CPFG
Version 3.4
User's Manual

Radomír Měch

May 7, 1998

based on the CPFG Version 2.7 User's Manual by
Mark James
Mark Hammel
Jim Hanan
Radomír Měch
Przemyslaw Prusinkiewicz

Contents

[1] This section is incorporated from [2].

1 Introduction

The plant and fractal generator with continuous parameters (cpfg) is a program for modeling plants and visualizing their development. It can also be used to generate images of 2D and 3D fractals. Models are expressed using the formalism of L-systems.

This manual assumes that the reader is familiar with the concepts of L-systems and turtle interpretation presented in *The Algorithmic Beauty of Plants* [7], as well as the elements of the C programming language.

Part I contains reference materials. It describes cpfg usage, user interaction, and input and output file formats. Part II contains examples, from an L-system for a simple fractal to realistic models of plants.

2 Machine requirements

The cpfg program runs on SGI workstations, and works best on machines with 24 bit planes. The program will run on machines with only 8 bit planes, but many of the example models will not show up in the correct colors. The described distribution has been compiled and tested using IRIX 5.3 release of the operating system.

A C macro preprocessor is also required (the default preprocessor is invoked by cpfg using the *cc -E* call).

3 Distribution

The simplest way to use cpfg is within the Virtual Laboratory Environment (vlab). The complete cpfg and vlab software distribution (binaries only) with sample models and documentation is available at:

`http://www.cpsc.ucalgary.ca/projects/bmv/vlab/index.html`

Part I
Reference

4 Command line parameters

A call to `cpfg` takes the following form:

`cpfg` [-s*string_size*] [-v] [-V] [-d] [-P *preprocessor*] [-a] [-e environmentfile] [-g] [-pipestrb] [-C *communication_setup_string*] [-S *socket_num*] [-c *mapnr*] [-w *xsize ysize*] [-wp *xpos ypos*] [-wt *window_title*] [-m[*n*] *colormap_file*] [-M[*n*] *material_file*] [-mb] [-pm] [-sb] [-o] [-homo] [-rgb *rgbfile*] [-ras *rasfile*] [-tga *tgafile*] [-rle *rlefile*] [-ray *rayfile*] [-ps *psfile*] [-str *textstringfile*] [-strb *binarystringfile*] [-gls *glsfile*] [-vv *vvfile*] [-iv *ivfile*] *L-system_file View_file* [*Animation_file*]

The *L-system_file* and *View_file* arguments are mandatory; the arguments in square brackets are optional. A call to `cpfg` without any arguments prints a message with a list of options.

4.1 General

-s*string_size* The value of the integer *string_size* defines the initial space allotment for a string generated by an L-system. The default value is 150,000. Note that there is no space between `-s` and *string_size*.

-V This option puts `cpfg` in verbose mode, in which a trace of input data and execution details are printed to *stdout*.

-v This option puts `cpfg` in warning mode. A trace of input data and execution details (significantly reduced compared to the verbose mode) are printed to *stdout*.

-d This option puts `cpfg` in debugging mode. Selected information pertinent to the `cpfg` operation is printed to *stdout*. This mode is intended only for code development.

-P *preprocessor* Changes the C macro preprocessor applied to the L-system file and view file. The default preprocessor is invoked by `cpfg` using the `cc -E` call. For example `-P acpp` invokes the ANSI C preprocessor.

-a The program starts in animate mode (with animate menu). This option is necessary in off-screen rendering modes (`-g`, `-pipestrb`) to create an animation according to the animate file.

-e *communication_specification_file* specifies parameters of plant-field communication and switches on environmental mode, in which `cpfg` communicates with an external program simulating the environment.

L-system_file The L-system definition. By convention, this file name has suffix *.l*. See Section 6.1 for details.

View_file Contains viewing, rendering, and drawing parameters including the names of surface-specification files. By convention, this file name has suffix *.v*. See Section 6.2 for details.

Animation_file Contains parameters controlling frame by frame production of images for animation purposes. By convention, this file name has suffix *.a*. See Section 6.3 for details.

4.2 Graphics and windowing

-c*map_nr* The value of the integer *map_nr* defines the 256-entry portion of the color or material table to be used by `cpfg`. A value of 0 indicates the first 256 entries, 1 the second 256, and so on (up to 15). The background is colored using the first entry of the selected color table or using the emissive color of the first entry of the selected material table. The colors or materials used by the turtle are indexed relative to this entry. The default value of *map_nr* is 1. Note that there is no space between `-c` and *map_nr* on the command line.

-m[*map_nr*] mapfile Instead of the index mode, which uses hardware colormap, an RGBA drawing mode is switched on. In this case, color indices are read from a file containing 256 triplets of bytes defining red, green, and blue. The integer *map_nr* specifies the 256-entry portion of the color table as for option `-c` (`-m` corresponds to `-m1`). Note that there is no space between `-m` and *map_nr* on the command line.

-M[*map_nr*] matfile Instead of the the index mode, which uses hardware colormap, an RGBA drawing mode with OpenGL lighting computations is used. In this case, the turtle parameter *color index* specifies an index to a material defined in a material file, which can be created by program `medit`. The integer *map_nr* specifies the 256-entry portion of the material table as for option `-c` (`-M` corresponds to `-M1`). Note that there is no space between `-M` and *map_nr* on the command line.

-w *xsize ysize* This option tells `cpfg` what should be the size of the opened drawing window. For example, `-w 1024 683` will open a window

suitable for saving image files with an aspect ratio of 3:2, appropriate for film recorders.

`-wp` *xpos ypos* Specifies the initial position of the window (of its top left corner). The automatic positioning of windows has to be switched on (see the IRIX window manager menu Desktop/Customize/Windows).

`-wt` *window_title* This option changes the title of the window (the icon name is still *cpfg*).

`-mb` In addition to popup menus, a menu bar is created at the top of the `cpfg` window.

`-pm` `cpfg` uses an X pixmap as a back buffer. The drawing is slower but handling of expose events is very fast plus the full 24 bit per pixel resolution can be used even in double-buffered mode.

`-sb` Single-buffer mode is set (this option takes precedence over the command `single buffer` in *Animation_file*). In the double-buffered RGBA mode, a single-buffer mode may be necessary to avoid dithering. This switch overrides switch `pm` and a pixmap is not used.

`-o` Menus do not use SGI overlay planes, which results in producing an expose event and redrawing the generated structure every time they overlap the drawing window.

4.3 Special working modes

`-g` Performs an off-screen rendering. A colormap or a material file must be specified (*i.e.* the off-screen rendering does not work with hardware colormaps). If the switch `a` is not included, `cpfg` will generate the string up to the last generation step (defined in *L-system_file*) and save particular files as specified by `-rgb`, `-ras`, *etc.* With switch `a`, all frames as defined in the animate file are saved. It is recommended to specify filenames as format strings, *e.g. plant%03d.rgb*.

ADD: Format strings are used in a few other places. It would be nice to have a section explaining the general setup, along with usage examples.

`-pipestrb` The program uses the off-screen mode and converts a binary L-system string file coming on *stdin* into a desired format (as specified by options `-rgb`, `-ras`, etc.), which can be output also to *stdout* (except for image formats) by specifying the word *stdout* as

a filename for a selected format. If -a is present, further generation (as specified by animate file) is performed. The option pipestrb can be used, for example, to pipe stored strings directly to rayshade and avoid keeping big rayshade files.

-C *communication_setup_string* Specifies connections to other processes in a distributed simulations. The communication setup string is a single string (with no spaces). Each connection starts with a symbol $-x$, where x is one of m, s, and c, followed by several parameters, divided by commas. The three types of connections are: *hoofs/cpfg3.2.examples/Distr.environ*

- -m for a master connection — cpfg controls the data exchange with the given program. The switch $-m$ is followed by a communication specification file for the given connection and a socket number (*-m,comm_spec_file,socket_num*). The same file and socket number has to be specified for the slave process.
- -s for a slave connection — the other program controls the data exchange. The switch $-s$ is followed by a communication specification file for the given connection, a socket number, and the name of the machine on which the master process is running (*-s,comm_spec_file,socket_num,master_machine,*). The same file and socket number has to be specified for the master process.
- -c connection to a process controlling the whole simulation. The program cpfg confirms its successful execution by sending a predefined character through this connection. The switch $-c$ is followed by a socket number and the name of the machine where the main process is running (*-c,socket_num,machine*).

See an example in Section 4.5 and more details on the distributed simulations in Section 8.5.

-S *socket_num* The program is able to process text commands (corresponding to menu items) coming through the specified socket (using program *command_client* — see Section 9.2). *hoofs/cpfg3.4.examples/socket.commands*

4.4 Output

-rgb *rgb_file* Specify name for the 24-bit SGI RGB image file. Note that any window that appears on top of the cpfg window when the snapshot is taken will be included in the image.

-ras *ras_file* Specify name for the color-index SGI image file. Note that any window that appears on top of the `cpfg` window when the snapshot is taken will be included in the image.

-tga *tga_file* Specify name for the Truevision Targa image file. Note that any window that appears on top of the `cpfg` window when the snapshot is taken will be included in the image. Available also in sun version (unlike rgb and ras formats).

-rle *rle_file* Specify name for the URT (Utah Raster Toolkit) run-length encoded image file. Note that any window that appears on top of the `cpfg` window when the snapshot is taken will be included in the image. Available also in sun version (unlike rgb and ras formats).

-ray *ray_file* Specify name for the rayshade output file[2].

-ps *ps_file* Specify name for the Postscript output file (see Section 7.2).

-iv *iv_file* Specify name for the Inventor output file.

-str *text_string_file* Specify the name of the file to which the string generated by `cpfg` will be output in text format (see Section 7.3).

-strb *binary_string_file* Specify the name of the file to which the string generated by `cpfg` will be output in binary format (see Section 7.3).

-homo strings are output after applying homomorphism.

-gls *gls_file* Specify the name of the file to which the structure generated by `cpfg` will be output in gls format (a set of OpenGL-like commands — see Section 7.4 for detailed description). So far only triangles of generalized cylinders and predefined surfaces are output.

-vv *vv_file* Specify the name of the file to which the bounding volume information will be output. See the `box` entry in Section 6.2.

-f *anim_path_format* Specify the directory and the format of file names for saving consecutive animation frames, when selecting `Begin Animate` from the animation menu (see Section 5.2). This switch is obsolete with the possibility to specify file names as format strings, e.g. *plant%03d.rgb*.

[2] Rayshade is a public domain ray tracer developed by Craig Kolb. `Cpfg` currently supports rayshade version 4.0, which is available at
`ftp://graphics.stanford.edu/pub/rayshade/rayshade4.0.tar.Z`.

A filename may be specified as a format string (e.g. *plant%03d.rgb*) and the number of the generation step is automatically inserted. This can be used for saving animations.

A filename may be specified also as "stdout" and the files are sent to a standard output. Often used in pipe mode (`-pipestrb`).

In the case of string output (both in the text or binary format), the specified filename is also used as the default name for the input of the string.

4.5 Usage examples

The most basic call to `cpfg` contains only an L-system and a view file:

```
cpfg fractal.l fractal.v
```

The next call includes an animation file, and specifies that Postscript output is to be written to the file `fractal.ps`. The user must choose the **Output postscript** item from the main menu to write to this file (Section 5.1).

```
cpfg -ps fractal.ps fractal.l fractal.v fractal.a
```

For large models, model generation can be made faster by specifying a larger initial string size on the command line. The default initial string size is large enough for most models. If a model is too large for the given string size, `cpfg` will reallocate the string and output the message `String is too long; reallocating`.

```
cpfg -s1000000 complexplant.l complexplant.v
```

In the next example `cpfg` uses color map number 3 and enters verbose mode. Output file names are specified for both rayshade and Postscript formats. Note that the options can be listed in any order, but that the L-system, view and animation files must be specified last.

```
cpfg -ray plant.ray -c3 -v -ps plant.ps plant.l plant.v
plant.a
```

In all previous examples, a hardware colormap was used for coloring the surfaces. A colormap can be specified on the command line.

```
cpfg -m plant1.map -m2 plant2.map plant.l plant.v
```

Instead of a colormap a material file can be included, improving the results of the shading calculations.

```
cpfg -M plant.mat plant.l plant.v
```

It is possible to generate output files without the drawing window, in an off-screen mode. In the example below, for all steps specified in the animate file a rayshade file will be output.

```
cpfg -g -a -ray plant%03d.ray -M plant.mat plant.l plant.v
plant.a
```

Instead of keeping potentially big rayshade files, it is possible to output only the L-system string and then pipe it through `cpfg` directly to rayshade.

```
cat plant015.strb | cpfg -g -pipestrb -ray stdout -M plant.mat
plant.l plant.v | rayshade -O plant015.rle
```

In the next example, the program monitors a specified socket and if a text command representing a menu item comes through the socket, it performs it as if the item was interactively chosen in the menu.

```
cpfg -S 1000 -M plant.mat plant.l plant.v plant.a
```

The following call runs an interactive simulations with the environment defined by an external process (as specified in the file *plant.e*)

```
cpfg -M plant.mat -e plant.e plant.l plant.v plant.e
```

The plant can be also executed as a part of a distributed simulation (although it will be very likely done by another program and not by the user). In this case, there are two connections to two environmental processes (whose executables are specified in files $plant1.e$ and $plant2.e$).

```
cpfg -C -m,plant1.e,200,-m,plant2.e,300 -M plant.mat plant.l
plant.v plant.e
```

5 User interaction

The left mouse button is used to rotate the model in the `cpfg` window. Holding down the left button and moving the mouse will cause the model to rotate in the direction of mouse movement. Moving mouse up and down while holding down the middle mouse button rescales the model in the `cpfg` window.

A menu, activated using the right mouse button, is provided for interaction with `cpfg`. The available menu items depend on command line options and the current state of the program. The menu controls re-reading of input files, regeneration of the image, output in a variety of formats, and the switch to and from animate mode. Once animate mode is selected, animation items are added to the menu. These items control the animation process.

With the menu bar present in the `cpfg` window (program option `-mb`), the program starts in animate mode and all menu items are accessible also from the two pulldown menus on the menu bar.

5.1 Main menu

The main menu is composed of the following items:

New Model Rereads the L-system and view files, generates a new string and interprets it to create a new image. The model is automatically centered in the window, or placed according to user-specified viewing parameters as described in Section 6.2.

New L-system Rereads the L-system file, generates a new string and interprets it to create a new image without modifying the view.

New homomorphism Rereads the L-system file containing homomorphism and reinterprets the current string using the new homomorphism (for more details see Section 6.1.7).

New View Rereads the view file and re-interprets the existing string to create a new image. The model is automatically centered in the window, or placed according to user-specified viewing parameters as described in Section 6.2.

