The Virtual Laboratory

CPFG
Reference Manual

Last updated: November 30, 2021

vlab was developed in the labs of Przemyslaw Prusinkiewicz at the University of Regina and the University of Calgary, Canada

Contents

1 Introduction

cpfg is a program for modeling plants and visualizing their development using the formalism of L-systems. It is assumed that the reader is familiar with the concepts of L-systems and turtle interpretation presented in *The Algorithmic Beauty of Plants* [1], as well as elements of the C programming language.

1.1 Installation

cpfg is distributed with *vlab*. Refer to the *vlab* documentation for installation instructions.

1.2 Running *cpfg*

The *cpfg* command line is defined as:

`cpfg [-a] [-C` *commstr*`] [-d] [-g] [-homo] [-rmode` *mode*`] [-v] [-V] [-w` *xsize ysize*`] [-wnb] [-wp` *xpos ypos*`] [-wpr` *x y*`] [-wr` *w h*`] [-wt` *wintitle*`] [-e` *environfile*`] [-gls` *glsfile*`] [-m` *mapfile*`] [-M` *matfile*`] [-ps` *psfile*`] [-ray` *rayfile*`] [-str` *textstrfile*`] [-strb` *binstrfile*`] [-vv` *vvfile*`]` *Lsystemfile viewfile* `[`*animationfile*`]`

Note that the options can be listed in any order, but the L-system, view, and animation files must be specified in order at the end of the command line. The L-system and view files are mandatory. For example, the following command line sets the size of the drawing window and its title, and specifies a colormap file:

```
cpfg -w 600 400 -wt Daisy -m daisy.map daisy.l daisy.v
```

1.2.1 Command line options

Parameter	Description
`-a`	Start *cpfg* in animation mode with the Animation menu (Section 1.3.3), rather than with the main menu. If *animationfile* is specified, it will be used to control the animation. Otherwise default values will be used (Section 8.2).
`-d`	Set program output to debug mode. Information is sent to *stdout*. This mode is intended only for code development. See `-v` and `-V` for warning and verbose modes.
`-g`	Perform off-screen rendering. A colormap or material file must be specified (using `-m` or `-M`). *cpfg* will generate the string up to the last derivation step as defined in *Lsystemfile*. The results can be stored in a specified output file (Section 1.2.2).
`-homo`	Output the string after applying homomorphisms (Section 4.3), using a specified output file (Section 1.2.2).
`-rmode` *mode*	Set the method for re-reading input files. The values of *mode* are: `expl` = explicit `cont` = continuous `trig` = triggered Refresh mode may also be set with the Refresh mode menu item (Section 1.3.2), and within a *vlab* object's specification file (see the *Vlab Framework* manual). If not set, the default is `explicit`.
`-v`	Set program output to warning mode. Warning and error messages (significantly less than `-V` mode) are sent to *stdout*.
`-V`	Set program output to verbose mode. Detailed warning and error messages are sent to *stdout*.

Parameter	Description
`-w` *xsize ysize*	Set the size of the drawing window, in pixels. For example, `-w 1024 683` will open a window suitable for saving image files with an aspect ration of 3:2, appropriate for video recordings.
`-wnb`	Open the drawing window without borders or title bar.
`-wp` *xpos ypos*	Specify the initial position of the top left corner of the window.
`-wpr` *x y*	Specify the relative window position: *x* and *y* are numbers between 0 and 1 giving the position of the top left corner of the drawing window relative to the top left corner of the screen.
`-wr` *w h*	Set the relative window size: *w* and *h* are numbers between 0 and 1 specifying the relative width and height of the drawing window with respect to the screen.
`-wt` *wintitle*	Change the title of the window to *wintitle*.

1.2.2 Optional files

The following input files are optional:

Parameter	Description
`-e` *filename*	Input parameters for environment communication. Also sets environment mode for communicating with an external program. See the *Vlab Environment Programs* manual for more information on environmental processes.
`-m` *filename*`.map` `-m`*n filename*`.map`	Input a colormap from a `.map` file. See the colormap editor (*palette*) in the *Vlab Tools* manual to interactively create a colormap file. Use `-m`*n* to specify more than one colormap file. The first file (`-m1`) will contain color indices 0-255, the second file (`-m2`) color indices 256-511, etc.
`-M` *filename*`.mat`	Input material records from a `.mat` file. Used instead of a colormap to improve the results of shading calculations. See the Materials editor (*medit*) in the *Vlab* Tools manual to interactively create a materials file.

One of the following file specifications should be used in batch mode (`-g`) and for output from animate mode (`-a`):

Parameter	Description

`-gls` *filename*`.gls`
`-ps` *filename*`.ps`
`-ray` *filename*`.ray`
`-str` *filename*`.str`
`-strb` *filename*`.strb`
`-vv` *filename*`.vv`
	In batch mode (`-g`), output the final result to the specified file.
	In animate mode (`-a`), or Animate on the pop-up menu), output each frame as it is generated with the Step and/or Run menu items (Section 1.3.3). The *filename* will be appended with the frame number, beginning with 000.
	Results can also be output interactively using the Save as and String > Output menu items (Section 1.3.2), as well as the Start recording menu item in animate mode (Section 1.3.3).

For *rayshade* output (`-ray`), the projection type must be set to `perspective` using the view file command `projection` (Section 8.1.2).

1.2.3 Main files

The following files are identified by their position at the end of the command line. The standard convention for each file extension is used here. They must be in this order:

Parameter	Description
Lsystemfile.l	Input the L-system. This file is mandatory.
viewfile.v	Input view parameters (Section 8.1). This file is mandatory, although many of the commands within the file have default values.
animationfile.a	Input the animation parameters (Section 8.2). This file is optional; default values will be used.

1.3 User Interface

1.3.1 View manipulation

The view in the drawing window is manipulated using the left and middle mouse buttons and the SHIFT key.

Action	Description
Rotate	Hold the left mouse button down. Rotate around the Y axis by moving the mouse horizontally, and around the X axis by moving the mouse vertically.
Pan	Hold the SHIFT key and left mouse button down. Move the mouse to pan in that direction.
Zoom	Hold the middle mouse button down. Zoom in by moving the mouse up, and zoom out by moving down.
Roll	Hold the SHIFT key and middle mouse button down. Roll clockwise around the Z axis by moving the mouse to the right, and roll counter-clockwise by moving the mouse to the left.

1.3.2 Main menu

A pop-up menu is displayed by clicking the right mouse button.

Menu item	Description
New model	Reread the L-system and view files, generate a new string, and interpret it to create a new image. The model is automatically centered in the window, or placed according to the commands in the view file (Section 8.1).
New L-system	Reread the L-system file, generate a new string and interpret it to create a new image without modifying the view.
New view	Reread the view file and re-interpret the existing string to create a new image. The model is automatically centered in the window, or placed according to the commands in the file.
New rendering	Reread the view file and update the image using the rendering commands in the file, but without changing the view (i.e. do not change the scale, camera, rotation, etc.).
Save	Save to the default file name and type. The initial file name is the same as the original L-system file, with a PNG extension. Use Save as to change the name and/or file type. The file used in the Save as window will then become the default file for this Save command.
Save as...	Open a dialog window to save the current view with a different name and/or in a different format. (See below for a description of the dialog window).

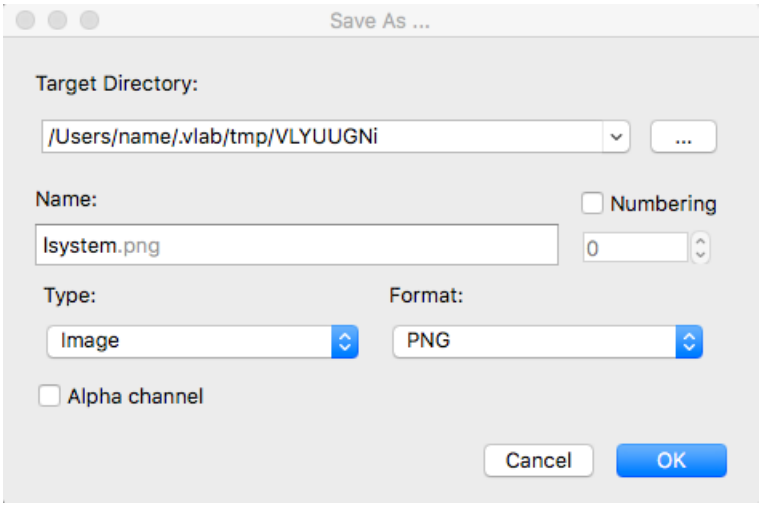


Figure 1: An example of the Save as... dialog window.

Menu item	Description
String	Input or Output the L-system string to either a Text file or a Binary file. This can be used to externally manipulate the string, and then re-read it.
Animate	Set *cpfg* to animate mode. The menu is updated with new items (Section 1.3.3).
Refresh mode	Set the mode for re-reading files and updating the object in the drawing window. The default is Explicit - the files are re-read by invoking one of the New... menu items above. Alternatively, Triggered/Continuous mode uses *cpfg*'s ability to receive messages from external programs (e.g. panels) to re-read a file as changes are made.
Exit	Exit *cpfg*.

