object, scalar)  
   - rotate an object scalar degrees about the global x-axis

rotgy(object, scalar)  
   - rotate an object scalar degrees about the global y-axis

rotgz(object, scalar)  
   - rotate an object scalar degrees about the global z-axis

arot(object, vector, scalar)  
   - rotate an object scalar degrees about the vector

trnsx(object, scalar)  
   - translate an object scalar units along its local x-axis

trnsy(object, scalar)  
   - translate an object scalar units along its local y-axis

trnsz(object, scalar)  
   - translate an object scalar units along its local z-axis

trnsgx(object, scalar)  
   - translate an object scalar units along the global x-axis

trnsgy(object, scalar)  
   - translate an object scalar units along the global y-axis

trnsgz(object, scalar)  
   - translate an object scalar units along the global z-axis

atrans(object, vector)  
   - translate an object along vector by the length of vector units

atrans(object, vector, scalar)  
   - translate an object along vector by scalar units
Camera management functions:

**lookat**(object) - set the camera to focus on the specified object and to stay focused, no matter where the object may move. This is done until any other camera-specific function is called.

**follow**(object) - set the camera to focus on the specified object and to follow it in the current distance, no matter where the object may move. This is done until any other camera-specific function is called.

**active**(camera) - used to set the active camera. If it doesn't exist, the default camera will be used.

**zoom**(camera, scalar) - sets the zoom of the specified camera to the scalar zoom factor.

Engine management functions:

**console**(string) - brings up the console, displaying the specified welcome message.

**prterr**(scalar) - translates an error code into an understandable message and prints the message on the console.

**prtwrn**(scalar) - translates a warning code into an understandable message and prints the message on the console.

**write**(string) - prints a character string on the console.

**read**(scalar) - reads a maximum of characters specified by the scalar into the specified string.

**exec**(string) - tries to execute the specified string within the virtual machine and returns a boolean value to show success or failure.

**throw**(scalar) - throws an exception of the specified number/code.

**shutdown**(scalar) - ends execution of the game engine in the specified mode.

The following statements also belong into the engine management category:

**pause** - stop execution of the script and open the console. The event handler remains active while the program is paused and events will be processed if they are triggered.

**wait** - similar to **pause**, the execution of the script is stopped, but the console will not be opened. The event handler remains active while the program is waiting and events will be processed if they are triggered. Waiting stops if the terminate trigger is activated.
**[liquid5] scripting language reference**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cont</strong></td>
<td>continue execution of the script and close console</td>
<td>X004B</td>
</tr>
<tr>
<td><strong>trace</strong></td>
<td>continue execution of script protocolled with open console.</td>
<td>X004C</td>
</tr>
<tr>
<td></td>
<td>this command only works if the virtual machine is running in debug mode - otherwise it is ignored</td>
<td></td>
</tr>
<tr>
<td><strong>close</strong></td>
<td>close the console (and resume program execution if it was paused)</td>
<td>X004D</td>
</tr>
</tbody>
</table>

**variable management functions:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean = <strong>isalpha</strong>(scalar)</td>
<td>returns true if the specified scalar is a string</td>
<td>X004E</td>
</tr>
<tr>
<td>boolean = <strong>isnum</strong>(scalar)</td>
<td>returns true if the specified scalar is a number</td>
<td>X004F</td>
</tr>
<tr>
<td>boolean = <strong>isreal</strong>(scalar)</td>
<td>returns true if the specified scalar is a floating point number</td>
<td>X0050</td>
</tr>
<tr>
<td>boolean = <strong>isbool</strong>(scalar)</td>
<td>returns true if the specified scalar is a boolean</td>
<td>X0051</td>
</tr>
<tr>
<td>boolean = <strong>iscam</strong>(object)</td>
<td>returns true if the specified object is a camera</td>
<td>X0052</td>
</tr>
<tr>
<td>boolean = <strong>isgroup</strong>(object)</td>
<td>returns true if the specified object is an object group</td>
<td>X0053</td>
</tr>
</tbody>
</table>
program structure:

The program structure of a [liquid5] scripting language program is similar to the program structure of other high level languages like C/C++ or Pascal:

```
[global constant declarations]
| [global variable declarations]
| [function definitions]
| [trigger (event) declarations]
| [trigger (event) definitions]
| [main program block]
```

The block structure is very similar to the program structure itself:

```
[local constant declarations]
| [local variable declarations]
| [statements]
```

glossary:

- **array**: an array is a block of memory containing a number of variables of the same type
- **block**: a sequence of commands which is grouped as a single entity
- **camera**: the virtual camera is the viewport through which the computer generated scene is visible on screen, consisting of the camera's position in the virtual
environment, it's focus and zoom

compile-time anything that happens to the program while it is being compiled, the period during which a program is compiled

compiler a program which uses the source code of a program and translates it into binary code which can be understood by a computer or virtual machine

compiler error an error generated by the compiler during compile-time

compiler warning a warning generated by the compiler during compile-time

console the console is a simple user interface which allows commands to be typed in to the computer with the keyboard. The user can see the text that is being typed displayed in a reserved partition of the screen, the console window. Feedback is usually provided in the form of text displayed in the console window.

game the core program without any multi-media content attached to it

game state the current contents of the variables in the engine

error see compiler error, run-time error

error recovery the ability of the virtual machine to survive a "fatal" error without crashing, and to continue program execution

event a condition of/in the program, which does not have a pre-defined point of occurrence

event handler a block of code that is executed in reaction to the occurrence of an event

exception in the context of the scripting language an exception is a user-defined program error

function a function is a command for the virtual machine which takes one or more parameters and returns a value

library a library is a collection of commands which is stored in a separate module or file and only loaded and used for compilation of a program, if a program makes use of the commands from that library and explicitly requests the library
a literal is a set value, identified by a string or a character in the sourcecode. A literal has a datatype, but it is not a variable. As soon as a literal has been used for the first time, it has the constant value of itself:

```
a = 1 ;
```

\( a \) is a variable and \( 1 \) is a literal with the value 1.

the value on the left side of a function. e.g.: in \( x = a+b \) \( x \) would be the lval

a polygon mesh is the geometrical information (vertices) associated with an object

a vector which is perpendicular to a plane

objects in a game engine are the models that move around on the screen and make up the scene in the game. They usually have a position in the virtual environment of the game and have geometric and texture information which determines the shape and look of the model

a procedure is a command for the virtual machine which takes one or more parameters

a colour value stored in 4 channels for red, green, blue and alpha, where the alpha value stands for opacity

anything that happens in/to a program while it is executed

an error generated by the virtual machine during program execution

a warning generated by the virtual machine during program execution

the value on the right side of a function. e.g.: in \( x = a+b \) \( a+b \) would be the rval

the scope of a variable is the range within a programming language source file in which the variable is known to the program and can be validly used in that program

see engine state
**statement**

a statement is any command line in the source code of the scripting language which is terminated by a semicolon

in the context of the scripting language a *statement* is a command for the virtual machine which takes no parameters

**unary operator**

an operator which is 'attached' to a single value, which is changed by the operator

**unit vector**

a vector with the length 1 unit

**vector**

a line in space, which has a value (length) and a direction

**vertex**

a point in space which is used as a corner in one or more polygons

the *viewing frustum* is the space (volume) which is visible to the camera. any objects which are beyond its boundaries will be clipped and therefore not shown on screen

**virtual machine**

a computer with its own binary instruction set which does not exist physically but which is simulated by a software program

**warning**

see *compiler warning, run-time warning*