New environment Restarts the process simulating the environment. This menu item appears only if a switch `-e` *environmentfile* is included in the command line (`cpfg` communicates with an environmental program). May cause problems if the environmental programs relies on the data from previous simulation steps.

Window size Allows the user to set the size of the output window.

Output Allows access to a sub-menu of output file formats:

Image Saves the image in the window in various image formats:

- **RGB** SGI rgb format
- **RAS** SGI colormap format
- **TGA** TrueVision Targa format
- **RLE** Utah Raster Toolkit image format

Rayshade rayshade 4.0 scene description file

Postscript Postscript scene description file

String current string in two formats:

- **text** simple text format
- **binary** internal representation

GLS format graphics library statements format

View Volume bounding volume information

Inventor SGI Inventor format object.

See Section 6 and Section 7 for more detail on file formats. Each of these entries invoke a sub-menu allowing the user to save using a default filename (the name of the L-system file with an appropriate extension or the file specified on the command line) or to save under a different name. In the second case, a special window is opened, allowing the user to browse through the current directory and select the output file or to type in the name. The modified filename is then stored and it appears as the default output name next time user want to output the structure using the same output format. In the case of string output (both in the text or binary format), the stored filename is also used as the default name for the input of the string.

Input Inputs data from the following formats:

String current string is read from a file of type:

- **text** simple text format
- **binary** internal representation

Both entries invoke a sub-menu allowing the user to input from a file with the default name (the name of the L-system file with an appropriate extension or the file specified on the command line — using `-str` or `-strb`) or from a selected file. In the second case, a special window is opened allowing the user to browse through the current directory and select the input file. The modified name is then stored and it appears as the default input name next time user want to input a string using the same format. The stored filename is also used as the default name for the output of the string.

Animate mode selects an animation mode, which has its own menu (see Section 5.2).

Exit Exits `cpfg`.

5.2 Animation menu

The animation process begins with input of parameters, including *first frame*, *last frame* and *step* from the animation file (see Section 6.3). If the animation file is not specified on the command line, the animation parameters are set to its defaults (*i.e.* the *first frame* is 1, the *last frame* is equal to the *derivation steps* specified in the L-system file, and the *step* is equal to 1).

In the animation mode, an animation menu becomes available. The menu contains all items as the main menu (except the item `Animate mode`). In addition, the following items become available, allowing the user to control the animation process:

Step Displays the frame resulting from the next *step* derivation steps. If this goes past the *last frame*, the *first frame* will be displayed.

Run Displays consecutive animation frames after each *step* derivation steps until the *last frame* is reached or passed.

Forever The same as **Run** except that when the *last frame* is reached, the animation returns automatically to the *first frame* and continues.

Stop Pauses the animation at the current frame.

Rewind Redisplays the animation from the *first frame*.

Clear Removes the current image from the window.

New animate Rereads the animation parameter file.

Begin Recording Gives access to a sub-menu allowing the user to initiate recording in a selected file format. The sub-menu is similar to the `Output` sub-menu. After picking **Run** from the animate menu, all subsequent frames are recorded, until **Stop Recording** is selected. For an L-system "`foobar.l`" and selecting the rgb output, for example, the frames are recorded by default as `foobar1.rgb`, `foobar2.rgb`, etc., unless the default file is set using a corresponding command line option (in this case `-rgb` *filename*). Note that the *filename* can be specified as a format string, *e.g. plant%03d.rgb*. Recording formats and default file extensions are the same as for Output files.

Stop Recording Stops the recording of animation frames.

Don't Animate leaves the animate mode and switches back to the main menu.

6 Input files

6.1 L-system file

The essential theoretical notions of L-systems are described in *The Algorithmic Beauty of Plants* [7]. The syntax is defined formally in Appendix A. Every L-system file specifies a derivation length, a list of symbols to be ignored (or considered) when context matching, an axiom, and a set of productions, which may be either deterministic or stochastic. A production consists of a predecessor and a successor with optional left and/or right contexts. The L-system can be defined over an arbitrary alphabet which does not contain the asterisk (*) or any separators (space, tab, etc.). Section 6.1.9 lists the symbols which have a graphical interpretation.

The typical file has the following format in the deterministic case:

`lsystem:` *label*
`derivation length:` *d*
`ignore:` *symbols*
`axiom:` *axiom*
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*
...
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*
`endlsystem`

The text in `typewriter font`, all spaces, and all special symbols must be entered as shown. The derivation length d must be a positive integer or zero. The $symbols$ list should include all symbols to be ignored while context matching. Alternatively, symbols to be considered when context matching can be specified following a `consider:` keyword. The $axiom$, and the $predecessor$ string in each production must be nonempty. The $successor$ strings in each production may be empty, in which case it must be represented by an asterisk (*). The $lcontext$ and $rcontext$ strings may be empty, in which case they can be represented by an asterisk (*) or left out completely along with the respective < and > symbols. A production may optionally include a condition C, which is a boolean expression using C-like syntax. The production will be used only if this expression evaluates to true. If a production has a condition, either $\{\alpha\}$ or $\{\beta\}$ or both may also be included. They represent lists of semicolon terminated statements expressed using C-like syntax. If $\{\alpha\}$ is given, it specifies statements to be executed *before* evaluating the condition C. If $\{\beta\}$ is given, it specifies statements to be executed *after* if the result of evaluating the condition is true. For example, the following is a valid production:

```
A(x,y) :  {z = x+y;} z>10 {n = cos(x-y);} --> A(n,z)
```

Note that all parameters are assumed to have real (floating point) values.

The end of an L-system specification is signaled by the `endlsystem` keyword.

For a stochastic L-system, a seed for the random number generator is also required and the typical file has the following, slightly modified, format:

`lsystem:` *label*
`seed:` *i*
`derivation length:` *d*
`ignore:` *symbols*
`axiom:` *axiom*
lcontext < *predecessor* > *rcontext* : { α } *C* { β } --> *successor* : p_1
lcontext < *predecessor* > *rcontext* : { α } *C* { β } --> *successor* : p_2
...
lcontext < *predecessor* > *rcontext* : { α } *C* { β } --> *successor* : p_n
`endlsystem`

The new first line specifies the integer seed i for the random number generator. Each production has a probability factor represented by the floating point value p associated with it. See *The Algorithmic Beauty of Plants* [7] page 28 for more information.

6.1.1 Variables

Variable names are defined as in C. There are two types of variables, float and character string, but character strings require special handling as they are passed by reference.

The variables are defined in the whole scope of an L-system. Often, if a variable is used in a production, it has to be defined in

There is a block structure controlling their scope. There is a special definition section for arrays, which are indexed as in C (Section 6.1.4).

ADD (Jim?): an explanation of the "external" statement that can now appear in the define section of cpfg. Basic syntax is the keyword "external" followed by a comma-separated list of variable names and array definitions. The arrays require their dimensions to be specified I believe.

6.1.2 Programming statements

There are three types of statements which can be included in L-system productions:

- The first type is represented by assignment statements of the form

  varname = expression;

  where variable names are specified as in C and $expression$ is an arithmetic expression. In this case the expression can also include local variables which have been assigned a value in previous assignment statements (within the same production), as in the following production:

  *A(y): {x=y/2; s=x*x;} s<5 {z=sqrt(y);}* → *B(z)C(z+1).*

  All variables have floating point values.

  In case of function $printf$, it is possible to omit the assignment part and ignore the value returned by the function:

{*a=a+1;printf("a=%f\n'",a);*}.

- The second type of statement includes conditional statements
  if (condition) $\{statmt_1$*; ...* $statmt_n$*;*$\}$
  and
  if (condition) $\{statmt_1$*; ...* $statmt_n$*;*$\}$ *else* $\{statmt_1$*; ...* $statmt_m$*;*$\}$
  where $condition$ is a logical expression and $statmt_i$ are statements.

- The third type of statement is represented by loop statements
  while (condition) $\{statmt_1$*; ...* $statmt_n$*;*$\}$
  and
  do $\{statmt_1$*; ...* $statmt_n$*;*$\}$ *while (condition);*
  The meaning of $condition$ and $statmt_i$ is the same as for conditional statements.

Statements α are performed every time the predecessor and left and right contexts match, before the condition is evaluated (even if it results in not applying the production) and before the matching production is found. Thus statements α can be applied to precompute expressions used in the condition. The β statements are performed after a condition is evaluated as true, but before the predecessor is replaced by the successor.

If a production does not have a condition, the empty condition $*$ has to be used:

lcontext $<$ *predecessor* $>$ *rcontext* : $*$ { β } $\rightarrow$ *successor*

Compared to the C syntax, the syntax of L-system programming statements has to follow these specific rules:

- Each assignment statement or a function call has to be terminated with a semicolon, even if it is a last statement in a block of statements (just before '}').
- Even if there is a single statement following the keyword if, $else$, $while$, and do, it has to be enclosed in curly brackets.
- Operators like $++$, $--$, $+=$, $-=$, $*=$, $/=$, *etc.* are not supported.

6.1.3 Global programming statements

Similarly to programming statements associated with productions operating on local variables, global statements, executed at specific points of the simulation, can be used to define global variables accessible in all productions. In the *cpfg* language, it is possible to define four blocks of statements, which are defined before the list of L-system productions, using the commands:

Start: $\{statements\}$	processed at the beginning of the simulation,
End: $\{statements\}$	processed at the end of the simulation,
StartEach: $\{statements\}$	processed at the beginning of each step,
EndEach: $\{statements\}$	processed at the end of each step.

The statements are of the same type as production statements introduced in the previous section. Variables used on the right side of an assignment statement in one of these four statement blocks are considered as global variables and can be accessed in any other block or production. A conflict of two productions accessing the same global variable at the same time is avoided because in the modeling program *cpfg*, the parallel rewriting process is captured by applying the productions sequentially, from left to right [5].

6.1.4 Arrays

The values of parameters of a plant model depend frequently on the order of an apex, or a branch or on another value such as the apex age or vigor. It is possible, for example, to have a separate production for each order with a successor using different values of growth parameters. But it is more effective to define an array of values and use only one production.

To define arrays, the *cpfg* language was extended by the command $define$ followed by a specification of all arrays used in the model: *hofs/cpfg3.0.features/-arrays*

define: { array $name_1[N_{1,1}]...[N_{1,D_1}] = \{v_{[0]...[0][0]}, v_{[0]...[1]}, ..., v_{[N_{1,1}-1]...[N_{1,D_1}-1]}\}$,
...
$name_n[N_{n,1}]...[N_{n,D_n}];$ }

The command can be placed anywhere before the list of productions. A single array is specified by its name $name_i$ and sizes $N_{i,i}...N_{i,D_i}$ for each of D_i array dimensions. The array can be initialized by including a list of all array values between a single pair of curly brackets. The first N_{i,D_i} values initialize array items $name_i[0]...[0][0], ..., name_i[0]...[0][N_{i,D_i}-1]$, next N_{i,D_i} values initialize array items $name_i[0]...[1][0], ..., name_i[0]...[1][N_{i,D_i}-1]$, and so on. Several arrays can be defined, with each specification separated by a comma and the last one terminated by a semicolon. The specification can extend over several lines.

In the following example, three one-dimensional arrays are defined and the first two are immediately initialized.

define: { array $GrowthRate[5] = \{1, 0.8, 0.7, 0.6, 0.5\}$,
$BranchingAngle[4] = \{60, 55, 55, 50\}$,
$ReceivedNutrients[5];$ }

6.1.5 Predefined functions

The following predefined functions can be included in L-system expressions:

`sin`(α), `cos`(α), `tan`(α) standard trigonometric functions. Argument α is in degrees.

`asin`(x), `acos`(x), `atan`(x), `atan2`(x, y) standard inverse trigonometric functions. Functions `asin` and `atan` return value of an angle between -90° and $+90^\circ$. Function `acos` returns a value between 0° and 90°. Function `atan2` returns arctangent of y/x in the range -90° to $+90^\circ$.

`exp`(x), `log`(x), `sqrt`(x), `fabs`(x), x`^`y other standard functions.

`floor`(x), `ceil`(x), `trunc`(x) rounding functions.

`sign`(x) returns 0 for $x = 0$, 1 for positive x, and -1 for negative values of x.

`srand`$(seed)$ initializes a random number generator used in all four following functions.

`ran`(x) generates floating point values uniformly distributed in interval $\langle 0, x)$.

`nran`$(mean, \sigma)$ generates random numbers with normal distribution with mean $mean$ and standard deviation σ. *hofs/cpfg3.0.features/-random*

`bran`(α, β) returns random values with beta distribution.

`biran`(n, p) generates random values with binomial distribution — how many out of n numbers are below p;

`stop`(n) stops animation. When the parameter n is equal to 1 and `Run` or `Forever` is selected from the menu, `cpfg` only draws the current string and continues the simulation. Otherwise, the simulation is stopped. *hofs/environment/-chiba/two_competing*

`printf`$("format\ \ string", var_1, var_2, ...)$ prints variables to standard output. All variables are of type *float*, thus the format string should contain only $\%f$ or $\%g$.

`fopen`$("file\ \ name", "type")$ opens a file specified by its name for input ($type = r$) or output ($type = w$). The function returns an index of the file, used in the functions below.

`fclose`$(file)$ closes the file $file$.

`fscanf`$(file, "format\ \ string", \&var_1, \&var_2, ...)$ allows to input data from an external file specified by file index $file$. *hofs/cpfg3.0.features/-file.input.Rootmap*

`fprintf`$(file, "format\ \ string", var_1, var_2, ...)$ outputs specified variables into the file $file$, using the format string. As in the case of function `printf`, the string should contain only $\%f$ or $\%g$. *hofs/environment/-MonteCarlo/test.runs*

`fflush`$(file)$ flushes the buffers associated with the file $file$.

6.1.6 Sub L-systems

It is often convenient to apply concepts of structural programming to L-system models and to divide bigger structures into independent parts. This allows the modeler to first describe the development of some parts of the plant, and then combine the pieces together in the complete model. Thus the design of a model is more efficient and it is possible to reuse productions simulating the growth of certain plant organs in other models.

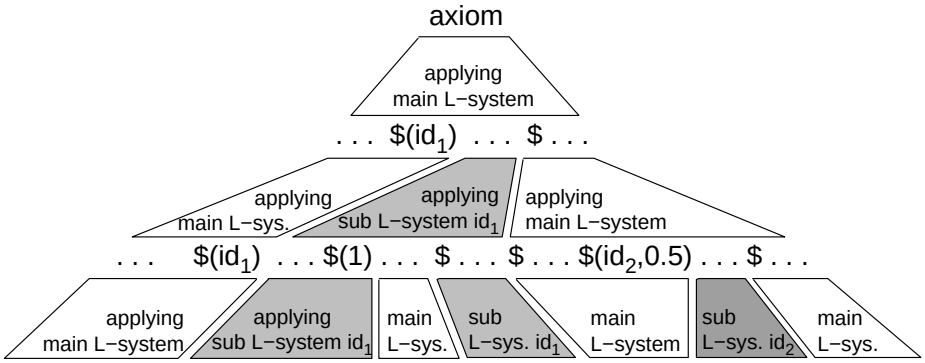


Figure 1: Example of a developmental sequence generated by an L-system with two sub L-systems

To this purpose, Hanan, in his thesis [1], extended parametric L-systems to include multiple sets of productions. The framework consists of a *main* L-system, controlling the development of the overall structure, and additional sets of productions, *sub L-systems*, which are invoked from the main L-system or from each other in a manner similar to calling subroutines in a program. Unlike subroutines, no values are returned to the main L-system upon a completion of a sub L-system.