The Save as menu item opens a dialog window such as the one in Figure 1. The fields in the window are:

Menu item	Description
Target Directory	The directory in which to save the file. The default is the *lab table* directory.
Name	The name of the file. The default is the same as the Save command. If this name is changed, it will become the default for subsequent Save and String commands. Note that the file extension cannot be edited; it is set based on the Type (and possibly the Format) field.
Numbering	Check this box to add a number to the file name. The number will be incremented automatically each time the file is saved. For example, if the file name in this dialog box is set to lsystem0000.png, subsequent Save commands will automatically save lsystem0001.png, lsystem0002.png, and so on. This can be used to save the frames of an animation.
Type	The file type. This will set the extension in the Name field for all type except Image.
Format	Set the extension when Type is Image. This field is ignored for other types.
Alpha channel	Check this box to make the background transparent in the saved file.

1.3.3 Animation menu

When Animate is selected from the menu, or the `-a` option is included on the command line (Section 1.2.1), the following items are added to the menu. The menu items use parameters from the animation file, or their default values (Section 8.2).

Menu item	Description	Shortcut
Step	Display the frame resulting from the next `step` derivation steps. If beyond the `last frame`, the `first frame` is displayed.	Ctrl+F
Run	Display consecutive animation frames after each `step` derivation steps until `last frame` is reached or passed.	Ctrl+R
Forever	Display consecutive animation frames after each `step` derivation steps. When `last frame` is reached, return to `first frame` and continue.	Ctrl+V
Stop	Pause the animation at the current frame.	Ctrl+S
Rewind	Redisplay the `first frame` of the animation.	Ctrl+W
Clear	Remove the current image from the window.	
New animate	Reread the animation file.	
Start recording	Initiate recording of animation frames. Use the Save as option before beginning the recording to set the file name and type and, most importantly, to set Numbering on. This menu item will change to Stop recording once it begins.	
Don't animate	(Replaces the Animate item on the main menu.). Quit animation mode and return to the main menu.	

2 The L-system file

An L-system can be defined over an arbitrary alphabet that does not contain the asterisk (*) or any separators (space, tab, etc.). Section 6 lists symbols that have a graphical interpretation.

A typical *cpfg* L-system file has the following format:

`#define` *statements*
`lsystem:` *label*
declarations
`derivation length:` *integer*
`axiom:` *list*
control statements
productions
`endlsystem`

The parameter *label* is optional. It is ignored except when using sub-L-systems (Section 7.2).

2.1 Axiom statement

The axiom defines the starting string and cannot be blank. It is composed of symbols with and without specific parameter values. For example:

```
axiom: I(DELAY)FA(1)
```

is an axiom with three symbols, `I`, `F` and `A`, two of which have a parameter.

2.2 Variables

Variable names are defined as in C, and are assumed to be of type `float`.

2.2.1 Local vs. global variables

Both local and global variables can be defined. Local variables are used within a production's programming statements (see Section 3.2). Global variables are defined in start and end blocks before the productions (see Section 4.1).

2.2.2 Arrays

Arrays in *cpfg* are defined in a single statement in the *declarations* section. The statement has the format:

`define: { array` *arrayname1*`[` *dimensions* `]; array` *arrayname2*`[` *dimensions* `]; ...}`

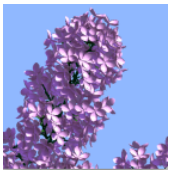
Each array is specified by its name, *arrayname*, and the size of each dimension, *dimensions*, separated by commas. All arrays are defined in a single `define:` statement, separated and ending with a semicolon. The specification may extend over several lines.

The arrays may also be initialized at the same time using the syntax:

`define: { array` *arrayname*`[` *dimensions* `] = {` *values* `}; }`

where *values* are also separated by commas. For example, the following statement defines three arrays, two of which are initialized:

```
define: { array GrowthRate[3,2] = {1, 0.9, 0.8, 1, 0.8, 0.6};
          array Angle[4] = {75, 60, 45, 30};
          array Nutrients[5]; }
```



See object:
Lilac

In a plant model, the value of a parameter frequently depends on the order, age, or vigor of the apex and each branch. It is possible to have separate productions for each branching order with different values in the successor. But in some cases it is more effective to define an array of values and use only one production.

3 Productions

A production in *cpfg* consists of three parts: the *predecessor* defines the string to be matched and the context in which it must be found; programming *statements* are evaluated once the predecessor is matched, and may include a condition that, if present, must evaluate to true; and the *successor* defines how the predecessor will change in the next derivation step.

The predecessor must be nonempty. The statements section is optional. An empty successor must be represented by an asterisk (*). The basic formats of a production are:

predecessor `-->` `*`
predecessor `-->` *successor*
predecessor `:` *statements* `-->` *successor*

3.1 The predecessor

The predecessor consists of three components: the *strict predecessor*, its *left context*, and its *right context*. It has the form:

lcontext `<` *strict-predecessor* `>` *rcontext*

Both the left and right context are optional (along with the corresponding separator), but the strict predecessor must be nonempty. Each component consists of symbols with or without parameters. The parameters are "formal": they are variables that represent actual parameter values. For example:

```
S(a,b) < A(i) > T
```

will match a symbol `A` that has a single parameter (of any value) and is preceded by a symbol `S` with any two parameters, and followed by a symbol `T` with no parameters.

3.2 Programming statements

A production may optionally include statements, which use C-like syntax. The statements are divided into three components formatted as:

{ *before-stmts* } *condition* { *after-stmts* }

where *condition* is a Boolean conditional expression. The production will only be used if *condition* evaluates to True.

Before-stmts are performed each time the predecessor (including the left and right context) match, before the *condition* is evaluated. Thus, these statements can be used to precompute expressions required in *condition*.

After-stmts are performed only after *condition* is evaluated as True, before the predecessor is replaced by the successor.

All components are optional. An empty *condition*, which is always true, is represented by an asterisk (*). Therefore, the general format of the statements can be:

{*before-stmts* } *condition* {*after-stmts* }
{*before-stmts* } * {*after-stmts* }
{*before-stmts* } *condition*
{*before-stmts* } *
condition {*after-stmts* }
* {*after-stmts* }

Three types of C statements may be used as part of the *before-stmts* and *after-stmts*:

- **Assignment statements** of the form:

  varname = *expression*;

  where *expression* is an arithmetic expression. The expression may include local variables assigned a value in previous assignment statements within the same production. For example:

  ```
  A(y) : { x=y/2; s=x*x; } s<10 --> B(x)
  ```

- **Conditional statements** have two forms:

  `if (` *condition* `) {` *statement*; `...` `}`
  `if (` *condition* `) {` *statement*; `...` `} else {` *statement*; `...` `}`

  where *condition* is a logical expression. Note that {} brackets are required even when there is only one *statement*. For example:

  ```
  A(y) : * { if ( y>10 ) { y=10; } } --> B(y)
  ```

- **Loop statements** also have two forms:

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

  As in conditional statements, {} brackets are required in loop statements even when there is only one *statement*.

However, compared to C syntax, the syntax of L-system programming statements must follow specific rules:

- Each assignment statement must terminate with a semicolon, even if it is a last statement in a block of statements (just before the '}').
- Statements following `if, else, while` and `do` must always be enclosed in curly brackets, even if there is only one statement.
- All parameters are assumed to have real (floating point) values.
- Operators such as ++, –, +=, -=, *=, and /= are not supported.

4 Control statements

4.1 Start and End blocks

There are four blocks of statements that define procedures to be executed at specific points in an L-system simulation:

- `Start` - called before the output string is initialized from the axiom
- `StartEach` - called before each derivation step
- `EndEach` - called after each derivation step
- `End` - called after the final derivation step

Each block has the syntax:

block name: { *statements* };

where *statements* have the same syntax as programming statements within a production (Section 3.2). The four blocks are added before the productions in the L-system file.

For example, to set a counter `i` that is incremented after each derivation step:

```
Start: { i=0; }
EndEach: { i=i+1; }
```

Variables used on the left side of an assignment statement in these blocks are considered global, and can be accessed within other blocks and within productions.

4.2 Ignore and Consider statements

Symbols can be ignored when context matching using the statement:

`ignore:`*symbols*

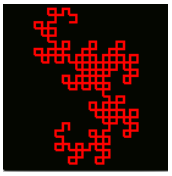
Alternatively, to consider only specific symbols when context matching, use the statement:

`consider:`*symbols*

In both cases, *symbols* are listed without parameters, and without separators. For example:

```
ignore: / + - G
```

4.3 Homomorphism



See object: DragonCurve

An L-system homomorphism is defined as a set of productions applied only for interpretation purposes (i.e. when drawing the object from the string). This allows the modeler to change the details of the appearance without modifying the underlying logic of the model captured by the main L-system productions.

In *cpfg*, the homomorphism is specified by productions that are placed at the end of the L-system, beginning with the keyword `homomorphism` and ending with the `endlsystem` statement.

Once the string has been derived using the main L-system productions and is ready for interpretation, each module in the string is matched to the homomorphism productions creating the string to be interpreted. If a match is found, the successor in the homomorphism production is used in the string. If no match is found, the original module is retained as the successor. Note that the resulting string is used for interpretation purposes only. The next derivation step will use the string produced by the main L-system productions, excluding the changes made by the homomorphism productions.

For example, a simple plant can be described in terms of apices (`A`) and internodes (`I`):

```
Axiom: A
A --> I[+A][-A]IA
I --> II
```

The turtle commands to draw the apices and internodes can then be described in a homomorphism, and changed as needed:

```
homomorphism
A --> ;F
I --> F
endlsystem
```

Homomorphism productions with parameters or programming statements operate similarly to L-system productions: the expressions are evaluated and replace the formal parameters.

4.3.1 Recursive homomorphism

It is possible to repeatedly apply homomorphism productions to the string to be interpreted. To enable this, the following statement can be added after `homomorphism` and before the productions:

`maximum depth:` *d*

where *d* is an integer representing the number of times the homomorphism productions are applied (to avoid infinite recursion). If `maximum depth` is omitted, the default is 1 - i.e. no recursion.

A warning may be issued if the maximum depth is reached and it is possible to further apply the homomorphism productions. To enable this, add `warnings` to the `homomorphism` statement:

```
homomorphism: warnings
```

No warnings are issued for the simple `homomorphism` statement. For backward compatibility, it is also possible to state:

```
homomorphism: no warnings
```

4.3.2 Context in homomorphism productions

The context for a homomorphism production is the context in the original L-system string; homomorphism successors do not affect the context search.

For example, the following homomorphism is used to draw only branches whose endpoint `P` has a `y` coordinate less than 3:

```
homomorphism
F > P(x,y) : y > 3 --> f
```

If the homomorphism in the previous example was defined as:

```
homomorphism
maximum depth: 2
F --> G P(0,0)
G > P(x,y) : y > 3 --> f
```

The context for the second homomorphism production would be a module `P` in the original L-system string (with its parameters), and not the module `P` introduced by the first homomorphism production.

4.3.3 Random numbers in homomorphism

The use of random values in a homomorphism is not recommended during an animation, since the values used in one simulation step would be different from values used in next step and visible discontinuities may result. The resulting structure may change after each redraw.

To prevent this, it is possible to define a separate random number generator used only with the homomorphism productions by including a `seed:` statement after the `homomorphism` statement and before the productions.

4.4 Decomposition

Decomposition productions make it possible to decompose a module in the string into several components. Thus the L-system productions can focus only on the development of main building blocks of a plant, such as an apex, meristem, or leaf.

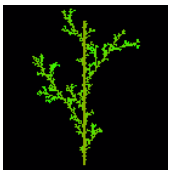
After each simulation step, before the string is interpreted (and a possible homomorphism is applied), modules representing these organs can be replaced by several other modules, representing parts of the organs. Unlike homomorphism productions, the results of decomposition stay in the string.

Decomposition productions must be specified after L-system productions and before homomorphism productions (or at the end of an L-system if no homomorphism productions are included).

The syntax of a decomposition block is similar to a homomorphism block, beginning with a `decomposition` statement, with or without a `warning`. There can also be a `maximum depth` statement. However, decomposition productions use the same random number generator as the L-system productions. Therefore, the `seed` statement cannot be used.

4.5 Stochastic L-systems

Stochastic L-systems require a seed for the random number generator. This is specified with the statement:



See object: RandomMoss

`seed:` *svalue*

where *svalue* is an integer number. This should be the first statement in the L-system directly after the `lsystem:` statement.

Productions have an added element as well:

predecessor `:` *statements* `-->` *successor* `:` *probability*

where *probability* is a percentage (1-99) representing the probability factor for that production. For example:

```
F --> F[+F]F : 40
F --> F[-F]F : 60
```

will create a branch to the left with 40% probability, and a branch to the right with 60% probability.

The seed is re-initialized to *svalue* when the New model menu item is selected, but not with the New L-system menu item. This means that a random new image is displayed with each call to New L-system.

5 Predefined Functions

The following predefined functions can be included in L-system expressions (e.g. to define a parameter value, or in a predecessor statement).

5.1 Mathematical functions

Function	Description
`sin`(α) `cos`(α) `tan`(α)	Return the `sin/cos/tan` of angle α, where α is in degrees
`asin`(x) `atan`(x)	Return an `asin/atan` value between -90° and +90°
`acos`(x)	Return an `acos` value between 0° and 90°
`atan`(x,y)	Return the arctangent of y/x in the range -90° to +90°
`exp`(x)	Return e^x.
`log`(x)	Return the log of x
`sqrt`(x)	Return the square root of x
`fabs`(x)	Return the absolute value of x
x^y	Return x^y

5.2 Rounding functions

Function	Description
`floor`(x)	Return the largest integer <= x
`Ceil`(x)	Return the smallest integer >= x
`trunc`(x)	Truncate x to an integer
`sign`(x)	Return 0 if $x = 0$ Return 1 if x is positive Return -1 if x is negative

5.3 Random number functions

Function	Description
`srand`(*seed*)	Initialize the random number generator used in the following functions.
`ran`(x)	Generate floating point values uniformly distributed in the interval (0, x).
`nran`(m, δ)	Generate a normal distribution of random numbers with mean m and standard deviation δ.
`bran`(α, β)	Return random values with β distribution.
`biran`(n,p)	Generate random numbers with a binomial distribution - i.e. how many out of n numbers are below p.

See object:
WeepingBirch

5.4 Input and output functions

Function	Description
`printf(` *"format-string", var1, var2, ...*`)`	Print variables to standard output using the specified format.
`fopen(` *"filename", "type"*`)`	Open the file *filename* for input (*type* = `r`) or output (*type* = `w`). Returns an identifier for the file, which is used in the functions below.
`fclose(`*file-id*`)`	Close the file with identifier *file-id*.
`fscan(`*file-id, "format-string", &var1, &var2, ...*`)`	Input data from the file with identifier *file-id*, using the specified format.
`fprintf(`*file-id, "format-string", var1, var2, ...*`)`	Output the specified variables to the file with identifier *file-id*, using the specified format.
`fflush(`*file-id*`)`	Flush the buffers of the file with identifier *file-id*.

Note that all variables are of type `float`. Therefore, the format strings should contain `%f` and `%g` only.

5.5 User-defined contours and functions

User-defined contours and functions are described in files (see the *cuspy* and *funcedit* tools in the *Vlab Tools* manual for the format of the files). The files are specified in the view file (Sections 8.1.5 and 8.1.6, respectively) and assigned an *id* that is used in the following functions.

Function	Description

`curveX(`*id,t*`)`
`curveY(`*id,t*`)`
`curveZ(`*id,t*`)`
	Return the *x, y,* or *z* coordinate, respectively, at coordinate t of the contour identified by *id*. The value of t should be in the range [0,1]. If $t < 0$, return the position of the start of the contour. If $t > 1$, return the position of the endpoint of the contour.

`curveGAL(`*id*`)`	Return the length of the contour identified by *id*.
`func(`*id, x*`)`	Return the value at x of the user-defined function with identifier *id*. The value of x should be in the range [0,1]. If $x < 0$, return the value of the function at 0. If $x > 1$, return the value of the function at 1.

5.6 Other functions

Function	Description

`vvXmin()`
`vvXmax()`
`vvYmin()`
`vvymax()`
`vvZmin()`
`vvZmax()`
	Return the minimum and maximum extents of the rendered volume, in *x, y,* and *z* coordinates.

`stop(`*n*`)`	Stop the animation. If n=1, and Run or Forever is selected from the menu, draw the current string and continue. Otherwise, stop the simulation.

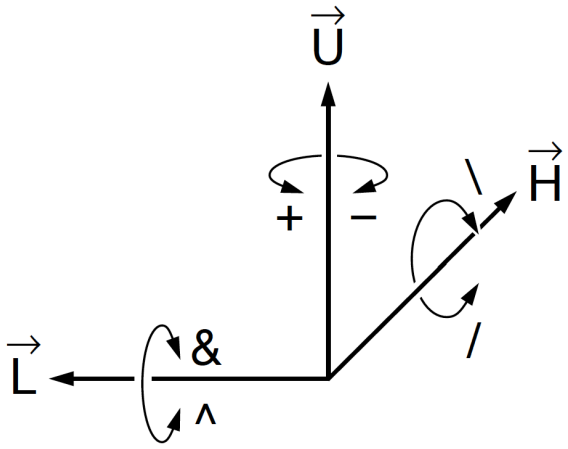


Figure 2: Controlling the turtle in three dimensions

6 Predefined modules

During visualization, the string of symbols is parsed from left to right. When a predefined symbol controlling the turtle is encountered, the function associated with the symbol is performed. Symbols with predefined interpretations are listed below. If a symbol has more parameters than those specified below, the additional parameters are ignored.

6.1 Position and drawing

In the following modules, the parameter l is optional. If not specified, the default is used.

Module	Description	Default
`F(`l`)`	Move forward a step of length l and draw a line segment from the original position to the new position of the turtle. If the polygon flag (Section 6.5) is on, the final position is recorded as a vertex of the current polygon.	1
`f(`l`)`	Move forward a step of length l without drawing a line. If the polygon flag (Section 6.5) is on, the final position is recorded as a vertex of the current polygon.	1
`G(`l`)`	Move forward a step of length l and draw a line.	1
`g(`l`)`	Move forward a step of length l without drawing a line.	1
`@M(`x,y,z`)`	Set the turtle position to (x,y,z).	

The view file command `line style` (Section 8.1.5) specifies whether the line is drawn as a line, polygon, or cylinder.

6.2 Turtle rotations

The turtle can be rotated around its heading, left, or up vector only (Figure 2).

Module	Description
`+(`θ`)`	Turn left by angle $\theta°$ around the U axis.
`-(`θ`)`	Turn right by angle $\theta°$ around the U axis.
`&(`θ`)`	Pitch down by angle $\theta°$ around the L axis.
`^ (`θ`)`	Pitch up by angle $\theta°$ around the L axis.

Module	Description
`\(`θ`)`	Roll left by angle θ° around the H axis.
`/(`θ`)`	Roll right by angle θ° around the H axis.
`|`	Turn 180° around the U axis. This is equivalent to `+(180)` or `-(180)`. It does not roll or pitch the turtle.
`@v`	Roll the turtle around the H axis such that H and U lie in a common vertical plane with U closest to up.