In the *cpfg* language, the main L-system is the first set of productions in the L-system file. Each set of productions has assigned to it a unique index using command *Lsystem* on the first line of the specification of the main L-system (index 1) or a sub L-system. During the application of productions, module $\$(id)$ switches the control to an L-system with index id, *i.e.* all following modules are replaced by productions from this L-system. An optional second parameter can specify the scale applied to all geometry produced by the L-system with index id. Module $ without parameters returns the control to the original (parent) L-system (see Figure 1).

In the following example, productions for development of the main axis and development of lateral branches are separated.

hofs/cpfg3.0.features/-subLsystems

L-system 1

Lsystem: 1 /* Main L-system simulating growth of the main axis */

ω : $A(2,1)$

$p_{1,1}$: $A(l,o) \rightarrow !F(l)\ [\&(86)\mathbf{?}(2,R_b)B(l)\mathbf{?}]/(95)\ [\&(86)\mathbf{?}(2,R_b)B(l)\mathbf{?}]/(95)$
$[\&(86)\mathbf{?}(2,R_b)B(l)\mathbf{?}]/(95)\ [\&(86)\mathbf{?}(2,R_b)B(l)\mathbf{?}]/(95)$
$A(l*R_1, o+1)\ :\ 6-o$

$p_{1,2}$: $A(l,o):\ o>1\ \rightarrow !F(l)\ [\&(86)\mathbf{?}(2,R_b)B(l)\mathbf{?}]/(129)\ [\&(86)\mathbf{?}(2,R_b)B(l)\mathbf{?}]$
$/(129)[\&(86)\mathbf{?}(2,R_b)B(l)\mathbf{?}]/(129)\ A(l*R_1, o+1)\ :\ o$

endLsystem

Lsystem: 2 /* Sub L-system simulating growth of branches */
ω : $B(1)$
$p_{2,1}$: $B(l) \rightarrow !F(l)\ [+B(l * R_2)][-B(l * R_2)]\ :\ 0.7$
$p_{2,2}$: $B(l) \rightarrow !+(10)F(l)/(180)B(l * R_2)\ :\ 0.3$
endLsystem

Each branch apex B introduced in the main L-system by productions $p_{1,1}$ or $p_{1,2}$ is enclosed by modules ? and $ (presented in bold to make them more visible). In the next simulation step, the module $?(2, R_b)$, inserted just before the apex B, switches the control to the sub L-system and sets the scaling to R_b. Thus the module B is replaced by applying either production $p_{2,1}$ or $p_{2,2}$. The symbol $ behind module B returns the control to the main L-system. In the subsequent steps, all modules of the lateral branch, enclosed between modules $, are processed using productions of the sub L-system.

The axiom ω of the sub L-system does not affect the simulation, but it is useful when the sub L-system is being developed and tested (without the main L-system).

6.1.7 Homomorphism

hofs/cpfg3.0.features/-homomorphism

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

In `cpfg`, an L-system homomorphism is specified by productions that are placed at the end of an L-system between keywords `homomorphism` and `endlsystem`. During the interpretation of the L-system generated string, a matching homomorphism production is selected for each module in the string. The homomorphism image of a module is then defined by the successor of the matching homomorphism production. If there is no matching production, the homomorphism image of this module is the module itself. Productions with parameters or local programming statements operate similarly as L-system productions, *i.e.* the values of formal parameters in expressions are replaced by the real values of the module's parameters.

The resulting overall structure of an L-system with homomorphism is shown below:

`lsystem:` *label*
`derivation length:` *d*
`ignore:` *symbols*
`axiom:` *axiom*
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*
...
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*

`homomorphism [: [no] warnings]`
`seed:` s
`maximum depth:` d
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*

lcontext < *predecessor* > *rcontext* : { α } C { β } --> *successor*
...
lcontext < *predecessor* > *rcontext* : { α } C { β } --> *successor*
endlsystem

It is possible to repeatedly apply the homomorphism productions to the resulting homomorphism image of a module. To enable this operation, the keyword `maximum depth` should follow the keyword `homomorphism`. The value d then specifies the maximum depth of application of homomorphism productions (to avoid an infinite recursion). *hofs/cpfg3.4.examples/homomorphism/-recursive*

A warning is issued if the maximum depth is reached and it is possible to further apply the homomorphism productions (only in version 3.4 and higher). This warning can be avoided by specifying an optional keyword `no warnings` following the keyword `homomorphism`: *hofs/cpfg3.4.examples/homomorphism/-recursive*

```
homomorphism: no warnings
```

which is equivalent to:

```
homomorphism
```

(kept for backward compatibility). It is possible to use only the keyword `warnings` to specify that the warnings are switched on:

```
homomorphism: warnings
```

A context for a homomorphism production is defined as the context of the module in the L-system string, to which the homomorphism is applied, *i.e.* the homomorphism image of the modules on the left and right will not affect the context search. In the following example the context of production h_1 is used to draw only branches whose end point has y coordinate less than 3: *hofs/cpfg3.4.examples/homomorphism/-context*

L-system 2

$\omega:\quad A(1)$
$p_1:\quad A(o)\ o<6\ \rightarrow\ [+(20)F?P(0,0)A(o+1)][-(20)F?P(0,0)A(o+1)]$
homomorphism
$h_1:\quad F\ >?P(x,y)\ :\ y>3\ \rightarrow\ f$

Even if the homomorphism productions in the previous example were:

homomorphism
maximum depth: 2
$h_1:\quad F\ \rightarrow\ G?P(0,0)$
$h_2:\quad G\ >?P(x,y)\ :\ y>3\ \rightarrow\ f$

The context for the production h_2 would be the module $?P$ in the L-system string (with the properly set parameters) and not the module $?P$ introduced by the homomorphism production h_1.

The use of random values in a homomorphism is not recommended during an animation of the plant development, because the values used in one simulation step are

different from values used in another step and visible discontinuities may result. The resulting structure may change after each redraw, for example during rotations or window expose events. To prevent this from happening, it is possible to use a separate random number generator used only by the homomoprhims productions. This option is switched on by specifying the seed for this generator, using a keyword `seed:` following either the keyword `homomorphism` or the keyword `maximum depth`.

If sub L-systems are also used, each sub L-system has its own homomorphism, which has to be specified at the end of the sub L-system.

ADD: It would be nice to have a global homomorphism that would be shared by the main L-systems and all sub L-systems (not implemented yet).

If a homomorphism production is specified with the delimiter `-o>` (*an object production*) instead of `-->`, the operation of such a production is similar to the operation of a production with delimiter `-->`. During outputting the geometry to a rayshade file the object productions specify objects which should be instantiated. It is possible to specify (using the view file command `rayshade objects`) a format string for module's parameters that controls the precision used for differentiating between two objects created by the same modules with the same number of parameters. It is also possible to control whether even the turtle is considered when comparing two objects created by the same module with the same parameters (if the objects are different the second one is not an instantiation of the first one).

hofs/cpfg3.4.examples/homomorphism/rayshade.instancing

Note that the homomorphism productions are applied also during the environmental step to be able to properly determine the turtle parameters to be sent to the environmental program (with communication module $?E$) and to set the parameters of query modules $?P$, $?H$, $?L$, and $?U$ (see Section 6.1.9). Consequently, if you use a homomorphism production that is applied to one of the module $?E$, the module will not be sent to the environment. Similarly, a homomorphism production with $?P$, $?H$, $?L$, or $?U$ in the predecessor will cause that the values of the parameters of this module will not be set. For example, if there was an additional production

$$h_3: \quad ?P(x,y) \rightarrow @O$$

in the example above, the parameters of $?P$ would stay 0 and all branches would be drawn.

6.1.8 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 for homomorphism productions, the result of decomposition stays in the string.

Decomposition productions have to be specified after L-system productions and

hofs/cpfg3.4.examples/decomposition

before homomorphism productions (or at the end of an L-system if no homomorphism productions are included).

`lsystem:` *label*
`derivation length:` d
`ignore:` *symbols*
`axiom:` *axiom*
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*
...
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*

`decomposition`
`maximum depth:` d
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*
...
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*

`homomorphism`
`seed:` s
`maximum depth:` d
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*
...
lcontext < *predecessor* > *rcontext* `:` { α } *C* { β } `-->` *successor*
`endlsystem`

The syntax of decomposition productions is similar to homomorphism productions, in that there are decomposition productions specific for each sub L-system and the user can specify the maximum depth or whether warnings about reaching the maximum depth are printed (the only difference is that the productions with delimiter `-o>` have no special effect on the rayshade output). The command `seed` cannot be included at the beginning of decomposition, decomposition productions use the same random number generator as the L-system productions.

6.1.9 Interpreted symbols

During the visualization, the string of symbols is parsed from left to right and every time a special symbol controlling the turtle is encountered the function associated with the symbol is performed. Symbols with predefined interpretations are listed below.

Symbols with no parameters use default values specified at the beginning of the simulation. If a symbol has more parameters than those specified below, the additional parameters are ignored.

Changing position and drawing

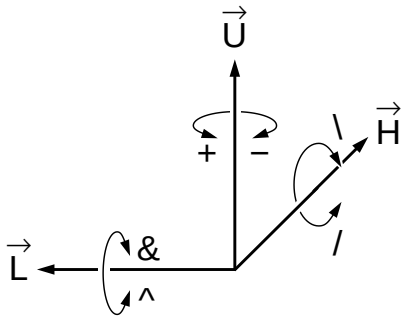


Figure 2: Controlling the turtle in three dimensions

F(d) Move forward a step of length d and draw a line segment from the original position to the new position of the turtle. If the polygon flag is on (see the symbols { and }), the final position is recorded as a vertex of the current polygon. If no parameter is given, the default step size 1 is used.

f(d) Move forward a step of length d without drawing a line. If the polygon flag is on, the final position is recorded as a vertex of the current polygon. If no parameter is given, the default step size 1 is used.

G(d) Move forward a step of length d and draw a line. If no parameter is given, the default step size 1 is used.

g(d) Move forward a step of length d without drawing a line. If no parameter is given, the default step size 1 is used.

@M(x, y, z) sets the turtle position to (x, y, z).

The global parameter `line style` specifies whether the line is drawn as a line, polygon, or a cylinder.

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

+(θ) Turn left by angle θ° around the U axis.

$-$(θ) Turn right by angle θ° around the U axis.

&(θ) Pitch down by angle θ° around the L axis.

$\wedge$(θ) Pitch up by angle θ° around the L axis.

\(θ) Roll left by angle θ° around the H axis.

/(θ) Roll right by angle θ° around the H axis.

| Turn around 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 so that H and U lie in a common vertical plane with U closest to up.

@R(hx, hy, hz,[ux, ux, uz]) Set the turtle heading to (hx, hy, hz) (if the vector is not normalized the program will automatically do it). If only the first three parameters are specified, the turtle up and left vectors are adjusted minimizing their rotation with respect to their previous orientation. Otherwise the next three parameters specify the turtle up vector (also this vector does not have to be normalized). In this case, the left vector is computed directly from the specified heading and up vectors. *hofs/environment/-soil/2d.no.avoiding/-animation*

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

If no parameter is given for the symbols `+`, `-`, `&`, $\wedge$, $\backslash$, and `/`, the value of the view file parameter `angle increment` (see Section 6.2) is used.

Changing turtle parameters

The following symbols change turtle parameters:

;(n) Increase the value of the current color index or material index by the `color increment`, or set to n if a parameter is given.

,(n) Decrease the value of the current color index or material by the `color increment`, or set to n if a parameter is given.

@;(n) Increase the value of the current color index or material index of the back side of a surface by the second parameter of command `color increment` (in the view file), or set to n if a parameter is given. Surfaces can have different colors or materials specified for each side only if the view file command `initial color` has two parameters. *hofs/cpfg3.0.features/-interpretation/-double_sides.surfaces*

@,(n) Decrease the value of the current color index or material of the back side of a surface by the `color increment`, or set to n if a parameter is given.

#(n) Increase the value of the current line width by the global parameter `line width increment`, or set to n if a parameter is given.

!(n) Decrease the value of the current line width by the global parameter `line width increment`, or set to n if a parameter is given.

@Tx($index$) Sets texture with index $index$ (the order of the texture specification in the view file). Index 0 switches off texturing. If a predefined bicubic surface has associated a texture index in the view file, its texture is fixed and cannot be changed by module $@Tx$. *hofs/cpfg3.0.features/-interpretation/textures*

@D($scale$) Sets the current turtle scale to $scale$. All subsequent geometry will be scaled by the specified value. The default value is set by the view file command `initial scale`.

@Di($factor$) Multiplies the current turtle scale by $factor$. If no parameter is specified the default value, set by the view file command `scale multiplier`, is used.

@Dd($factor$) Divides the current turtle scale by $factor$. If no parameter is specified the default value, set by the view file command `scale multiplier`, is used.

Modeling of structures with branches (Figure 3)

[Push the current state of the turtle (all its parameters) onto a pushdown stack.

] Pop a state from the stack and make it the current state of the turtle.

% The symbol % cuts the remainder of a branch. Whenever it is detected in the string during the generation process, it and all following symbols up to the closest unmatched right bracket] are ignored for derivation purposes, and will therefore disappear from the generated string. If an unmatched right bracket is not found, symbols are ignored until the end of the string. The symbols is ignored, if it is introduced by a homomorphism production.

hofs/cpfg3.0.features/-cut_module

$\%(par)$ Supports fragmentation. If the symbol $\%$ is found on the right side of any L-system production, a special interpretation step is performed after each generate step (if also the environmental pass is performed, these two passes are done together).

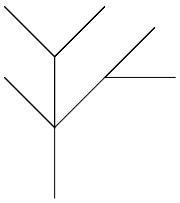
hoofs/cpfg3.0features/fragmentation

When the module is encountered during this pass, the following substring (up to the closing ']' at the same level or up to the next $\%(par)$) is moved to the end of the L-system string and it is preceded by a symbol $\%(par, turtle\_index)$, where $turtle\_index$ points to a special array of turtles. This array stores the turtle parameters as they were when the module $\%(par)$ was encountered. After the substring is moved to the end of the L-system string, every time a module $\%(par, turtle\_index)$ is encountered in the following interpretation steps, the turtle parameters are set to the values stored in the array of turtles under index $turtle\_index$.