`@R(`*hx,hy,hz*`)`
 	
`@R(`*hx,hy,hz,ux,uy,uz*`)`
	Set the turtle heading to *(hx,hy,hz)*. If the vector is not normalized, it will be done automatically. If *(ux,uy,uz)* are not specified, the turtle up and left vectors are adjusted to minimize their rotation with respect to their previous orientation. Otherwise, *(ux,uy,uz)* specify the turtle up vector (which will also be normalized automatically), the left vector will be computed directly from the heading and up vectors.

If the parameter θ is not specified, the value of the view file command `angle increment`, or its default, will be used (Section 8.1.1).

Modules `@v` and `@R` adjust the turtle orientation with respect to absolute coordinates (as compared to other rotations, which are performed with respect to the current turtle orientation).

6.3 Changing turtle parameters

The parameter is optional in all the following modules except `@D(`*s*`)`. When no parameter is given, the value of the indicated command in the view file or its default (Section 8.1.1) is used to increase or decrease the current value. When a parameter is given, it sets a new value.

Module	Description
`;(`*n*`)`	Increase the value of the current color index or material index by `color increment`, or set it to *n*.
`,(`*n*`)`	Decrease the value of the current color index or material index by `color increment`, or set it to *n*.
`@;(`*n*`)`	Increase the value of the current color index or material index of the back side of a surface by the second parameter of `color increment`, or set it to *n*.
`@,(`*n*`)`	Decrease the value of the current color index or material index of the back side of a surface by the second parameter of `color increment`, or set it to *n*.
`#(`*n*`)`	Increase the value of the current line width by `line width increment`, or set it to *n*.
`!(`*n*`)`	Decrease the value of the current line width by `line width increment`, or set it to *n*.
`@D(`*s*`)`	Set the current turtle scale to *s*. All subsequent geometry will be scaled by this value. The default scaling factor is `initial scale`.
`@Di(`*s*`)`	Multiply the current turtle scale by `scale multiplier`, or set it to *s*.
`@Dd(`*s*`)`	Divide the current turtle scale by `scale multiplier`, or set it to *s*.

Note that surfaces can have different colors or materials specified for each side only if the view file command `initial color` has two parameters.

6.4 Branching structures

A branch is created by pushing the current state of the turtle (all its parameters) onto a stack, and then popping them off the stack to return to the start of the branch when complete.

Module	Description
`[`	Begin a new branch by pushing the current state of the turtle onto the stack.
`]`	Complete a branch by popping the last turtle state from the stack and setting the turtle to this state (i.e. to beginning of the branch).
`%`	Cut the remainder of the branch: ignore this symbol and all following modules up to the closest unmatched right bracket `]`, therefore eliminating them from the generated string. If no unmatched right bracket is found, all modules to the end of the string are eliminated. See example below. This symbol is ignored if introduced by a homomorphism production.
`@mc(`*flag*`)`	Conditionally cut the remainder of the branch as above only if the value of *flag* is 1. Otherwise, this module has no effect.

For example, if a new branch is defined in the string as

```
...[D(1)]...
```

where the parameter of `D` is the age of the branch, it can be removed from the string when it reaches the age of 20 using the cut symbol `%`:

```
D(a) : a<20 --> D(a+1)
D(a) : a>=20 --> %
```

See object: Shedding

6.5 Polygons

The following modules are used in conjunction with `F` and `f` to create a polygon (Section 6.1).

Module	Description
{	Start a new polygon by pushing the current turtle position onto the polygon stack and setting the polygon flag on.
}	Pop a polygon from the stack and render it, filling it with the current color. If there are no more polygons on the stack, turn the polygon flag off.
.	Create a polygon vertex at the current position, if the polygon flag is on, by placing the current state of the turtle on the polygon stack.

See also the generalized cylinder modules using the { and } symbols with a parameter (Section 6.7.3).

6.6 Circles and spheres

Module	Description	Default
`@o(`*d*`)`	Draw a circle of diameter *d* in the plane of the screen.	current line width
`@c(`*d*`)`	Draw a circle of diameter *d* in the HL plane.	current line width
`@bo(`*d*`)`	Draw the boundary a circle of diameter *d* in the plane of the screen. The width of the boundary is set to the current line width	
`@bc(`*d*`)`	Draw the boundary a circle of diameter *d* in the HL plane. The width of the boundary is set to the current line width	
`@O(`*d*`)`	Draw a sphere of diameter *d*.	current line width

6.7 Surfaces and generalized cylinders

6.7.1 Surface files

Surfaces may be defined in a file that contains all the control points, geometry and neighbourhood information. See the *bezieredit* and *stedit* tools in the *Vlab Tools* manual for interactively creating surface files.

Surface files are specified in the view file (Section 8.1.5) and read at the beginning of the simulation. Each file is identified by an *id*, and a single module within the L-system is used to draw it:

Module	Description	Default
~*id* ~*id*(*scale*) ~*id*(*sx*,*sy*,*sz*)	Draw the surface identified by *id* at the turtle's current location and orientation. If the module has one parameter, the surface is uniformly scaled by *scale*. If there are three parameters, they specify the scaling amount in the *x*, *y*, and *z* directions.	no scaling

6.7.2 L-system defined surfaces

Surfaces can also be defined within the L-system using an internal 4x4 array of control points, identified by an *id*. To define and draw the surface, the following modules are used:

Module	Description	Default
`@PS`(*id*,*type*)	Initialize the 4x4 set of control points for surface *id* to (0,0,0). The *type* can be one of: 1 = Bézier, 2 = B-spline, 3 = Cardinal spline.	*type* = 1
`@PC`(*id*,*r*,*c*)	Set the current position of the turtle to the control point at row *r* and column *c* of surface *id*.	
`@PD`(*id*,*s*,*t*)	Draw the surface defined by the control points for surface *id*, forming a mesh with *s* lines along the rows and *t* lines along the columns.	

6.7.3 Generalized cylinders

The default contour for a generalized cylinder is a circle. However, it is possible to define the contour using the following modules.

Module	Description	Default
{(*type*)	Start a generalized cylinder. The parameter *type* can be one of: 1 = an open curve of Hermite spline segments 2 = a closed curve of Hermite spline segments 3 = an open curve of B-spline segments 4 = a closed curve of B-spline segments This module does not define the first control point. If *type* is not specified, a polygon is defined (Section 6.5).	
}(*type*)	Finish a generalized cylinder started by module {(*type*). The parameter *type* must match in both modules.	
`@Gs`	Start a generalized cylinder at the current turtle position. This is equivalent to {(1).	
`@Ge`(*strips*)	Finish a generalized cylinder. The parameter *strips* defines the number of mesh strips drawn between this final control point and the previous one. This is similar to }(1).	

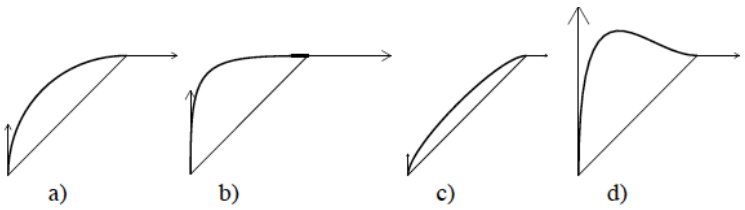


Figure 3: Hermite curves defined as: `@Gs-(45)f-(45)@Gt(start,end)@Ge(20)`, where `start` and `end` are (a) the default: 1.2, 1.2; (b) 2, 2; (c) 0.5, 0.5; (d) 4, 1. Visualized tangent vectors are scaled by 1/4 to fit the figure.

Module	Description	Default
`@Gc(`*strips*`)`	Specify a control point on the central line of the cylinder. The parameter *strips* defines the number of mesh strips drawn between this control point and the previous one.	*strips* = 1
`.(`*strips*`)`	Equivalent to `@Gc(`*strips*`)`. If *strips* is not specified, a polygon vertex is defined (Section 6.5).	
`@!(`*npolygons*`)`	Set the number of polygons around a generalized cylinder, or a cylinder represented by `F` or `G`, to *npolygons*.	
`@#(`*contour-id*`)`	Set the contour specified by *contour-id* as the current contour for generalized cylinders. Contours are specified in the view file (Section 8.1.5). When *contour-id* = 0, the default circle is used.	
`@Gt(`*start,end*`)`	Modify the tangent coefficients at the start and end points of a Hermite curve specifying the generalized cylinder axis. The tangent lengths are equal to the distance between the two control points multiplied by the tangent coefficients. This module must be inserted before the module defining the second control point of the Hermite curve segment. (See Figure 3 and [2]).	*start* = *end* = 1.2
`@Gr(`*angle1, length1,angle2, length2*`)`	Specify the slope and length of two tangents of a Hermite curve which describe the change of radius of the generalized cylinder axis. (See [2]).	
`@Gr(`*flag*`)`	Switch on or off an automatic adjustment of radius tangents for segments of non-unit length. When on (*flag* = 1), the tangents are defined for a segment of unit length and then stretched onto a segment of non-unit length, thus the specified tangent angles do not correspond to the real angles of the tangents. When off (*flag* = 0), tangents are not adjusted after the stretching.	

Only one generalized cylinder can be defined at a time unless it is defined within a branch delimited by square brackets. For example:

```
{(1)f(1)[{(3)f(1)}(3)]}(1)
```

If the generalized cylinder is started using the { module, control points are also defined after each `f` and `F` (Section 6.1), similar to polygons (Section 6.5). Since the number of strips cannot be specified, it defaults to 4.

6.7.4 Textures

Textures are images that can be mapped on surfaces, cylinders, cones, and generalized cylinders (not on disks or spheres). They are only used in material mode (with the `-M` command line option), and are defined in the view file (Section 8.1.7) and referenced by *id* number starting with 1.

Module	Description
`@Tx(`*id*`)`	Set the current texture to *id*. If $id = 0$, texturing is switched off.

If a predefined bicubic surface has an associated texture in the view file, its texture is fixed and cannot be changed by this module.

6.8 Tropisms

Tropisms are specified in the view file (Section 8.1.8) and referenced by *id* number starting with 1.

Module	Description
`@Ts(`*id*`,`*value*`)`	Set the elasticity parameter of tropism to *value*. This overrides the `E:` parameter of the `tropism` command in the view file.
`@T(`*id*`,`*value*`)`	Increase the elasticity parameter. If *value* is not specified, the `S:` parameter of the `tropism` command in the view file is used.
`@Td(`*id*`,`*value*`)`	Decrease the elasticity parameter. If *value* is not specified, the `S:` parameter of the `tropism` command in the view file is used.
`@Tp`	Adjust the turtle's up and left vector to minimize twist. The command operates locally, i.e. it adjust the turtle's vectors only at the current point.
`@Tf`	Force the twist. If the orientation of a segment following symbols `/` or `\` is adjusted due to a tropism (which as a default adjusts the segment's up vector to prevent twist), the effect of symbols `/` or `\` is nullified and it is necessary to add this module to force the twist. It operates locally, i.e. it prevents twist only for symbols `/` or `\` to the left of `@Tf`.

6.9 Query and communication

If a query module (beginning with `?`) is present in any L-system production, an interpretation step is performed after each generate step, even if *cpfg* does not draw to the window. The parameters to the module can then be accessed in the following generate step and affect the selection of productions.

Module	Description
`?P(`x`,`y`)` `?P(`x`,`y`,`z`)`	Query the turtle position.
`?H(`x`,`y`)` `?H(`x`,`y`,`z`)`	Query the current turtle heading vector.
`?L(`x`,`y`)` `?L(`x`,`y`,`z`)`	Query the current turtle left vector.
`?U(`x`,`y`)` `?U(`x`,`y`,`z`)`	Query the current turtle up vector.
`?E(`*p1*`,`*p2*`,...)`	Communicate with an external process. Used to both send and receive environmental information. The parameters can be set by the model and transferred to the environment process, or set by the environment process and transferred to the model. See the *Vlab Environment Programs* manual for more information on environmental processes, as well as [3] and [4].

6.10 Labels

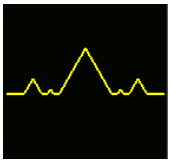
Module	Description
`@L("`*label*`")` `@L("`*format*`",`*p1*`,`*p2*`,...)`	Print *label*, or specify a `printf`-like *format* and print the values of parameters *p1,p2,....* The content is printed in the drawing window at the current turtle location using the font specified in the view file (Section 8.1.9).

7 Advanced topics

7.1 Mouse interaction

It is possible to interact with the generated image using the mouse, by holding down the Shift and Command keys and clicking with the left mouse button. When the mouse is over an element of the model, an `X` module is inserted into the string immediately before the selected module.

Note that if there are several `F` modules in a row, `X` will be inserted before the first `F`. For more granularity, the `F` modules can be separated by a null module, such as `f(0)` or `[]`. To avoid complicating the main L-system, this can be done in a homomorphism:



See object:
Snowflake

```
homomorphism:
F --> Gf(0)
```

Replacing `F` with `G` ensure that the production is not applied recursively.

7.2 Sub-L-systems

Several L-systems can be combined to create a single model, by defining each with a separate label (Section 2). To call a sub-L-system, the following statement is added to the string:

`?(`*label*`,`*s*`)`*mod* `$`

See object:
Sedge

where *label* is the number assigned in the `lsystem:` statement, *s* is the scaling factor, *mod* represents the module that will be processed by the sub-L-system, and `$` returns control to the calling L-system. The module *mod* may have parameters.

For example, the main L-system (`lsystem:1`) can have a production that replaces module `A(t)` with a call to a sub-L-system (`lsystem:2`), along with the module `B(t-1)`. The sub-L-system then has a production to match module `B`:

```
...
lsystem: 1
...
A(t) --> ?(2,1)B(t-1)$
...
endlsystem

lsystem: 2
...
B(x) --> ....
...
endlsystem
```

Note that each L-system requires an `axiom` statement, but the axiom is ignored when it is called as a sub-L-system. However, if the L-system is run independently, it will use the axiom, which should include the module defined in the sub-L-system call. For example, the sub-L-system above could have the following statement:

```
axiom:  !(3);(48)B(5)
```

to provide a starting point for the L-system that includes module `B(x)` with the defined parameter `5`.

8 *cpfg*-SPECIFIC INPUT FILES

8.1 VIEW FILE

The view file contains drawing, viewing, and rendering commands, as well as the names of external files such as surfaces and functions.

Unless otherwise stated, the values have the following formats:

- *x,y,z* - floating point numbers
- *i* - integer
- *id* - a single character
- *flag* - `on` or `off`

Comments may be included in the file using the standard C notation:

```
/* ... */
```

Note that the commands are processed in the order they are specified in the file. Thus if there are multiple instances of the same command, the last one takes precedence. Exceptions are noted for individual commands that allow for multiples.

8.1.1 Turtle commands

Command	Description	Default
`angle increment:` *x* `angle factor:` *y*	Set the angle associated with the `+`, `-`, `&`, `^`, `\`, `/`, and `|` modules. The angle is set to $x°$, or to $360°/y$. These are alternative methods of setting the same parameter; therefore, the last one specified will be applied.	45°
`initial color:` *i1 i2*	Set the initial index into the colormap or material file. The value should be a number between 0 and 255. The second value is optional. If present, *i2* specified the index to be used for the back side of the surface.	128
`color increment:` *i*	Set the color increment associated with the current colormap or material file.	1
`initial line width:` *x style*	Set the initial line width to *x*. The *style* parameter is optional; if included, the values are: `pixels` or `p` - flat shaded lines with width in pixels `shaded` or `s` - shaded cylinders in world units Alternatively, the `line style` command (Section 8.1.5) can be used.	*width* = 1 *style* = `p`
`line width increment:` *x*	Set the line width increment associated with the `#` and `!` modules to *x*, using the same units as `initial line width`.	0
`initial scale:` *x*	Set the initial scale factor associated with the turtle to *x*. All geometry will be scaled by this factor. This value can be modified by modules `@D`, `@Di`, and `@Dd` (Section 6.3).	1

Command	Description	Default
`scale multiplier:` *x*	Set the multiplicative factor by which the turtle scale is multiplied or divided when module `@Di` or `@Dd` (Section 6.3) is interpreted.	1
`interpretation past %:` *flag*	Interpret past the cut symbol `%` if *flag* = `on`. When *flag* = `off`, the symbols after `%` are not interpreted.	`on`

8.1.2 Setting the view

Command	Description	Default
`viewpoint:` *x,y,z*	Set the *x,y,* and *z* coordinates of the viewpoint in world space.	0,0,1
`view reference point:` *x,y,z*	Set the *x,y,* and *z* coordinates of the view reference point in world space.	0,0,0
`twist:` *i*	Rotate the image on the screen by *i* tenths of a degree.	0
`projection:` *type*	Set the projection to *type*, which can be either `parallel` or `perspective`. Auto-centering and auto-scaling work only in parallel mode.	`parallel`
`viewing angle:` *x*	Set the viewing angle of perspective projection. This command is ignored in parallel projection.	45°
`front distance:` *x*	Set the distance from the viewer to the front clipping plane in perspective project, or the position of the clipping plane with respect to the viewpoint in parallel projection (thus a negative value must be used).	-100000
`back distance:` *x*	Set the distance from the viewer to the back clipping plane in perspective project, or the position of the clipping plane with respect to the viewpoint in parallel projection.	100000
`scale factor:` *x*	Set the size of the final image on the screen. When *x* = 1, the image is full size.	1

`box: x:` *xmin, xmax*
`y:` *ymin, ymax*
`z:` *zmin, zmax*
	Set the bounding box for the model. The view is adjusted so that the whole bounding box is visible. Effective only in parallel projection.	`x: 0,1`
		`y: 0,1`
		`z: 0,1`

`min zoom:` *zmin* `max zoom:` *zmax*	Set how much the user can zoom in or out on the view.	*zmin* = 0.05 *zmax* = 50
`generate on view change:` *flag*	Regenerate the L-system string (the simulator rewinds to the axiom and performs the derivations again) every time the view changes through rotation, zoom, or pan (*flag* = `on`), or after the user releases the mouse button (*flag* = `triggered`).	`off`

Note that modifying `scale factor` in perspective projection moves the viewer closer or farther from the viewpoint and therefore the `front distance` and `back distance` will need adjusting.

8.1.3 Lights

Colormap mode. The following commands are used in colormap mode only.

Command	Description	Default
`light direction:` *x,y,z*	Define the direction of light as the *x, y,* and *z* coordinates of a vector.	(1,0,0)
`diffuse reflection:` *i*	Define the coefficient used to determine the range of colors for lighting a shaded surface. See below for the formulas that use this coefficient.	10
`surface ambient:` *a* `surface diffuse:` *d*	Define the amount of ambient and diffuse light present for shading bicubic surfaces and tsurfaces, where *a* and *d* are numbers between 0 and 1. See below for the formula used.	$a = 0.15$ $d = 0.75$

The `diffuse reflection`, `surface ambient` and `surface diffuse` commands are used to vary the color based on the orientation of polygons representing the surface or cylinder with respect to the direction of the light source, where $\vec{N}$ is the normal of the polygon, and $\vec{L}$ is the direction towards the light source. Only the first light source is used, if more than one is specified in the view file.