The context searches are not passing over this module (if the parameter is present). Be careful when defining sub L-systems, because if the module $\%(par)$ appears within a set of $?(id)$ and $ (see below) the result after the cut is undefined. A production with $\%(par)$ as a predecessor will prevent the moving of the substring. The value of parameter par can be arbitrary.

@mc($flag$) Conditional cut. Operates as the module $\%$ (with no parameter) only if the value of $flag$ is equal to 1. Otherwise, it has no effect.

hofs/cpfg3.0.features/-cut_module/conditional_cut



$F[+F][-F[-F]F]F[+F][-F]$

Figure 3: Turtle interpretation of a bracketed string

Symbols used to create polygons along with `F` and `f`

{ Start a new polygon by pushing the current turtle position onto the polygon stack and set the polygon flag on. See also module $\{(type)$ in the section on generalized cylinders (below). *hofs/cpfg3.0.features/-interpretation/-polygons*

} Pop a polygon from the stack and render it. If no more polygons are on the stack, turn the polygon flag off. See also module $\}(type)$ in the section on generalized cylinders (below).

. Place the current state of the turtle on the polygon stack if the polygon flag is on.

Drawing circles and spheres

@o(d) Draw a circle of diameter d in the plane of the screen. If no parameter is given, the current line width will be used.

@c(d) Draw a circle of diameter d in the HL plane. If no parameter is given, the current line width will be used.

@O(d) Draw a sphere of diameter d. If no parameter is given, the current line width will be used. The spheres produced can be shaded even in the colormap mode, since a set of polygons approximating a sphere is generated using code from the widely available `sphere.c` file by Jon Leech (`leech@cs.unc.edu`).

Drawing parametric bicubic surfaces

$\sim$ Draw the predefined surface identified by the symbol immediately following the $\sim$ at the turtle's current location and orientation. The control points, geometry and neighborhood information for surfaces are read from surface specification files at the beginning of the simulation. *hofs/cpfg3.0.features*

@PS(*i,basis*) Initializes the four rows and columns of control points for an L-system defined surface i to $(0,0,0)$. The optional parameter *basis* specifies the type of patch as: *hofs/cpfg3.0.features/-interpretation/Lsys.-defined.surf.textured*

1. Bèzier,
2. B-spline,
3. Cardinal spline.

If no basis is given, the default, Bèzier, is used.

@PC(*i,r,c*) Assigns the current position of the turtle to the control point of the L-system defined surface i in row r and column c.

@PD(*i,s,t*) Draws the surface defined by the control points of surface i using s lines along the rows and t lines along the columns.

Drawing generalized cylinders

@Gs Start a generalized cylinder in the current turtle position. Equivalent to *{(1)* followed by '.' (see below). *hofs/cpfg3.0.features/-interpretation/-gen_cylinders*

{($type$) Start a generalized cylinder. The parameter type is one of the following:

1 an open curve consisting of Hermite spline segments (as in the case of `@Gs`);

2 a closed curve consisting of Hermite spline segments;

3 an open curve consisting of B-spline segments;

4 a closed curve consisting of B-spline segments.

If the parameter $type$ is 0 or is not specified, the points between a pair of curly brackets { and } specify a polygon (see above). The module does not specify the first control point.

@Gc($strips$) Specifies a control point on the central line of the generalized cylinder. The value of $strips$ specifies how many mesh strips are drawn between the control point and the previous one. The more strips are drawn, the smoother the generalized cylinder looks. If no parameter is given, one strip is drawn. Equivalent to .($strips$) (see below).

.($strips$) Equivalent to @Gc($strips$), only it can be used also for specifying vertices of a polygon (see the Section **Symbols used to create polygons** above). If the generalized cylinder is started using symbol {, a control point is also defined after each f or F (the same way as in the case of polygons — the number of strips is then set to the default value of 4).

@Ge($strips$) End a generalized cylinder. The parameter $strips$ controls the number of strips as for symbol `@Gc`.

}($type$) Finishes a generalized cylinder started by a module {($type_s$) The parameter $type$ has to match the value of $type_s$. If a new generalized cylinder is started before an old one is finished, the result is undefined (unless it is defined in a branch delimited by square brackets, *e.g.* {`(1)f(1)[{(3)f(1)}(3)]`}`(1)`).

@Gt($start, end$) Multiplicative parameter for the length of tangents of a Hermite curve that specify the axis of the generalized cylinder between two consecutive control points. The tangent lengths are equal to the distance between the two control points multiplied by the tangent coefficients (the default value is 1.2). *hofs/cpfg3.0.features/-interpretation/-gen_cylinders/-tangents*

@Gr($angle_1, length_1, angle_2, length_2$) Specifies the slope and length of two tangents of a Hermite curve which describes the change of radius of a generalized cylinder. The command defines the angle of the tangent and its length for a segment finishing at the next specified control point and for the following segment starting at the same point. *hofs/cpfg3.0.features/-interpretation/-gen_cylinders/radii*

The angle is defined with respect to the axis of a straight segment of a unit length, thus the real slope of the radius may not correspond to the set value for curved segment or segment of a different length (the second problem can be avoided by using module `@Gr(1)`). In addition if you increase the length of the tangents of the axis too much (by module `@Gt`) the strips close to the control points will be wider that the strip in the middle and the angle of radius tangents will be skewed as well.

As a default or when both lengths are equal to 0, the radius at the control points is set so that it is linearly interpolated along the segment (if only one length is set to 0, the tangent at the point is set as if the radius was interpolated linearly).

@Gr($flag$) Switches on (*flag*=1) or off (*flag*=0) an automatic adjustment of radius tangents for segments of a non-unit length. If the flag is 1, the tangents are defined for a segment of a unit length and then stretched onto the segment of a non-unit length, thus the specified tangent angles do not correspond to the real angles of the tangents. As a default, tangents are not adjusted after the stretching.

@#($contour\_id$) Sets the contour specified by $contour\_id$ as the current contour for generalized cylinders. Contours are specified in the view file (see the command `contour` in Section 6.2). A contour with `id` 0 is the default circle. Unlike in case of textures or tropisms, contour *id* is specified in the view file for each contour separately and it does not depend on the order of commands `contour`. *hofs/cpfg3.0.features/-interpretation/-gen_cylinders/-contours*

@!($polygons$) Sets the number of polygons around a generalized cylinder or a cylinder that is represented by F or G.

Changing tropisms parameters

@Ts($index$,$value$) Set elasticity parameter of tropism with index $index$ to $value$. Index is given by the order of the tropism specification in the view file (starting with 1). *hofs/cpfg3.0.features/-interpretation/-tropisms*

@Td($index$[,$value$]) Decrease the elasticity parameter by the default elasticity increment specified in the view file or by the given value $value$.

@Ti($index$[,$value$]) Increase the elasticity parameter by the default elasticity increment specified in the view file or by the given value $value$.

@Tp Prevent twist. This command adjusts the turtle's up and left vector to minimize the twist [3]. This command operates locally, *i.e.* it adjusts 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 the symbols `/` or \ is nullified. In such cases it is necessary to add the symbol `@Tf` to force the twist. This command operates localy, *i.e.* it prevents twist only for symbols `/` or \ to the left of `@Tf`.

Symbols for Sub-L-systems

?(*id,scale*) Causes the generator to save a reference to the current L-system on a stack and to use the list of productions from the sub-L-system identified by *id* during subsequent production matching and application. During interpretation, the current scale is saved on a stack and the structure resulting from interpretation of the generated substring is scaled by *scale*. *hofs/cpfg3.0.features/subL-systems*

$ End the sub-L-system and return to the previous set of productions and scale.

Query and communication symbols

?P($x, y[, z]$) queries the current turtle position. If the module is present in any L-system production, an interpretation step is performed after each generate step, when productions are applied. The string is thus interpreted even if `cpfg` does not draw to the window. During the interpretation, the two or three parameters of the module are set to the x, y, or x, y, and z coordinates of the current turtle position, respectively. These parameters then can be accessed in the following generate step and affect the selection of productions (see the definition of environmentally-sensitive L-systems in [6]). *hofs/cpfg3.0.features/query_module*

?H($x, y[, z]$) queries the current turtle heading vector (similarly as `?P`).

?L($x, y[, z]$) queries the current turtle left vector (similarly as `?P`).

?U($x, y[, z]$) queries the current turtle up vector (similarly as `?P`).

?E($x_1, \ldots, x_m$) module $?E$ (communication module) is used both to send and receive environmental information represented by the values of parameters $x_1, \ldots, x_m$. Specifically, parameters $x_1, \ldots, x_m$ act as an interface between the plant and the environment, simulated by an external process. They can be set by the plant model and transferred to the environment or set by the environment and transferred to the plant model (see the definition of Open L-systems in [2, 4]) *hofs/environment/...*

Miscellaneous commands

@L("Label") Prints the "label" in the drawing window at the current turtle location using the font specified in the view file. It is also possible to specify a printf-like format string and print out values of subsequent parameters (*e.g.* `@L("a=%g",a)`). *hofs/cpfg3.0.features/labels*

@S("any system call") Will make the system call when interpreted.

@I("rayshade object"[, $scale$]) Includes a rayshade objects with a given name, located at the current turtle location, and scaled by a given value (only for a rayshade output). The second parameter is optional.

@J($size_1, size_2, size_3$) As a default, all objects output into a rayshade file are enclosed in one grid. To be able to create more grids, tightly enclosing each plant, for example, the module `@J` closes the current grid and starts a new grid of a given size (in number of voxels). This module is interpreted only during the rayshade output. Usually, a value of 20 for the longest object dimension is sufficient. The shorter dimensions then can be reduced accordingly (this has to be done by the user). If the object dimension are not known, values $20 \times 20 \times 20$ would work.

If a module starting with a letter @ is not one of the recognized interpreted modules, a warning message is issued during the interpretation of the string. An exception are modules *@Z* and *@Y*, which are used for controlling the tropism elasticities in a program for interactive editing of L-system generated strings (currently in development).

6.2 View file

A view file contains drawing, viewing, and rendering parameters as well as the names of surface specification files for any surfaces to be included in the image. The format of the view file is given below. All text in `typewriter font`, special symbols and all spaces should be entered as shown. Unless stated otherwise, the symbols $x, y,$ and z represent floating point numbers, i represents an integer, id represents a single character, and other text in *italics* represent character strings. Comments can be included using standard C notation: `/* ... */`. Many of the parameters in this file have default values, and can be omitted, but it is good practice to have everything in the file. This makes it easier to change default values because the appropriate keywords are already in the file (also, it makes it easier to change parameters with control panels).

Note that the following commands are processed in the order they are specified in the view file. Thus if there are two commands controlling the same parameter, the second command takes precedence. This does not apply to commands such as `light`, `texture`, and others that specify a new set of parameters every time the command occurs.

6.2.1 Setting turtle's parameters

Line Contents **Comments**

`angle factor:` x $360°/x$ is the angle increment associated with the `+`, `-`, `&`, `^`, `\`, `/` and `|` symbols.

`angle increment:` x Set the angle increment associated with the `+`, `-`, `&`, `^`, `\`, `/` and `|` symbols to x. The commands `angle increment` and `angle factor` are alternatives and the last one appearing in the file will be used.

`initial color:` i_1 [i_2] number between 0 and 255 specifying the initial value of the index to the color map or a set of materials. The second number, if present, specifies the index of the color or material of the back side of the surface. The program `cpfg` then considers two different colors/materials for each surface.

`color increment:` i_1 [i_2] number specifying the color or material index increment associated with the `;` and `,` for the front index and `@;` and `@,` for the back index.

`initial line width:` x [| pixels | shaded] the number x represents initial line width in the specified line style. If no string is listed after the number, then `F`s and `G`s are drawn as flat shaded polygons with a width in world units. The width of line in this case is rescaled when the `cpfg` window is resized. If `pixels` or just `p` is listed, flat shaded lines are drawn with their width in pixels (or screen units). If `shaded` or simply `s` is listed, lines are drawn as shaded cylinders in world units. In versions 3.0 and higher, line style should

be set by command `line style` (see below in Section **Lines, surfaces, and generalized cylinders**).

`line width increment:` *x* a number specifying the line width increment associated with the symbols # and ! with units taken from the initial line width specification.

`initial scale:` *x* the parameter x specifies the initial scale factor associated with the turtle (the default is 1). All geometry will be scaled by this factor. This initial value can be modified by modules `@D`, `@Di`, and `@Dd`.

`scale multiplier:` *x* modifies the default value (1) of the multiplicative factor by which the turtle scale is multiplied or divided, when module `@Di` or `@Dd` is interpreted.

6.2.2 Setting the view

`viewpoint:` *x,y,z* $x, y,$ and z coordinates of the view point in world space[3].

`view reference point:` *x,y,z* $x, y,$ and z coordinates of the view reference point in world space[3].

`twist:` *i* tenths of degrees to rotate the image on the screen.

`projection:` *type* *type* identifies the desired projection, either `parallel` or `perspective`. Perspective viewing mode is the preferred mode to use if you intend to save a rayshade format object since rayshade also uses perspective viewing. Auto-centering and auto-scaling work only in parallel mode.

`viewing angle:` *x* the viewing angle of perspective projection (the default is 45°). It is ignored in parallel projection.

`front distance:` *x* the distance from the viewer to the front clipping plane in perspective projection or the position of the clipping plane with respect to the viewpoint in parallel projection (thus a negative value has to be used). Note that modifying scale factor (see below) in perspective projection moves the viewer closer or farther from the view point and the front distance has to be adjusted.

`back distance:` *x* the distance from the viewer to the back clipping plane in perspective projection or the the position of the clipping plane with respect to the viewpoint in parallel projection. Note that modifying scale factor (see below) in perspective projection moves the viewer closer or farther from the view point and the back distance has to be adjusted.

[3] As described in the SGI Graphics Library Programming Guide.

`scale factor:` *x* a parameter indicating the size of the final image on the screen. A value of 1.0 corresponds to full size. In perspective projection, the scaling amounts to moving the viewer closer or farther from the view points, which may require adjustments in front and back distance.

`box:` `x:` *xmin*, *xmax* `y:` *ymin*, *ymax* `z:` *zmin*, *zmax* sets a bounding box for the model. The view is adjusted so the whole bounding box is visible (effective only in parallel projection).

6.2.3 General drawing parameters

`shade mode:` *i* an integer defining the type of rendering to be applied:

1. simple fill,
2. interpolated fill,
3. Gouraud shade,
4. B-spline,
5. closed B-spline,
6. two sided,
7. wireframe.

This command is kept only for backward compatibility. Use `render mode` instead.