The `diffuse reflection` coefficient i is used as follows:

- For a surface color *col*, the range of colors is varied within the interval [*col-i, col+i*].
- For a polygon representing a cylinder or generalized cylinder, the original color index *col* is used to calculate the final color using the formula: $col + i \cdot \vec{N} \cdot \vec{L}$

The `surface` coefficients, a and d, are used to calculate the final color of a surface with the formula:

$$64 \cdot int \cdot (a + d \cdot abs(\vec{N} \cdot \vec{L}))$$

where the color intensity *int* is calculated using the color index *col* associated with the surface:

$$int = col/64 - \texttt{floor}(col/64)$$

Material mode. In material mode the `light` command is used. There may be multiple light commands, each with a set of subcommands. Therefore, the syntax is:

```
light:  subcmd1 subcmd2 subcmd3 ...
light:  subcmd1 subcmd2 subcmd3 ...
...
```

The subcommands of `light` are:

Subcommand	Description	Default
`O:` *x y z*	Define the origin (x, y, z) of a point light source.	0 0 1
`V:` *x y z*	Define the vector (x, y, z) for a directional light source.	
`A:` *r g b*	Define the ambient light color in RGB components.	1 1 1
`D:` *r g b*	Define the diffuse light color in RGB components.	1 1 1
`S:` *r g b*	Define the specular light color in RGB components.	1 1 1
`P:` *x y z e c*	Define a spotlight with direction *(x,y,z)*, exponent *e*, and cutoff angle *c*.	0 0 -1 0 180
`T:` *c l q*	Define attenuation factors: constant *c*, linear *l*, and quadratic *q*.	1 0 0

8.1.4 Rendering

Command	Description	Default
`render mode:` *mode*	Render the object using one of the following modes: `filled` - All polygons have the same color. If materials are specified the diffuse color is used. `fast` - Same as `filled` except spheres and disks are drawn in wireframe. `wireframe` - All objects are drawn in wireframe. `interpolated` - Interpolate between colors at the beginning and end of a line or cylinder, or at different vertices of a polygon. `flat` - The color of each polygon is determined by its position with respect to the light. (See explanation below.) `shaded` - In colormap mode, the color is computed for each vertex of the polygon, similar to `flat` mode. In material mode, the normal for each polygon can be different at each vertex, resulting in smooth shading. `shadows` - Include shadows. Additional shadow parameters are defined using the `shadow map` command below.	`filled`

`shadow map:`
`size:` *n*
`color:` *r g b*
`offset:`
factor units
	Define parameters for shadow mapping when using `render mode: shadows`. The shadow map will be generated using the first directional or spot light source specified with the `light` command. The following parameters are optional:
	`size`: the width and height of the shadow map (*n* x *n*), where *n* must be an even number. Values that are too small ($n < 100$) or too large (dependent on graphics card) may cause shadows not to display.
	`color`: shadow color in *rgb* components.
	`offset`: polygon offset for generating depth map used to reduce shadow acne (erroneous self-shadowing). To reduce shadow acne, try increasing these values.
		n = 1024
		r = 0.2
		g = 0.2
		b = 0.4
		factor = 5
		units = 10

`z buffer:` *flag*	Define whether hidden surface elimination using the z buffer should be `on` or `off`. When `off`, the last object drawn will be visible.	`on`
`antialiasing:` *flag*	Draw with antialiasing (*flag* = `on`). Currently, this only works with line primitives.	`off`
`concave polygons:` *flag*	Enable the OpenGL tesselator (*flag* = `on`), which divides polygons into triangles. This allows for more complex concave polygon shapes, but will cause *cpfg* to run slower.	`off`
`background scene:` *filename1, filename2, ...*	Specify additional objects to be drawn after the L-system generated string is interpreted. Each object is in a file containing a set of graphics commands (Section 8.3).	

When using `flat` mode, the color of each polygon representing surfaces, lines, or generalized cylinders is determined according to its position with respect to the direction towards the first light source specified in the view file (Section 8.1.3); other sources are ignored. In material mode, a single normal is used for the whole polygon. For colormap mode, see the notes on the `diffuse reflection`, `surface ambient` and `surface diffuse` commands in the view file.

8.1.5 Lines, contours, and surfaces

Command	Description	Default
`line style:` *style*	Specify how lines (represented by modules `F` and `G`) are drawn, where *style* is one of the following: `pixel` - flat shaded lines with width in pixels `polygon` - flat shaded polygons with width in world units `cylinder` - cylinders with width in world units	`pixel`
`tapered lines:` *flag*	Control whether lines and cylinders are drawn tapered (*flag* = `on`).	`on`
`line:` *id filename*`.s` *s*	Specify a line using a surface file *filename*`.s` and assign it an *id*. The scaling factor *s* is mandatory.	
`contour:` *id filename*	Specify a contour file *filename* and assign it an *id*. Contours are defined as planar B-spline curves, and are considered cross-sections of generalized cylinders. See the *cuspy* tool in the *Vlab Tools* manual for interactively creating a contour file.	
`contour sides:` *n*	Define the level of detail used in generating the polygons for spheres and cylinders. For cylinders, *n* is the number of polygons around the circumference ($n > 3$). For spheres, the closest upper power of two is used. For smooth connections between cylinders and spheres when using small values of *n*, use a power of 2. This initial value can be modified by module `@!` (Section 6.7.3).	3
`twist of cylinders:` *flag*	Do not minimize the twist when drawing generalized cylinders (*flag* = `on`).	`off`
`surface:` *id filename*`.s` *s i j texfile*	Specify a surface file *filename*`.s` and assign it an *id*. The scaling factor *s* is mandatory. The values *i* and *j* are optional, but if present, define the level of detail used when drawing patches: *i* polygons along the rows, and *j* polygons along the columns. The final argument, *texfile*, is also optional, and specifies a texture file associated with the surface. This value takes precedence over the texture *id* assigned with the `@Tx` module (Section 6.7.4). All instances of this surface will have the same texture. See the *bezieredit* and *stedit* tools in the *Vlab Tools* manual for interactively creating surface files.	$i = j = 0$
`tsurface:` *id filename*`.ray` *s*	Specify a surface file *filename*`.ray` in *rayshade* format (Section 8.4), and assign it an *id*. The scaling factor *s* is mandatory.	

There may be several `line`, `contour` and `surface` commands in the view file. Each should have a unique *id*.

8.1.6 User-defined functions

The following commands specify files containing user-defined functions called with `func` (Section 5.5). The files contain function(s) of one variable defined as B-spline curves constrained to assign exactly one *y* value for every *x* in the normalized function domain [0,1]. See the *funcedit* tool in the *Vlab Tools* manual to interactively create a single function, and the *gallery* tool to create a set of functions.

The files are specified using the following commands:

Command	Description
`function:` *filename n*	Specify the *filename* containing a function. If the optional parameter *n* is present, *cpfg* will precompute *n* values of the function, evenly spaced in [0,1] and the value returned when the function is called will be a linear interpolation of the two closest precomputed values. If the parameter is omitted, the return value will be computed exactly each time the function is called.
`function set:` *filename*	Specify the *filename* containing a set of user-defined functions, rather than a single function. The number of precalculated samples (parameter *n* above) is specified in the function set file.

The *id* parameter used to identify the function in `func(`*id, x*`)` is based on its order in the view file or function set file, beginning with 1. When preprocessing the L-system file, *cpfg* will also create define commands of the form:

`-D` *FUNCNAME=id*

where *FUNCNAME* is an all-capitals version of the function's name as specified in the file, and *id* is its order number. This makes it possible to call a function by name rather than its position, using the format:

`func(`*FUNCNAME,x*`)`

where *x* is the parameter to the function. This applies to functions specified by both the `function` and `function set` commands.

8.1.7 Textures

Textures can be mapped on surfaces, cylinders, cones, and generalized cylinders (not disks or spheres). The command has the same format as the `light` command (Section 8.1.3):

`texture:` *subcmd1 subcmd2 subcmd3 ...*
`texture:` *subcmd1 subcmd2 subcmd3 ...*
`...`

The subcommands are:

Subcommand	Description	Default
`F:` *filename*	Specify the image file containing the texture, in PNG or RGB format. This subcommand is mandatory. The image width and height are clamped in such a way that the image size is $2^m \times 2^n$.	
`E:` *mode*	Control the way the texture is combined with the surface colors, where *mode* is one of the following: `modulate` or `m` - multiply the surface color with the texel color `decal` or `d` - use the texel color; the surface is not shaded `blend` or `b` - interpolate between surface and texture color using the color index value of the surface (colormap mode) or the material properties (material mode).	`modulate`
`S:`	Map the surface texture per surface, not per patch. The surface boundaries are found and then the texture is mapped into the $z = 0$ plane with respect to the computed boundaries.	
`R:` *ratio*	Define the aspect ratio of a texture mapped on a cylinder or generalized cylinder. A value greater than 1 causes the texture to be more stretched along the cylinder.	1

Subcommand	Description	Default
`H:` *filter*	Define the display mode for textures with texels larger than image pixels, where *filter* can be: `linear` or `l` - texture image is smoothed `near` or `n` - texture pixels are visible	`near`
`L:` *filter*	Define the display mode for textures with texels smaller than image pixels, where *filter* can be: `linear` or `l` - more texture pixels are used to compute the given pixel `near` or `n` - only one texture pixel is used to compute the given pixel (may result in aliasing). `mnn` - use the nearest mipmap image and the nearest pixel in this mipmap. Produces some artefacts but is the fastest. `mln` - use the nearest mipmap image and linearly interpolate between neighbouring pixels. Still produces some artefacts. `mnl` - use the nearest pixels in the two best mipmaps and interpolate between values. `mll` or `m` - linearly interpolate between neighbouring pixels in the two best mipmaps and interpolate between the values. Produces the best result, but may be slower.	`near`

For mipmaps, the OpenGL library creates a smaller version of the texture (down to a size of 1×1), and for smaller objects uses the smaller texture, resulting in faster displaying. These are the four filters for `L:` beginning with `m`.

8.1.8 Tropisms and torque

There are two commands that have the same set of subcommands: `tropism` sets the tropism parameters, and `torque` sets the parameters of movement that adjust segments around their heading without modifying the heading orientation. The `A:` subcommand is ignored in the `torque` command. There may be multiple tropisms defined, each assigned a sequential *id* starting with 1.

The commands consists of a series of subcommands:

```
tropism:  subcmd1 subcmd2 subcmd3 ...
...
torque:  subcmd1 subcmd2 subcmd3 ...
...
```

The subcommands are:

Subcommand	Description	Default
`T:` *x y z*	Define the tropism vector. This subcommand is mandatory.	
`A:` *angle*	Define the *angle*, in degrees, with respect to the tropism vector that segments are trying to reach. For example, a 90° angle corresponds to diatropism.	0°
`I:` *intensity*	Define the global *intensity* of the tropism.	1
`E:` *elasticity*	Set the initial *elasticity*.	0
`S:` *step*	Set the elasticity *step*.	0

See Section 6.8 for modules using these tropisms. The elasticity modules `@Ts, @T` and `@Td` may override the `E` and `S` values defined here.

8.1.9 Fonts

Command	Description	Default
`font:` *font*	Define the font type to be used when interpreting the `@L` modules (Section 6.10), using the X font specification.	`-*-courier-bold-r-*-*-12-*-*-*-*-*-*-*`

8.1.10 Deprecated view commands

The following commands exist only for backwards compatibility. For all new view files, use the command in the '**Use instead**' column.

Command	Description	Use instead
`shade mode:` *i*	Render using one of the following modes: 1 - Simple fill (default) 2 - Interpolated fill 3 - Gouraud shading 4 - B-spline 5 - Closed B-spline 6 - Two sided 7 - Wireframe	`render mode`
`polygonization level:` *n*	Define the level of detail used in generating the polygons for spheres and cylinders. A high value, such as 4, will generate very smooth surfaces, but take longer to display. The lowest value is 1 and produces very rough approximations of the surfaces. The default is 3.	`contour sides`
`tropism direction:` *x,y,z*	Define the direction toward which branches tend to bend using the vector with coordinates *x, y,* and *z*. The default is (0,1,0).	`tropism`
`initial elasticity:` *x*	Specify the susceptibility of a branch to bending. The default is 0.	`tropism`
`elasticity increment:` *x*	Increment or decrement the elasticity by *x*. The default is 0.	`tropism`

The following commands are ignored in the current version, but may exist in old models.

Command	Description	Use instead
`ambient light:` *red, green, blue*	Define the *red, green,* and *blue* components of ambient light.	In colormap mode: `surface ambient` In material mode: `light`
`background color:` *red, green, blue*	Define the *red, green,* and *blue* components of the background color.	In colormap mode: color index 0 In material mode: emission color
`cue range:` *x*	Specify the range of color indices used for depth cueing. A default of 0 indicates no depth cueing. Usual values are between 10 and 100.	
`interpretation step:` *i*	Set the number of interpreted symbols before the next X event is checked.	

8.2 Animation file

This file is input when *cpfg* enters Animation mode, either from the main menu (Section 1.3.2) or with the `-a` option on the command line (Section 1.2.1). If this file is not specified on the command line, default values are set for the animation commands.

Command	Description	Default
`first frame:` *i*	Set the first frame to be interpreted.	1
`last frame:` *i*	Set the last frame to be interpreted.	`derivation step`
`step:` *i*	Set the number of derivation steps between drawing (and recording) of frames.	1
`clear between frames:` *flag*	Specify whether the screen should be cleared between frames (*flag* = `on`). When `off`, successive images are superimposed.	`on`
`new view between frames:` *flag*	Specify whether to reset the view between frames (*flag* = `on`), or not (*flag* = `off`).	`off`
`scale between frames:` *flag*	Specify whether the view should be adjusted to fit the entire object in the window (*flag* = `on`), or not (*flag* = `off`). If `on`, this adjustment is made before applying the `scale factor` command (Section 8.1.2). This command works only in parallel projection.	`off`
`double buffer:` *flag*	Specify whether double buffering is `on` or `off` during animation.	`on`
`frame intervals:` *f1, f2, f3, ...*, *fa-fb, ...*, *fc-fd* `step` *s*, *fe-ff* `rotate` *rx ry rz*, *fg-fh* `scale` *sx sy xz*	Define specific frames and/or ranges of frames to be interpreted. A range may also include: `step`, to override the `step` command with a specified step *s*; `rotate`, to rotate after each frame by a specified angle (in degrees) around each axis; or `scale`, to scale after each frame. See examples below. This command takes precedence over the `first frame`, `last frame`, and `step` command regardless of the order in the file.	

When `clear between frames` is turned off, `double buffer` should also be turned off. Otherwise only every second frame will be displayed.

The `frame interval` command can simply select specific frames to be drawn. For example, to select frames 1, 3, 4, 5, 10, 12, 14, 16, 19, 21, the command would be:

```
frame intervals: 1, 3-5, 10-16 step 2, 19, 21
```

It can also specify a rotation after each frame. For example, to rotate the object 1.5° around the x axis after each frame is drawn, beginning with frame 11:

```
frame intervals: 1-10, 11-100 rotate 1.5 0 0
```

Or it can specify a scaling factor for each frame. For example, to scale the object to 90% of its size for each of the first 100 frames:

```
frame intervals: 1-100 scale 0.9 0.9 0.9
```

Deprecated commands The following animation commands are ignored in the current version, but may exist in old models.

Command	Description
`swap interval:` i	Set the minimum time i (in tenths of a second) between swapping of buffers in double buffer mode. The time is measured between the moment *cpfg* begins drawing one frame until the moment it begins drawing the next frame. If it takes longer to draw a frame, the delay between frames will be longer.

8.3 Background file

A background scene can be effectively used to define additional objects around a simulated plant. It can also be used during simulation of plant-environment interactions, for visualizing the environmental field together with the plant (see the *vlab* Environmental Programs manual).

The name of the background file is specified by the `background scene` command in the view file (Section 8.1.4). The file consists of commands similar to statements in the OpenGL® graphics library.

8.3.1 Primitives

The following commands define basic geometric primitives, where the coordinates of the vertices and/or the size of the primitives are defined with respect to the local coordinate system. It is possible to translate or scale the objects by translating and/or scaling the coordinate system using the transformation commands (Section 8.3.3).

Command	Description
`polygon` $x_1\ y_1\ z_1\ \ldots\ x_n\ y_n\ z_n$	A polygon with n vertices, where $n \geq 3$.
`polygonuv` $x_1\ y_1\ z_1\ nx_1\ ny_1\ nz_1$ $\ldots\ x_m\ y_m\ z_m\ nx_m\ ny_m\ nz_m$	A polygon with m vertices, where $m \geq 3$. Each vertex also has an associated normal (nx, ny, nz).
`rectangle` $a\ b$	A rectangle with one vertex at (0,0,0) and edges of length a and b along the positive x and y axes respectively.
`mesh` $x_1\ y_1\ z_1\ \ldots\ x_n\ y_n\ z_n$	A rectangular mesh, where $n = 4 + 2k$ and $k \geq 0$. Vertices $2k, 2k+1, 2k+2$, and $2k+3$ define a single rectangle of the mesh.
`box` $a\ b\ c$	A box with one vertex at (0,0,0) and edges of length a, b, and c along the positive x, y and z axes, respectively.
`cone` $r_1\ r_2\ h$	A cone with its axis along the y axis, a radius of r_1 at the base and r_2 at the top, and a height of h.
`cylinder` $r\ h$	A cylinder with its axis along the y axis, a radius of r, and height h.
`sphere` r	A sphere with center at (0,0,0) and radius r.

8.3.2 Materials

Command	Description
`material` $n_1\ n_2\ \ldots\ n_{17}$	This material is applied to all subsequently defined primitives. n_1 - n_4: ambient light n_5 - n_8: diffuse color n_9 - n_{12}: specular color n_{13} - n_{16}: emissive color n_{17}: the specular exponent

The four values for each color are the R, G, and B components of the color (in the range of 0 to 1), and the alpha value controlling the opacity (1 = opaque, 0 = transparent). The specular exponent is a number in the range 0 to 128.

8.3.3 Transformations

The primitives above are defined with respect to a local coordinate system. The world coordinate system is expressed by a single matrix, specifying the transformation necessary to map the world coordinate system into the local coordinate system. Thus every rotation, translation, or scaling modifies only the transformation matrix. This approach is equivalent to the use of the *modelview matrix* in OpenGL®.