`render mode:` *mode* where *mode* defines one of the following render modes:

fast similar to mode *filled* (see below), only spheres and disks are drawn in wireframe.

wireframe the wireframe of all objects is visualized. If the image is output to a postscript pairs of neighboring triangles are visualized as a single polygon to reduce the number of lines.

filled all polygons representing a surface have the same color associated with the surface. If materials are specified the diffuse color is used.

interpolated similar to mode *filled*. If the color or material at the beginning of a straight line or cylinder (using modules `F`s and `G`s) is different from the color at the end, the two colors are interpolated along the line. Similarly, the color of L-system defined polygons is interpolated, if the colors at different vertices are different.

flat in this mode, the color of each polygon representing surfaces, lines, or generalized cylinders is determined according to the position of the polygon with respect to the light. If materials are specified, the color is determined according to the material specification, using a single normal for the whole polygon. Otherwise, the colormap is used. In the

case of cylinders or generalized cylinders, the color of the polygon is chosen from interval $[col - diff\_refl, col + diff\_refl]$, where col is the color index associated with the surface and $diff\_refl$ is a range defined using command `diffuse reflection` (see below). The color is chosen according to the position of the polygon with respect to the direction towards the first light source (other sources are ignored). In the case of surfaces and tsurfaces, the color selection is more complicated (see command `surface reflection` below).

shaded similar to mode *flat*. If materials are specified, the normal for each polygon representing a surface can be different at each vertex of the polygon, resulting in a smooth shading. If colormap is used, a color is computed for each vertex of the polygon (see the commands `diffuse reflection` and `surface diffuse` below).

`z buffer:` *flag* a string identifying whether hidden surface elimination (using z buffer) should be provided (`on`) or not (`off`).

`cue range:` *x* a number specifying the range of color indices used for depth cueing. A value of 0 indicates no depth cueing. Usual values of *n* are 10 to 100. Depth cueing is not used in versions 3.0 and higher.

`font:` *Xfont* *Xfont* specifies the font type to be used in $@L$ interpretation using the X font specification. If the font is not found or not specified, the default is *-*-courier-bold-r-*-*-12-*-*-*-*-*-*-*-*.* *hofs/cpfg3.0.features/labels*

`interpretation past %:` *flag* *flag* equal to `on` (default) allows the turtle to interpret past symbol % which in subsequent step cuts a substring. When the *flag* is set to `off`, the symbols after % are not interpreted.

`interpretation step:` *i* an integer value *i* specifies number of interpreted symbols between an X event is checked. The interpretation during rotation or after selecting *New model*, *New L-system*, *New view* or *New Homomorphism* from the menu can be interrupted by an X server event. This allows one, for example, to quit the program before the drawing is finished, to rotate much quicker — just a part of the string is drawn (depending on the machine speed and value of *i*), or to reduce the number of redraw events when the window is resized or exposed several times in a row. Setting *i* to -1 switches off this feature, *i.e.* all modules are interpreted without checking for the next event. *hofs/cpfg3.0.features/-interpretation/-interpr.step*

`rayshade objects:` $format$ [`turtle` $flag$] controls the output of instantiated objects into rayshade file. If you specify a homomorphism production with delimiter `-o>` instead of `-->`, during the rayshade output the predecessor will be instantiated if it appears again (if it has the same parameters and possibly also the same turtle parameters). The format string controls the precision of object parameters (used for differentiating between two objects *hofs/cpfg3.4.examples/homomorphism/-rayshade.instancing*

created by the same modules with the same number parameters). For example, if the format is set to $\%.2f$, the precision of two decimal points is used in comparisons ($\%f$ or $\%g$ results in the full precision comparisons).

The parameter $flag$ is equal either to $considered$ or $ignored$ and it controls whether even the turtle is considered when comparing two objects created by the same module with the same parameters (if the objects are different the second one is not an instantiation of the first one).

`rayshade scale:` $scale$ specifies a scale factor which is applied to the rayshade objects output by `cpfg`. This command can be used for scaling up and down specific plants generated by different L-systems in different scales. Note: it is usually better to use the parameter turtle scale (see command `initial scale`).

6.2.4 Lines, surfaces, and generalized cylinders

`line style:` *style* specifies how the lines (represented by modules `F` or `G`) are drawn. Parameter *style* is one of the following:

- `pixel` flat shaded lines are drawn with their width in pixels;
- `polygon` lines are drawn as flat shaded polygons with a width in world units;
- `cylinder` lines are drawn as cylinders with the width specified in world units.

`tapered lines:` *flag* controls whether lines or cylinders are drawn tapered or not (*flag* is equal to `on` or `off` — the default is `on`).

`polygonization level:` *n* determines the level of detail used in generating the polygons for spheres and cylinders. For stems, for example, there is 2^{n+1} polygons around the circumference. A high value, such as 4, will generate very smooth surfaces, but take longer to display. A lower value, such as 1 — the lowest, produces very rough approximations to these surfaces. If this line is not specified, the default value is 2. This command is kept only for backward ompatibility. Use `contour sides` instead.

`contour:` *i file* defines a contour with integer id *i* specified by a set of 2d or 3d control points read from the file *file*. For more details, see Section 6.4.2.

hofs/cpfg3.0.features/-interpretation/-gen_cylinders/-contours

`contour sides:` *n* determines the level of detail used in generating the polygons for spheres and cylinders (this initial value can be modified by module `@!`). In the case of cylinders, n ($n > 3$) polygons around circumference is drawn. For spheres, the closest upper power of two is used. If you want to have smooth connections between cylinders and spheres for small values of n, use a power of 2. If this command or command `polygonization`

`level:` is not specified, the default value is 8. Make sure this command is not followed by command `polygonization level` later in the view file, because then the parameter could be changed by the second command.

`surface:` *id name.s x* [*s t*] [tex] the character used to identify the surface, a string containing the file name of the surface specification file (see Section 6.4.1), and a surface scaling factor. The parameters s, t, and tex are optional. If parameters s and t are included, they specify the level of detail used when drawing patches. Patches are drawn using s polygons along the rows and t along the columns. If s and t are not specified, they default to 5. Several surfaces may be specified in this manner.

The last parameter (tex), if present, specifies a texture associated with the surface. This value takes precedence over the texture index associated with the turtle during the interpretation and all instances of this surfaces will have the same texture. It is better not to include this parameter and set the texture inside the L-system. Note that the parameter tex can be present even if the couple of parameters s and t is omitted.

`line:` *id name.s x* the character used to identify the line to be drawn, a string containing the name of the surface specification file, and a surface scaling factor.

`tsurface:` *id name.ray s* the character used to identify the surface, a string containing the name of a file using rayshade-like file format, and a surface scaling factor s. The file should contain a set of triangles with 6 or 8 values per vertex, specifying vertex point, vertex normal, and optionally texture coordinates (see Section 6.4.3). *hofs/cpfg3.0.features/-interpretation/-tsurfaces*

`twist of cylinders:` *flag* As a default, generalized cylinders are drawn in such a way that their twist is minimalized. If the twist is desired, set *flag* to `on`. *hofs/cpfg3.0.features/-interpretation/-gen_cylinders/twist*

`background scene:` *list* *list* is a list of file names (separated by a space, comma, or semicolon). Each file contains a set of OpenGL-like graphics commands (see Section 6.4.5) which specify additional objects drawn after the L-system generated string is interpreted. *hofs/cpfg3.0.features/-interpretation/-gen_cylinders/-background_scene*

6.2.5 Color-map mode: Colors and lights

`light direction:` *x,y,z* x, y, and z coordinates of the vector indicating the direction of light for shading purposes. This command should be used only if the program is running in the colormap mode. In the material mode, use command `light`.

`ambient light:` *red, green, blue* the ambient light specified as red, green and blue components. This command is effective only in version 2.7 and lower.

`diffuse reflection:` *i* an integer number indicating the range of colors chosen for lighting a shaded surface (effective only in colormap mode). The surface color col is varied within the interval $[col - i, col + i]$ to achieve a color variation due to the different orientation of polygons representing the surface with respect to the direction of the light source (only of the first light source if more then one source is specified). The color of a polygon representing a cylinder or generalized cylinder is chosen in the following way. If col is the color index associated with the cylinder, i is the diffuse refection coefficient, $\vec{N}$ is the normal of the polygon, and $\vec{L}$ is the direction towards the light source, the resulting index is:

$$index = col + i \cdot \vec{N} \cdot \vec{L}.$$

`surface ambient:` *x* a number between 0 and 1 indicating the amount of ambient light present for shading bicubic surfaces and tsurfaces. This command is effective only if the program is running in the colormap mode. In the material mode, materials specify ambient light for surfaces. See the following command for the description of computing the resulting color.

`surface diffuse:` *x* a number between 0 and 1 indicating the amount of diffuse light present for shading bicubic surfaces and tsurfaces. This command is effective only if the program is running in the colormap mode. In the material mode, materials specify diffuse light for surfaces. The color of a polygon representing a surface is chosen in the following way. If col is the color index associated with the surface, int is the intensity of the color (int = $(col/64) - floor(col/64)$), amb is the predefined ambient intensity, $diff$ is the predefined diffuse intensity ($diff = x$), $\vec{N}$ is the normal of the polygon, and $\vec{L}$ is the direction towards the light source, the resulting index is:

$$index = 64 \cdot int \cdot (amb + diff \cdot abs(\vec{N} \cdot \vec{L})).$$

`background color:` *red, green, blue* the background color specified as red, green and blue components. In `cpfg` version 3.0 and above, this command is ignored. The background color is then either the colormap color with index 0 (in the given set of 256 colors — controlled by the command line parameter `c`) or the emission color of the material with index 0.

6.2.6 Material mode: Lights and textures

`light:` $subcommand_1\ subcommand_2$ ... sets a light source. The subcommands are:

- `O: x y z` origin of a point light source (the default light source is a point source, located at (0,0,1));
- `V: x y z` vector specifying a directional source;

`A: r g b` ambient (default 1 1 1);

`D: r g b` diffuse (default 1 1 1);

`S: r g b` specular (default 1 1 1);

`P: x y z e c` specified a spotlight with the direction (x, y, z), exponent e , and cutoff angle c (default 0 0 -1 0 180);

`T: c l q` attenuation factors (constant, linear, and quadratic) (default 1 0 0).

More than one light can be specified by including several commands `light` into the view file.

`texture:` $subcommand_1$ $subcommand_2$ ... defines a texture mapped on surfaces, cylinders, conses, and generalized cylinders (not disks and spheres). The subcommands are:

hofs/cpfg3.0.features/-interpretation/-textures/...

`F: image` specifies the image file name (a necessary subcommand). The image width and height is clamped in such a way that the image size is ($2^m \times 2^n$).

Currently, it is possible to specify rgb, rle, and tga images (with the proper extension).

`H: filter` for textures with texels bigger than image pixels. The parameter $filter$ is either $linear$ or $near$ (only l or n can be used). When set to $linear$ texture image is smoothed, while setting to $near$ makes the texture pixels visible.

The default is $near$.

`L: filter` for textures with texels smaller than image pixels The parameter $filter$ is either $linear$ or $near$ (only l or n can be used). When set to $linear$ more texture pixels are used to compute the given pixel, while for $near$, just one texture pixel is used to compute the given pixel (which may result in aliasing).

It is also possible to use mipmaps in which case 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). There are four modes of operation when selectin a proper textel pixel:

mnn take the nearest mipmap image and the nearest pixel in this mipmap. Produces some artefacts, visible especially when moving object around or scaling it, but it is the fastest.

mln take the nearest mipmap image and the linearly interpolate between neighboring pixels (still produces some artefacts).

mnl take the nearest pixels in both best choices of pixmaps and interpolate between the values.

mll linearly interpolate between neighboring pixels in both best choices of pixmaps and interpolate between the values. Produces the best result, but may be slower.

If just m is used, the mll mode is selected.

The default is $near$.

`E: mode` controls the way the texture is combined with the surface colors[4]. The parameter $mode$ is one of the following:

modulate `cpfg` multiplies the surface color with the texel color;

decal only the texel color is taken and the surface is not shaded;

blend interpolates between surface and texture color using the color index value of the surface (only in colormap mode).

The default is $modulate$ (only m, d, or b can be specified):.

`S:` when present, the surface texture is mapped per surface not per patch. The default is mapping per patch, *i.e.* texture coordinates are derived from s and t coordinates of the Bèzier patch representing the surface (both s, and t vary from 0 to 1). In case of mapping per surface, first the surface boundaries are found and then the texture is mapped into $z = 0$ plane with respect to the computed boundaries.

`R: ratio` defines the aspect ratio of a texture mapped on cylinders and generalized cylinders. The default is 1. A value greater than 1 will cause the texture to be more stretched along the cylinder.

More than one texture can be specified by including several commands `texture` into the view file.

6.2.7 Tropisms

`tropism direction:` *x,y,z* $x, y,$ and z coordinates of the vector indicating the direction toward which branches tend to bend. This command is kept only for backward compatibility. Use the command `tropism` instead.

hofs/cpfg3.0.features/-interpretation/-tropisms

`initial elasticity:` *x* a value specifying the susceptibility of a branch to bending. This command is kept only for backward compatibility. Use the command `tropism` instead.

`elasticity increment:` *x* the value used to increment or decrement the elasticity associated with the _ symbol. This command is kept only for backward compatibility. Use the command `tropism` instead.

[4] See "The OpenGL Programming Guide", Chapter 9, Section *Modulating and Blending.*

`tropism:` $subcommand_1$ $subcommand_2$ ... sets tropism parameters. The subcommands are:

- `T: x y z` tropism vector (must be present);
- `A: ang` angle (in degrees) with respect to the tropism vector that segments are trying to reach (for example, the angle of $90°$ corresponds to diatropism). The default is $0°$.
- `I: int` intensity (global intensity of the tropism — default is 1);
- `E: ela` initial elasticity (default is 0);
- `S: step` elasticity step (default is 0).

`torque:` $subcommand_1$ $subcommand_2$ ... sets parameters of a movement that adjust rotates segments around their heading without modifying the heading orientation. The subcommands are the same as for command `tropism`, except that subcommand `A:` is ignored.

6.3 Animation file

An animation file contains parameters controlling frame by frame display of images for animation purposes.

Line Contents **Comments**

`double buffer:` *flag* specifies whether double buffering is `on` or `off` during animation. The default is `on`. In `cpfg` version 3.0 and higher, the command line setting of single- or double-buffering takes precedence, because buffering has to be set at the point of execution (using command line parameters) and cannot be changed afterwards. The only effect the command `double buffer` has is to set single-buffering even if the program starts with two buffers.

`clear between frames:` *flag* specifies whether screen clearing between frames is `on` or `off`. The default is `on`.

`scale between frames:` *flag* If the flag is `on`, the view is adjusted (in parallel projection only) so the whole structure fits into the window (before the scaling is applied — see command `scale` in the view file). The default is `off`.