The transformation matrix is set up using the commands:

Command	Description
`loadidentity`	Set the transformation matrix to identity (i.e. equal to the local coordinate system).
`loadmatrix` a_1 a_2 ... a_{16}	Set the transformation matrix to a 4×4 matrix where a_1 - a_4 are the values of the first column, a_5 - a_8 the second column, and so on.
`pushmatrix`	Store the current transformation matrix on a matrix stack.
`popmatrix`	Retrieve a matrix from the stack and set the transformation matrix to these values.

Transformations are then performed by modifying the current transformation matrix with the commands:

Command	Description
`translate` *tx ty tz*	Translate the local coordinate system by vector (tx, ty, tz).
`rotate` *angle vx vy vz*	Rotate the coordinate system around vector (vx, vy, vz) by *angle* degrees.
`scale` it sx sy sz	Scale the local coordinate system by factors *sx, sy,* and *sz* along the *x, y,* and *z* axes respectively.
`mulmatrix` a_1 a_2 ... a_{16}	Multiply the current transformation matrix by the specified matrix.

8.3.4 Lighting and projection

The `-gls` command line option outputs the geometry of the L-system string in the format described above. Thus, the geometry produced from one simulation can be included as a background scene in another simulation.

The output file may also include commands for lighting and projection, not required in a background file:

Command	Description
`clear` *red green blue*	The RGB components of the background color.
`light` *x y z w*	The light source, specified as four homogeneous coordinates of light position. If $w = 1$, the light is directional. The color of the light is always white.
`ortho` *minx maxx miny maxy front back*	An orthographic projection, where *front* and *back* are distances.
`perspective` *angle front back*	A perspective projection, where *angle* is the viewing angle, and *front* and *back* are distances.
`lookat` *posx posy posz refx refy refz upx upy upz*	The view: the camera position (*pos*), the view reference point (*ref*) and, optionally, the up vector (*up*).

Command	Description
`maxtrixmode` *mode*	The current matrix mode: 0 = *modelview* matrix 1 = projection matrix

8.3.5 Example of a background file

A background scene composed of a sphere, cone and box, all in grey, can be defined as follows:

```
material 0.1 0.1 0.1 1    /* subsequent surfaces are grey */
         0.16 0.21 0.27 1 /* with no specular reflections */
         0 0 0 1          /* and no emissive color */
         0 0 0 1
         0

pushmatrix
translate 3 -20 -3
scale 1 0.7 0.7
sphere 15 /* ellipsoid */
popmatrix

pushmatrix
translate -14 -55.0 8
cone 15 2 14 /* cone */
popmatrix

translate -10 -65 0
box 30 5 30 /* box */
```

The file is preprocessed by *cpfg*, and therefore macros and comments are allowed.

8.4 Tsurface specification file

An alternative to using a surface editor to create a surface as a bicubic patch (see the *Vlab Tools* manual), is to define a set of triangles in a text file, using the same syntax as a *rayshade* output file. Each triangle is define by its three vertices (one per row), the normal to the vertice, and (optionally) the coordinates of a texture at the vertex:

`triangle`
$x_1\ y_1\ z_1\ nx_1\ ny_1\ nz_1\ u_1\ v_1$
$x_2\ y_2\ z_2\ nx_2\ ny_2\ nz_2\ u_2\ v_2$
$x_3\ y_3\ z_3\ nx_3\ ny_3\ nz_3\ u_3\ v_3$

where x_n, y_n and z_n are the coordinates of the vertex, nx_n, ny_n and nz_n specify the normal to the vertex, and u_n and v_n are the coordinates of the texture at the vertex. For example, two triangles without texture coordinates could be specified as:

```
triangle
-0.5 1 0 0 0 1
 0.0 2 0 0 0 1
 0.5 1 0 0 0 1
```

```
triangle
 0.5 1 0 0 0 1
 0.0 0 0 0 0 1
-0.5 1 0 0 0 1
```

and with added texture coordinates:

```
triangle
-0.5 1 0 0 0 1 0.5 0.0
 0.0 2 0 0 0 1 1.0 0.5
 0.5 1 0 0 0 1 0.5 1.0
triangle
 0.5 1 0 0 0 1 0.5 1.0
 0.0 0 0 0 0 1 0.0 0.5
-0.5 1 0 0 0 1 0.5 0.0
```

9 Appendix: Deprecated / Undocumented features

The following features are no longer tested or supported, but may exist in older models.

9.1 Command line arguments

The following command line arguments are no longer in use:

Feature	Description	Use instead
`-c`*mapnr*	Define the 256-entry portion of the color or material table to be used.	`-m` and `-M` (Section 1.2.2)
`-C` *commstr*	Specify connections to other processes in a distributed simulation. Not available for Mac iOS.	
`-mb`	Create a menu bar, as well as the pop-up menu.	
`-pipestrb`	Convert a binary L-system string from *stdin* into a format specified by one of the output file options.	
`-pm`	Use X pixmap as back buffer.	
`-s`*strsize*	Define the initial space allotment for a string generated by an L-system.	
`-S` *socketno*	Define the socket number to be used to receive menu commands from external programs.	File monitoring using Refresh mode (Sections 1.2.1 and 1.3.2)
`-sb`	Set to single buffering during animation.	`double buffer` command in the Animation file (Section 8.2)

9.2 Move and save substrings

The modules `%(`*par*`)` and `%(`*par, turtle-index*`)` to move a substring but save its position (the turtle parameters) are no longer supported. However, the module `%` alone is still available to cut a branch from the string (Section 6.4).

The `%(`*par*`)` functionality enabled a substring between `%` and `%(`*par*`)` to be moved to the end of the string, preceded by `%(`*par, turtle-index*`)`, where *turtle-index* was a position in a special turtle array that stored the turtle parameters as they were when the module `%(`*par*`)` was encountered. After the substring was moved to the end of the L-system string, each time a module `%(`*par, turtle-index*`)` was encountered in the following interpretation steps, the turtle parameters were set to the values stored in the turtle array under *turtle-index*. This functionality is similar to the branching modules `[` and `]` (Section 6.4) without moving the substring.

9.3 Derivation Length

The functions `GetDerivationLength(`*dummy*`)` and `SetDerivationLength(`*s*`)` are no longer supported. If the derivation length is required within the L-system, it can be defined. For example:

```
#define DERIV 50
...
derivation length: DERIV
...
A(x) : x<DERIV-10 --> A(x+1)
A(x) : x>=DERIV-10 --> A(x+1)B
```

There is no functionality to change the derivation length during execution of *cpfg*, but it is possible to end before the final derivation step using the `stop` function (Section 5.6), and to end an animation earlier using the `last frame` command in the animation file (Section 8.2).

9.4 Rayshade functionality

It is possible to output a file in *rayshade* format using the Save As menu item (Section 1.3.2). However, specific *rayshade* functionality within L-system productions is no longer supported. This includes:

- The homomorphism production delimiter `-o>` to instantiate objects, as well as the `rayshade objects` command in the view file to set parameters for these objects.
- The `rayshade scale` command in the view file to scale objects.
- The module `@J(`*s1,s2,s3*`)` to close the current grid for objects being output to *rayshade*, and start a new grid with the given size.
- The module `@I("`*rayshade-object*`",`*s*`)` to include a *rayshade* object at the current turtle location.

To match the scaling of *rayshade* objects generated by different L-systems use the view file command `initial scale` (Section 8.1.1).

9.5 System calls

The module `@S("`*system-call*`")` is not available for MacOS systems.

10 Credits

The original version of *cpfg* was implemented by Jim Hanan as part of his Ph.D. work [5], and described in [6]. It has been extensively enhanced by Jim Hanan and Radomír Měch [2; 3], Mark Hammel, Mark James, Brendan Lane, and most recently by Pascal Ferraro and Mikolaj Cieslak.

Vlab uses a modified version of the rendering program *rayshade* written by Craig Kolb [7] for the Save as Rayshade and `-ray` options.

11 Document revision history

Date	Description	By
1992	Original documentation	Jim Hanan
1997	Based on Version 2.7 of *cpfg*	Mark James Mark Hammel Jim Hanan Radomír Měch Przemyslaw Prusinkiewicz
2016	Based on Version 4.0 of *cpfg*	Radomír Měch Radoslaw Karwowski Brenda Lane
2021	Updated to current version	Lynn Mercer Przemyslaw Prusinkiewicz Mikolaj Cieslak Pascal Ferraro

References

[1] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. with James S. Hanan, F. David Fracchia, Deborah R. Fowler, Martin J.M. de Boer, and Lynn Mercer. Springer-Verlag, 1990.

[2] Radomír Měch, Przemyslaw Prusinkiewicz, and James Hanan. Extensions to the graphical interpretation of l-systems based on turtle geometry. Technical Report 97/599/01, University of Calgary, Dept of Computer Science, 1997.

[3] Radomír Měch. *Modeling and Simulation of Interaction of Plants with the Environment using L-systems and Their Extensions*. Phd thesis, University of Calgary, 1997.

[4] Przemyslaw Prusinkiewicz, Mark James, and Radomír Měch. Synthetic topiary. In *Computer Graphics*, volume 38, pages 351–358. Proceedings of SIGGRAPH '94, 1994.

[5] James S. Hanan. *Parametric L-systems*. Phd thesis, University of Regina, 1992.

[6] Przemyslaw Prusinkiewicz and James Hanan. L-systems: From formalism to programming languages. In G. Rozenberg and A Salomaa, editors, *Lindenmayer systems: Impact on theoretical computer science, computer graphics, and developmental biology*, pages 193–211. Springer-Verlag, 1992.

[7] Craig Kolb. Rayshade. URL `http://www.graphics.stanford.edu/~cek/rayshade/rayshade.html`.