`new view between frames:` *on/off* If the flag is `on`, the view file is reread after each simulation step. Consequently, the view, textures, and all parameters specified in the view file are updated. Used, for example, for updating a background scene or a texture used for visualizing the environmental field[5]. The default is `off`.

hofs/environment/-soil/2d.no.avoiding/-animation

`swap interval:` *i* minimum time (in tenth of a second) between swapping of buffers in double buffer mode. The time is measured from the moment `cpfg` begins to draw the frame to the moment it begins to draw the next frame. If it takes longer to draw a frame, the delay between frames is then longer. The default is 1.

`first frame:` *i* derivation step of the L-system string to be interpreted as the first frame. The default is 1.

`last frame:` *i* derivation step of the L-system string to be interpreted as the last frame. The default is the number of derivation steps (specified in the L-system file).

`step:` *i* number of derivation steps between drawing (and recording) of frames. It defaults to 1.

[5] The memory allocated by `cpfg`'s (the resident size) increases with each New View. This increase may be significant in animations in which the new view is invoked after each animation step. See Section 11.

`frame intervals:` $frame_1, frame_2, from_1 - to_1, from_2 - to_2$ `step` $step_1$ ...

Allows the user to select frames or change the step during an animation. The command is followed by:

- a list of specific frames and/or
- by ranges of frames without specifying the step (thus step of 1 is used) and/or
- by ranges of frames with given step,

all divided by commas.

Example:

`frame intervals:` 1, 3-5, 8-12 `step` 2, 25

In addition, every time a range is specified, it is possible to change the scaling or rotate the given object by a certain amount after each frame, using commands:

hofs/cpfg3.0.features/-interpretation/-animate

`rotate` rx ry rz rotates by angle rx (in degrees) around axis x, angle ry around axis y, and angle rz around axis z;

`scale` sx sy sz scales by values sx, sy, and sz.

There can be only one command `rotate` or `scale` present for a single range.

Examples:

`frame intervals:` 1-99,100-150 `rotate` 1.5 0 0

`frame intervals:` 1-99,100-150 `scale` 0.9 0.9 0.9

If the command `frame intervals` is specified in the animation file (regardless the order), it takes precedence over the commands `first frame`, `last frame`, and `step`.

The command in the file can be in an arbitrary order. The file is not preprocessed, thus comments may cause problems (at least a warning will be issued).

6.4 Other input files

6.4.1 Surface specification file

A surface specification file details a bicubic surface in Bèzier form composed of an arbitrary number of patches. The file has the following format, where x, y, and z are real values, i is an integer value and the remaining strings in *italics* represent text strings.

hofs/cpfg3.0.features

$x_{min}\ x_{max}\ \ y_{min}\ y_{max}\ \ z_{min}\ z_{max}$
`CONTACT POINT X:` x `Y:` y `Z:` z
`END POINT X:` x `Y:` y `Z:` z
`HEADING X:` x `Y:` y `Z:` z
`UP X:` x `Y:` y `Z:` z
`SIZE:` x
patchname
`TOP COLOR:` i `DIFFUSE:` x
`BOTTOM COLOR:` i `DIFFUSE:` y
`AL:` $patch1$ `A:` $patch2$ `AR:` $patch3$
`L:` $patch4$ `R:` $patch5$
`BL:` $patch6$ `B:` $patch7$ `BR:` $patch8$
$x_{11}\ y_{11}\ z_{11}\ \ x_{12}\ y_{12}\ z_{12}\ \ x_{13}\ y_{13}\ z_{13}\ \ x_{14}\ y_{14}\ z_{14}$
$x_{21}\ y_{21}\ z_{21}\ \ x_{22}\ y_{22}\ z_{22}\ \ x_{23}\ y_{23}\ z_{23}\ \ x_{24}\ y_{24}\ z_{24}$
$x_{31}\ y_{31}\ z_{31}\ \ x_{32}\ y_{32}\ z_{32}\ \ x_{33}\ y_{33}\ z_{33}\ \ x_{34}\ y_{34}\ z_{34}$
$x_{41}\ y_{41}\ z_{41}\ \ x_{42}\ y_{42}\ z_{42}\ \ x_{43}\ y_{43}\ z_{43}\ \ x_{44}\ y_{44}\ z_{44}$

The first six lines of the file contain information about the surface as a whole. The first line lists the minimum and maximum values of x, y, and z for the surface. The next four lines detail geometry parameters required for integrating the surface with the remainder of the structure generated by `cpfg`. The contact point specifies where the turtle connects to the surface, the end point is where the turtle is positioned after drawing the surface, and the heading and up vectors are matched to the corresponding vectors of the turtle to determine the surface's orientation. Size is a scaling parameter giving the size in surface units to be considered as equivalent to the default unit length associated with the `F` symbol in `cpfg`.

The above section is followed by groups of ten lines, each describing one component patch. As many ten-line groups as there are patches making up the surface are specified. The first line gives the patch name. The next two lines contain patch-specific rendering information giving colors and diffuse lighting coefficients for either side of the surface. If the values are zero, the current turtle parameters are used. The next three lines contain patch neighborhood information. This information is used when rendering to determine if smooth shading is required across a patch boundary. The adjoining patches are specified by their *patchname* in the appropriate position: above left (`AL`), above (`A`), above right (`AR`), left (`L`), right (`R`), below left (`BL`), below (`B`), and below right (`BR`). The lack of a neighboring patch in a given direction is indicated by a ~

symbol. The corresponding entries must match in the neighboring patch specification. The last four lines contain patch control points, each line representing one row of four points each with an x, y, and z coordinate.

6.4.2 Contour specification file

hofs/cpfg3.0.features/-interpretation/-gen_cylinders/-contours

A contour specification file defines the cross-section (contour) of a generalized cylinder. As a default, the contour is a disk. It is possible to use an arbitrary contour defined as an open or closed three dimensional parametric curve consisting of several B-spline segments.

The contour curve is specified by a set of control points. Each control point is defined by two coordinates (in which case the third coordinate is assigned to be 0) or by three coordinates. The file starts with a single header line:

```
num_points dimension type
```

where value num_points specifies the number of control points in the file, value $dimension$ controls the dimension of the contour (2 or 3), and word $type$ is either $open$ for open contours or $closed$ for closed contours.

An example of a contour file follows:

```
12 3 closed
 0.16 -1.12 2.0
 0.41 -1.04 1.0
 0.58 -0.33 0.5
 1.08 -0.04 0.2
 1.08  0.49 0.0
 0.49  0.54 0.0
 0.33  0.91 0.1
-0.37  1.04 0.3
-0.70  0.62 0.2
-1.12  0.16 0.1
-0.87 -0.74 0.3
-0.41 -0.66 1.0
```

It is recommended to specify the control points in the counter-clockwise order (with respect to the point [0,0,0]), because interpolation between clockwise and counter-clockwise contour results in a twisted generalized cylinder.

Note that ff a contour includes some singularity (*e.g.* a sharp edge created by having three control points at the same location), the normals are not correct.

6.4.3 Tsurface specification file

hofs/cpfg3.0.features/-interpretation/-tsurfaces

It is also possible to specify a surface not as a bicubic patch, but by a set of triangles. These triangles are input from a text file which follows a syntax of a rayshade input

file, except that the only lines that are processed are those with the keyword triangle at the beginning of the line.

Following this keyword there are 3 lines, each containing 6 numbers, specifying the x, y, and z coordinates of triangle vertices and the normal in each vertex. Optionally, additional two numbers, defining u, and v coordinates of a texture at the vertex can be included on each line.

Example (without texture coordinates):

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

and with texture coordinates:

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

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

6.4.4 Texture image file

hofs/cpfg3.0.features/-interpretation/-textures/...

Each texture specification (in the view file) includes a file name of an image used for the texturing. Any of the following format can be used:

- RGB — SGI RGB format;
- RAS — SGI colormap (RAS) format;
- TGA — Truevision Targa format;
- RLE — Utah raster toolkit rle format (produced, for example, by rayshade).

The specific format is recognized automatically by the extension (rgb, ras, tga, and rle).

Both the image height and width has to be a power of two. If it is not the case, the texture image is clamped and only a part of the texture appears on the textured surfaces. If your texture size does not meet this condition, scale up or down your texture using command *imscale infile outfile -xres x -y res y*.

6.4.5 Background scene specification file

hofs/cpfg3.0.features/-interpretation/-gen_cylinders/-background_scene

The background scene can be effectively used for defining additional objects around the simulated plant, such as obstacles. It can be also used during the simulation of plant-environment interactions, for visualizing the environmental field together with the plant.

The name of the background scene file is specified in the view file. Often, the scene is read only at the beginning of the simulation, but it is also possible to update it automatically before each interpretation step (animate file command `new view between frames`) or manually (from a menu) for selected steps of the simulation.

Primitives of the background scene are defined in a text file using simple OpenGL-like statements [8]. The commands can be divided into several groups discussed below.

Primitives

The following statements specify basic geometric primitives similarly as in the OpenGL graphics library [8]. The coordinates of the vertices or the size of primitives are defined with respect to a local coordinate system. It is possible to translate the objects or scale them by translating or scaling the coordinate system using transformation statements (see below).

`polygon` $x_1\ y_1\ z_1\ ...\ x_n\ y_n\ z_n$ specifies a polygon with n vertices (x_1, y_1, z_1) to (x_n, y_n, z_n) ($n \geq 3$).

`polygonuv` $x_1\ y_1\ z_1\ nx_1\ ny_1\ nz_1\ ...\ x_n\ y_n\ z_n\ nx_n\ ny_n\ nz_n$ specifies a polygon with n vertices (x_1, y_1, z_1) to (x_n, y_n, z_n) ($n \geq 3$). Each vertex i has also associated a normal (nx_i, ny_i, nz_i).

`rectangle` *a b* defines a rectangle with one vertex in $(0, 0, 0)$ and edges of length a, b along the positive axes x, y, respectively.

`mesh` $x_1\ y_1\ z_1\ ...\ x_n\ y_n\ z_n$ specifies a rectangular mesh: vertices $2k$, $2k+1$, $2k+3$, and $2k+2$ define a single rectangle of the mesh ($n = 4 + 2k$, $k \geq 0$).

`box` *a b c* specifies a box with one vertex in $(0, 0, 0)$ and edges of length a, b, and c along the positive axes x, y, and z, respectively.

`cone` $r_1\ \ r_2\ \ h$ specifies a cone with its axis along y axis, radius at the base equal to r_1, radius at the top equal to r_2, and height h.

`cylinder` $r\ \ h$ specifies a cylinder with its axis along y axis, radius r, and height h.

`sphere` r specifies a sphere with center at $(0, 0, 0)$ and radius r.

Material specification

There is only one statement in this group.

`material` n_1 n_2 ... n_{17} specifies the current material using 17 values that follow the keyword $material$: four values for ambient color (red, green, blue, and alpha, all in the range of 0-1), four for diffuse color, four for specular color, four for emissive color, and one for specular exponent (a value between 0 and 128). The alpha value controls the opacity of the surface (1 for opaque, 0 for transparent). This material is applied to all subsequently defined primitives.

Transformations

All primitives are defined with respect to a local coordinate system. The system can be modified by transformation statements, listed below. The coordinate system is expressed by a single matrix, specifying the transformation necessary to map the world coordinate system into the current local system. Thus every rotation, translation, or scaling modifies only the current transformation matrix. This approach is equivalent to the use of the modelview matrix in OpenGL [8].

`loadidentity` sets the current transformation matrix to identity (*i.e.* the current local coordinate system is equal to the world coordinate system).

`loadmatrix` a_1 a_2 ... a_{16} sets the current matrix. The first four values specify the first column of the matrix, next four the second column, *etc.*

`pushmatrix` stores the current transformation matrix on a matrix stack.

`popmatrix` retrieves a matrix from the stack and sets it as the current transformation matrix.

`translate` *tx ty tz* translates the local coordinate system by vector (tx, ty, tz) (by modifying the current transformation matrix).

`rotate` *angle vx vy vz* rotates the coordinate system around vector (vx, vy, vz) by $angle$ degrees.

`scale` *sx sy sz* scales the local coordinate system by factors sx, sy, and sz in axis x, y, and z.

`multmatrix` a_1 a_2 ... a_{16} multiplies the current transformation matrix by specified matrix.

Example

A sample background scene is specified below.

```
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`, thus macros or comments can be part of it.

The format of the background scene file is also used in transferring the polygons representing selected modules from the plant simulator to the model of the environment. In addition, the same format can be used for the output of the generated structures from the plant simulator, in which case the file also includes statements specifying light sources and the projection (Section 7.4).

7 Output files

The cpfg menu allows the user to save output files in a number of different formats. Output file names can be specified on the command line with defaults derived by replacing the suffix .l of the L-system file with a different suffix: `.rgb`, `.ras`, `.tga`, `.rle`, `.ray`, `.ps`, `.str`, `.strb`, `.gls`, `.vv`, or `.iv` depending on the format chosen. The file name can also be modified interactively through the menu.

The supported output formats are:

- RGB — Saves the current window in SGI RGB format. The file name may be specified on the command line as `-rgb file.rgb`.
- RAS — Saves the current window in SGI colormap (RAS) format. The file name may be specified on the command line as `-ras file.ras`.
- TGA — Saves the current window in Truevision Targa format. The file name may be specified on the command line as `-tga file.tga`.
- RLE — Saves the current window in Utah raster toolkit rle format. The file name may be specified on the command line as `-rle file.rle`.
- Rayshade — Outputs a complete file in rayshade 4.0 format. The viewing parameters produce the same view as cpfg, provided that the perspective view is used. Surface color is chosen according to the current color map, using the basic color modified by the ambient light parameter (but not by the diffuse light component). The file name may be given on the command line as `-ray file.ray`.
- Postscript — Output the generated string in PostScript (see Section 7.2). The file name may be specified on the command line as `-ps file.ps`.
- String — Output the generated string in a text format (see Section 7.3). Two decimal digits of parameter values are output. The file name may be specified on the command line as `-str file.str`.
- String (binary) — Output the generated string in a binary format (see Section 7.3). The file name may be specified on the command line as `-strb file.strb`.
- Graphics Library statements — Output the generated geometry in a local text format (see Section 7.4). The file name may be specified on the command line as `-gls file.gls`.
- View Volume — Output the computed bounding box into a text file. The file format is as follows. The file consists of a single line:

$$box:\ x:\ xmin, xmax\ y:\ ymin, ymax\ z:\ zmin, zmax$$

  The file name may be specified on the command line as `-vv file.vv`.

- Inventor — Output in SGI Inventor format. The file name may be specified on the command line as `-iv file.iv`. This option requires the inventor shared libraries and for some executables may not be available.

The following sections describe some of the supported output formats in more detail.

7.1 Rayshade output

The program `cpfg` allows the user to output the geometry into a rayshade format, making it possible to render the generated objects with a high degree of realism. Rayshade is a public domain ray tracer developed by Craig Kolb. `Cpfg` currently supports rayshade version 4.0, which is available at `ftp://graphics.stanford.edu/pub/rayshade/rayshade4.0` The manual is also available for download from the same site.

The general structure of a file output by `cpfg` is as follows:

```
#ifndef NOSURFACES
/* material definitions */
#endif

#ifndef NOHEADERS
/* view settings */
/* screen resolution */
/* background colors */
/* lights */
#endif

#ifdef BBOX
/* defines only the bounding box */
#else
/* predefined surfaces (a set of triangles for each) */
name l grid 20 20 20
/* surface triangles */
end

/* instantiated objects */

name plant.ray grid 20 20 20
/* objects defining the plant */
end
#endif

/* rescale the object using specified values */

#ifndef NOHEADERS
/* define an instance of the object */
```

```
#endif
```

This structure allows the user to either use the rayshade file on its own or include it in a scene comprising several plants. In the second case, the surfaces and the view may be set by the main rayshade file that includes all plant's files and the local definitions can be ignored (by defining the macro *NOSURFACES* or *NOHEADERS*). Also, only bounding box can be defined for a fast preview of the scene (using the macro *BBOX*).

The following sections describe each feature of the rayshade file.

7.1.1 Materials

The rayshade file includes definitions of all materials specified in the material file used by `cpfg`. In the case a colormap is defined, only colors actually used by `cpfg` objects are output to the file. In rayshade, the material definition uses keyword *surface* (that is why the macro mentioned above is called *NOSURFACES*), followed by the assigned name. The name consists of the letter 's' and the index of the material (or color) corresponding to the index used by `cpfg`, increased by the index of the main material set or colormap multiplied by 256 (usually, the colormap index is equal to 1 and all surface indexes are increased by 256).

If `cpfg` uses materials, all material components except emissive color are included in the surface definition. The transparency parameter (only one per surface, not like in OpenGL where each color can have its own alpha channel) is determined from the alpha value of the emissive color.

If `cpfg` uses a colormap, only ambient and diffuse colors are specified, both equal to the r, g, b color values specified by the colormap.

7.1.2 View parameters and lights

The view and lights set in the rayshade file correspond to the view and lights set by the `cpfg`'s view file. (The object is rotated and scaled so it is oriented the same way as on the screen, using a transformation matrix specified together with the instance of the object at the end of the rayshade file.)

Only perspective projection can be defined in the rayshade file. Consequently, the parallel projection used by `cpfg` has to be converted to perspective projection. This process often produces views which are inconsistent with the view on the scene. For the best results, it is advisable to use perspective projection in `cpfg` if the plant is to be output to rayshade.

7.1.3 Bounding box

To be able to preview a scene that consists of a vast number of plants, it is very convenient to use only a box representing the bounding box of the plant. Note, that the bounding box is incorrect, if the rayshade file is output in the off-screen mode, during which the bounding box is not computed. It is possible, though, to force computing of

the bounding box by outputting also the view volume file (*e.g.* directly to */dev/null* if you do not want to keep it, using command line parameters `-vv /dev/null`).

7.1.4 Predefined surfaces

All predefined surfaces specified in the `cpfg`'s view file are included in the rayshade file. Each surface is named using the single letter name defined in the view file. The surface definition consists of a grid of fixed size ($20 \times 20 \times 20$) containg a set of triangles. The triangles are defined by their vertices, and possible also normals and texture parameters at each vertex. The normals are included if the smooth shading is used by `cpfg`. Since there are two ways of mapping a texture on a surface, it may happen that there are two surface definitions in the file (with index 1 and 2 — as the second letter of the name), each with different texture coordinates associated with vertices.

7.1.5 Instantiated objects

It is possible to take advantage of instantiation not only for surfaces, but also for parts of the plant, such as complex leaves or flowers. If the user specifies a homomorphism production with delimiter `-o>` instead of `-->`, during the rayshade output the predecessor will be instantiated if it appears again (if it has the same parameters and possibly also the same turtle parameters). The precision of object parameters (used for differentiating between two objects created by the same modules with the same number parameters) can be controlled by specifying a format string in the view file (using the command `rayshade objects` — see Section 6.2). It can be specified whether even the turtle is considered when comparing two objects created by the same module with the same parameters (if the objects are different the second one is not an instantiation of the first one).

Sometimes it may happen that an empty object is defined and rayshade would core dump on the file. Currently, a tiny transparent sphere is defined in such cases. This could be better solved by noting which instances are empty and not using them in other places.

ADD: It would be nice to include an example of rayshade instancing (`-o>` productions) where turtle's parameters are considered.

7.1.6 The main object

The name od the main object is generally equal to the name of the rayshade output file (without the path). The whole object is enclosed in a grid of resolution $20 \times 20 \times 20$ to speed up the rendering.

Sometimes, though, `cpfg` may define several plants positioned further away from each other and it is more efficient to use a separate grid for each plant. For this purpose, the user can use module *@J* in the L-system string (see Section 6.1.9). The module

@J($size_1, size_2, size_3$*)* closes the current grid and starts a new grid of a given size (in number of voxels). In this case, there are several objects defined, and the main object is defined at the end of the rayshade file as a list of the parts specified by the *@J* module.

The objects use references to surfaces and instantiated parts, defined earlier in the rayshade file. In addition, it is possible to define a reference to a rayshade object defined in another file, by specifying the name of the object as a parameter of the module *@I* (see also Section 6.1.9). In this case, an instance of the object with the given name is created at the current position, with the orientation given by the turtle, and the scale specified as the second parameter of module *@I*.

Before the instance of the main object is defined at the end of the rayshade file (after the *#ifdef NOHEADERS* statement), the object is possibly scaled using the scale parameter defined in the `cpfg`'s view file (using the command `rayshade scale:`). Note, it is usually better to use the parameter turtle scale (the view file command `initial scale`) which affects all primitives and the final scaling is not necessary.

Note that cylinders and cones are defined as a single primitive, thus they always appear smoothly shaded.

Also, the rayshade format does not support double-sided surfaces, thus if a surface has associated two different materials in `cpfg`, in the rayshade file, only the top material is specified for this surface.

Rayshade reports triangles with edges shorter than 0.00001 as degenerate triangles and cylinders or cones with length below 0.00001 as degenerate cones. The problem is that if these primitives are degenerate they are ignored and it may happen that there will be an object containing no primitives which will cause rayshade to core dump. To avoid this, make sure, for example, that you are not using generalized cylinders which starts or finish with width 0.

7.2 Postscript output

Similarly as for the rayshade output, an attempt was made to produce a PostScript file which is essentially a snapshot of the window. Therefore, the file consists of the L-system object in a box of the background color, positioned the same way as on the screen (even in the case the user interactively rotates and scales the object before the output, regardless the used projection). Care must be taken if standard black and white output is desired for inclusion in text documents (such as in LaTeX). For this purpose, the background is generally made white and the foreground black (or shades of grey).

The following *caveats* apply:

- Textures are not supported.
- Primitives are not drawn with interpolating colours, as `cpfg` draws them. An attempt is made to guess the best colour.
- PostScript has no Z-buffer and no additional depth testing is performed during the output, thus an object located later in the string will overlap another object

located earlier in the string even if on the screen it appears behind the earlier specified object.

Note that if your version of `cpfg` supports Inventor output, it may be preferable to output your models as Inventor objects, and then print them to PostScript using a facility such as `ivprint` or `SceneViewer`.

7.3 L-system string

L-system generated string can be output as a text or binary file. An example of a text file is:

```
A(3,0.25)F(3)[+FA(1,0.5)]
-F(4)@O(0.333333)A(2,0.75)
```

The numbers are output using maximum possible number of digits after the decimal point (*e.g.* 1/3 is output as 0.333333) unless it is possible to output less digits (*e.g.* for 3, 0.25, *etc.*).

The binary file starts with a text header:

```
L-system string: length_in_bytes generation_step_no
```

followed by *length_in_bytes* bytes of the string in internal representation in which module parameters are stored as 4-byte floats.
Example:

```
L-system string: 47 1
A(????,????)F(????)[+FA(????,????)]-F(????)@O(????)A(????,????)
```

7.4 Graphics Library Statements format

The program `cpfg` can output the geometry in a format, similar to the format of a background scene (Section 6.4.5). Thus the geometry produced in one simulation can be included as a background scene in another model. In addition, the GLS format is used by some environmental programs (*e.g. soil* or *arvo*) to define obstacles.

The output file can include all commands specified in Section 6.4.5 plus the following commands:

Lighting:

`clear` $red\ green\ blue$ this command clears the window and sets the background to a given color. Usually included at the beginning of the file.

`light` $posx\ posy\ posz\ posw$ specifies a light source by four homogeneous coordinates of light position. If $posw$ is equal to 0 the light is directional. The color of the light source is always white.

Projection:

`ortho` $minx\ maxx\ miny\ maxy\ front\_dist\ back\_dist$ specifies an orthographic projection the same way as OpenGL library does.

`perspective` $viewing\_angle\ front\_dist\ back\_dist$ specifies a perspective projection the same way as OpenGL library does.

`lookat` $posx\ posy\ posz\ refx\ refy\ refz\ upx\ upy\ upz$ defines the view by specifying the camera position, the view reference point and optionally also the up vector.

Matrices and transformations:

`matrixmode` $0/1$ sets the current matrix (0 for modelview matrix, 1 for projection matrix).

7.5 Inventor output

`Cpfg` can output the generated objects into Inventor format, if the executable was compiled on the system which has Inventor libraries installed on it. The output consists of a main file containing the definition of all objects except predefined surfaces, which are stored in separate files, one file per surface. (Note that sometimes there may be two files per surface, with two different sets of texture coordinates — see also Section 7.1.4.)

The inventor output has the following features:

- The camera is not defined in the file, thus the initial view in the inventor viewer (*e.g.* `ivview` will not correspond to the view in the `cpfg` window).
- Textures are supported, although Inventor always smoothens the texture image. Consequently, it is not possible to have a sharp chessboard texture, for example.
- Regardless the used rendering mode (*shaded*, *flat*, *wireframe*, *etc.*), the resulting objects are always smoothly shaded.
- `Cpfg` generated Inventor files have sometimes too big memory requirements (possibly related to too many items in a group).
- Directional lights do not convert properly to Inventor output (they are defined as a very distant point source).
- If a predefined surface is not included in the view file, the created input file cannot be viewed (the viewer does not read such a file).

ADD: Specify under what conditions which of the statements appear, assumedly by reference to the view file parameters. Although currently, `cpfg` outputs the viewing parameters in a single projection matrix.

8 Communication with environmental process

8.1 Open L-systems[6]

Open L-systems are a generalization of the concept of query modules $?P$, $?H$, $?L$, and $?U$ used in environmentally-sensitive L-systems [6] (see also Section 6.1.9). *Communication modules* of the form $?E(x_1, \ldots, x_m)$ are used both to send and receive environmental information represented by the values of parameters $x_1, \ldots, x_m$ (Figure 4). Specifically, parameters $x_1, \ldots, x_m$ act as an interface between the plant and the environment. They can be set by the plant model and transferred to the environment or set by the environment and transferred to the plant model.

This interface is sufficient for receiving the information from the environment, but the environment also has to obtain information about the position and orientation of plant organs affecting the environment or being affected by it. Thus in addition to parameters of a communication module, the environment receives the position and orientation of the communication module (retrieved from the current turtle parameters), and a module following the communication module (with its parameters).

To accommodate the exchange of information between the plant and its environment each derivation step (after which the interpretation step can be possibly performed[7]) is followed by an *environmental step*. In the environmental step, the string resulting from a derivation step is scanned from left to right to determine the state of the turtle associated with each symbol. This phase is similar to the graphical interpretation of the string, except that the results need not be visualized. Upon encountering a communication symbol, the plant process creates and sends a message to the environment including all or a part of the following information:

- the address (position in the string) of the communication module (mandatory field needed to identify this module when a reply comes from the environment),
- values of parameters x_i,
- the state of the turtle (coordinates of the position and orientation vector, as well as some other attributes, such as the current line width),
- the type and parameters of the module following the communication module in the string (module B in Figure 4). It is also possible to include the graphical representation of this module. Specifically, a set of triangles resulting from the interpretation of the module (or of its homomorphic image — Section 6.1.7) is transferred to the environment.

hofs/environment/-MonteCarlo/test.runs

The environment processes the received information and returns the results to the plant model using messages in the following format:

- the address of the target communication module,

[6] This section is incorporated from [2].

[7] It would be nice to have an option for having interpretation step both before and after the environmental step, only before it or only after it.

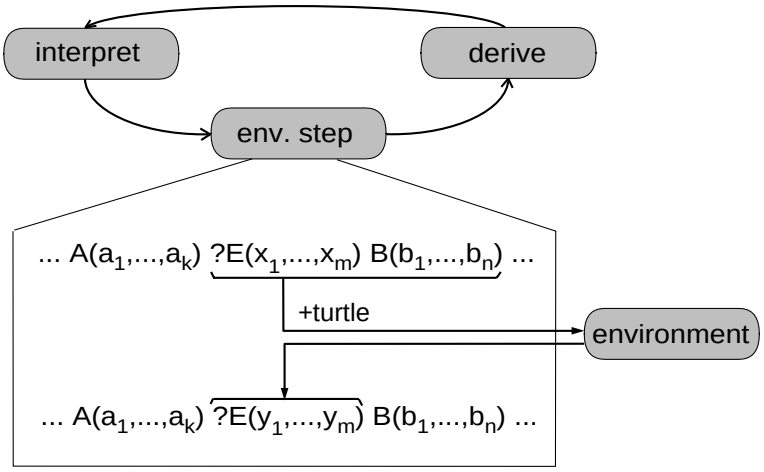


Figure 4: Information flow during the simulation of a plant interacting with the environment, implemented using an open L-system

- values of parameters y_i carrying the output from the environment.

The plant process uses the received information to set parameter values in the communication modules (Figure 4).

Note that by preceding every symbol in the string with a communication module it is possible to pass complete information about the model to the environment. Usually, however, only partial information about the state of a plant is needed as input to the environment, as illustrated in the example below. In addition, the use of addresses makes it possible to send replies from the environment only to selected communication modules. Proper placement of communication modules in the model, combined with careful selection of the information to be exchanged, provide a means for keeping the amount of transferred information at a manageable level.

You can use the communication modules in homomorphism productions, but only to send information to the environment. The environment will not be able to respond, because these modules exists only temporarily during the application of homomorphism to a given module. It is fine to use the communication modules in decomposition productions.

hofs/Thesis/5.OpenLsys/Sierpinski

The following simple example illustrates the operation of an open L-system. The model creates a branching structure consisting of straight line segments. The structure grows by adding a pair of segments to the end of existing branches unless a branch collides with another one. The occurrence of a collision is determined by the environment. To accomplish its task, the environment receives the information about the position of segment's end points and tests whether two points occupy the same place or not. The listing of the environmental process can be found in Section 8.4.5.

The L-system model is as follows.

L-system 3

$\omega: \quad ?E(0)$
$p_1: \quad ?E(c): \ c == 0 \ \rightarrow \ [+F/(180)?E(0)]F?E(0)$

The end point of a segment is represented by a communication module $?E$ with one parameter. This parameter is initialized to 0, and if the point collides with another point, the environment sets it to 1. If the point does not collide, the parameter stays 0. Production p_1 then creates two new branch segments only for points with parameter 0.

The communication is set up in such a way that with each communication module, the environment obtains its identification (the address in the string) and its position.

The first few steps of the simulation are described below.

Initialization. The simulation begins with a single point $?E$. Before the first derivation step, the environmental step is performed and the environment receives the following information:

$$address: 0, \ ?E(0), \ position: 0, 0, 0.$$

It is convenient to think of the address as the number of modules before the communication module. The position is equivalent to the initial position of the turtle. The point obviously does not collide with another point, thus the environment does not reply (*i.e.* sends an empty message) and the parameter of the module $?E$ stays 0.

Step 1. The environmental step is followed by a derivation step, in which production p_1 is applied, replacing module $?E$ with the string:

$$[+F/(180)?E(0)]F?E(0)$$

which is interpreted for visualization purposes (Figure 5a). Now the environment receives two modules:

$$\begin{array}{ll} address: 4, & ?E(0), \ position: 0.5, -0.866, 0, \\ address: 7, & ?E(0), \ position: -0.5, -0.866, 0. \end{array}$$

These two points do not collide and the environment again does not reply.

Step 2. In the next derivation step, production p_1 is applied to both modules $?E$ resulting in the string:

$$[+F/(180)[+F/(180)?E(0)]F?E(0)]F[+F/(180)?E(0)]F?E(0)$$

visualized in Figure 5b. In the following environmental step, the environment receives four modules:

$$\begin{array}{ll} address: 8, & ?E(0), \ position: 0, -1.7321, 0, \\ address: 11, & ?E(0), \ position: 1, -1.7321, 0, \\ address: 18, & ?E(0), \ position: 0, -1.7321, 0, \\ address: 21, & ?E(0), \ position: -1, -1.7321, 0. \end{array}$$

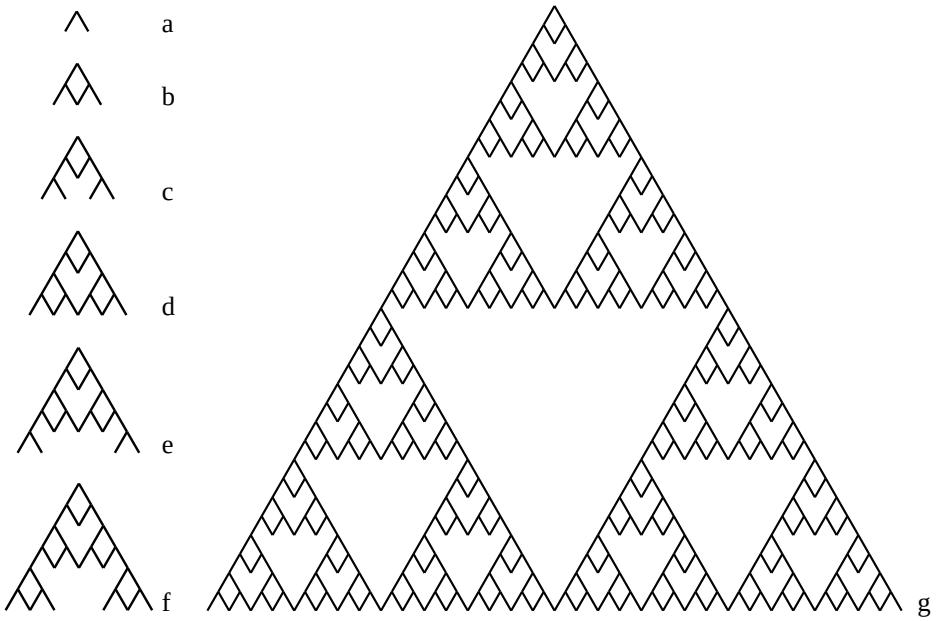


Figure 5: Sierpinski triangle generated by open L-system 3 in 1, 2, ..., 6, and 32 steps

Since the first and third module occupy the same point, the environment returns a message in the form:

$$address : 8, \quad ?E(1),$$
$$address : 18, \quad ?E(1).$$

The plant simulator receives this message and updates the parameters of the specified communication modules resulting in the string:

$$[+F/(180)[+F/(180)?E(1)]F?E(0)]F[+F/(180)?E(1)]F?E(0)$$

Step 3. In the next derivation step, only the second and fourth module $?E$ is replaced by a pair of branches, resulting in a structure shown in Figure 5c.

The simulation then continues generating a branching structure which is similar to the Sierpinski gasket (Figure 5g).

The implementation issues related to the incorporation of open L-systems and the specified communication interface to the plant simulator `cpfg` are discussed in the following section.

8.2 Implementation of the modeling framework

In order to implement the designed modeling framework, the L-system based plant modeling program `cpfg` has been extended by incorporating open L-systems into it

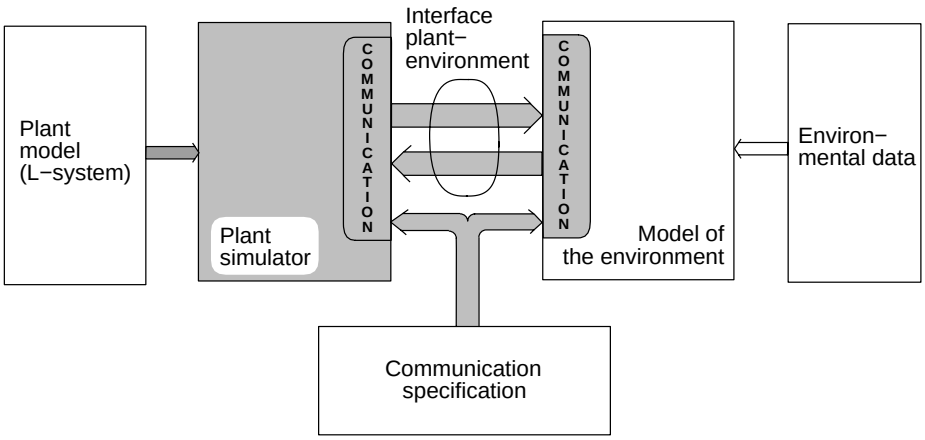


Figure 6: Organization of the software for modeling plants interacting with their environment. Shaded rectangles indicate components of the modeling framework, clear rectangles indicate programs and data that must be created by a user specifying a new model of a plant or environment. Shaded arrows indicate information exchanged in a standardized format.

and by including a special purpose communication library. The library facilitates the exchange of information between the plant model and the environmental process. Consequently, the library also has to be included in a program simulating the environment.

The parameters of the communication are defined in a *communication specification file*, shared between the programs modeling the plant and the environment (Figure 6). The communication specification file is a text file with commands specifying the name of the environmental program (with possible options and input files), the format of data the plant model sends to the environment, and the type of communication between the programs.

For example, the communication in the example from the previous section (L-system 3) has been defined using the following specification file:

```
executable: ulam
turtle position:%.5g %.5g
communication type: pipes
```

The environmental program is called $ulam$ (because it was originally used for generating Ulam's patterns — see [2]), the data between the two processes are transferred using a pair of Unix pipes, and only the turtle position is sent together with each communication module (in addition to the module's address).

The specification of the environmental program is included mainly for the plant simulator, which controls the communication and executes the environment at the beginning of the simulation.

To reduce the amount of transferred data, as a default, only the minimum information is transferred from the plant simulator to the environment, namely the address of the communication module and parameters of the module. All additional information, such as the module following the communication module (and its parameters), the turtle position, orientation, current line width, *etc.* , has to be specified in the communication file (see Section 8.4.1 for the list of all commands). On the other hand, the environment responds by sending selected communication modules with their address and parameters.

The communication between the two programs is implemented using mechanisms provided by the underlying operating system (Unix). Thus the data can be exchanged using a pair of Unix pipes, a pair of sockets, a pair of files, or shared memory. There are always two data streams, one for data going from the plant model to the environment and the other one for data coming back. The variety of communication mechanisms make it possible to choose one that provides an efficient data transfer between the processes (using pipes, sockets, or shared memory) or to choose a slower communication (using files) allowing the user to access the exchanged data (Section 8.4.6).

In the case of pipes or sockets, the synchronization of communication is straightforward: one of the processes waits for the input from the other process on a designated pipe or a socket and the system suspends its operation during that time. In the case of files or shared memory, the communication is synchronized using a pair of semaphores which inform the processes about the availability of data in a shared memory or a designated file.

The communication follows these steps (Figure 7):

1. Plant simulator `cpfg` is executed. It reads the communication specification file, establishes data structures necessary for the communication, starts the environmental process, and waits for the confirmation from the environment.

2. The environmental process reads the communication specification file, connects itself to data streams, confirms its initialization, and waits for the first transmission from the plant simulator.

3. The plant simulator starts the simulation and performs an environmental step to process the communication modules specified in the axiom. The communication modules are transferred to the environment using the specified streams. The last communication module in the string is followed by a reserved end-of-transmission message. The plant simulator then waits for data from the environment.

4. The environment recognizes the beginning of transmission (by being able to read from a pipe or a socket, or by checking a given semaphore) and starts receiving the data. After encountering the end-of-transmission message, the environment processes the queries and starts sending the response back to the plant simulator. The environment terminates the transmission by a similar end-of-transmission

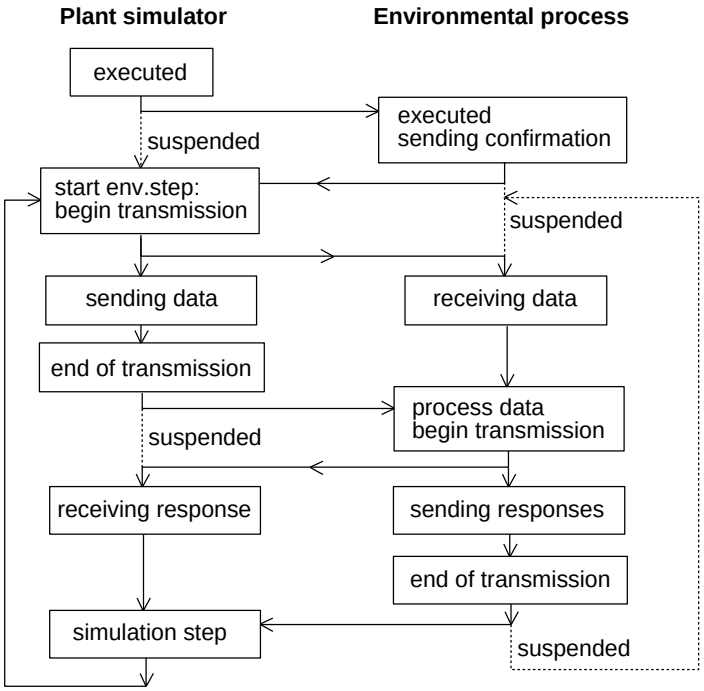


Figure 7: Flow of control during the simulation

message. It then waits again for the plant simulator (returning to the beginning of step 4)[8].

5. The plant simulator receives the data coming from the environment and sets the parameters of communication modules accordingly. After encountering the end-of-transmission message, it performs a simulation step and returns to step 3.

The simulation is terminated by the plant simulator which sends a special terminate message to the environment.

If you are using shared memory or files for the data exchange between `cpfg` and the environmental program and one of the program crashes, the other one will not be terminated and you have to do so manually. (You can list all your processes using `ps -u your_login_name` and kill the process by `kill process_id`. The process id (PID) is listed by `ps`.) Also, the semaphores and the shared memory stays allocated and after a while you may not be able to receive more semaphores from the system. In this case, use `ipcs` to list all your semaphores and shared memory and `ipcrm -m id` or `ipcrm -s id` to remove them.

[8] The organization of communication, with different channels used to send information to and from the environment, makes it also possible to send the response immediately — this situation is not captured in Figure 7.

Because of the great variety of environmental phenomena, there is no "universal" model of the environment. Various phenomena can be modeled by different environmental programs that use a specific representation of the environment suitable for particular problems.

To be able to communicate with the plant simulator, an environmental program has to be compiled with the communication library. The library provides a programmer with a set of functions which have to be called in a given order. Section 8.4.2 provides a list of functions of the communication library and explains how to use them in an environmental program. The section also includes the source code for the environmental program used as an example in Section 8.1.

8.3 Visualization of the environment

Visualization is an essential part of every simulation. It is often useful to visualize not only the plant model but also the environment (creating one composite scene), in order to better understand the interaction between them. The plant simulator `cpfg` provides the user with many useful graphical features [3] making it possible to visualize both the plant and the environment.

The environment can be visualized in two ways:

1. As the background image for the visualized structure. For this purpose, the environmental process outputs an image file which is used by the plant simulator to define a texture on a rectangle representing the background. *hofs/environment/density/Cohen*

2. As a set of primitives, forming a *background scene* which is displayed together with the generated plant. The primitives are read from a text file containing a list of OpenGL-like statements (Section 7.4). *hofs/environment/soil/3d.no.avoiding/tapered*

In the case where the environment is static, it is sufficient to read the texture image or the background scene file once at the beginning of the simulation. In the examples in this chapter, though, the environment is changing over time, thus it is necessary to update the image or the background scene every time the environment changes.

Consequently, the background file is periodically updated by the environmental process and read by the plant simulator after each simulation step, before the visualization. To limit the amount of transferred data, the environmental process can create the background files only at specific simulation steps. The number of the current simulation step is sent by the plant simulator together with the message about the end of transmission (after all communication modules from the string are sent to the environment — see Section 8.4.4).

8.4 Two process communication

8.4.1 Specification of the communication

The communication between the plant simulator `cpfg` and an environmental program is initiated when the simulator is executed with a command line parameter *-e*

comm_spec_file. The communication specification file *comm_spec_file* is a text file with the following commands:

communication type: *pipes/sockets/memory/files* Specifies the type of communication between the plant simulator and the environmental process. The default is *pipes*, because pipes provide the most efficient means of communication on a single machine.

Important note: The standard input stream *stdin* and the standard output stream *stdout* must not be used in an environmental process, because during the communication, the input and output pipes are connected to these streams.

In the case of the file communication, the plant simulator creates two files *.to_fieldXXXX.0* and *.from_fieldXXXX.0*, where *XXXX* is equal to `cpfg`'s process id (as returned by the system).

The communication through memory or files is synchronized by a pair of Unix semaphores, which are set automatically by the plant simulator. The size of the shared memory (in bytes) and communication files (in number of communication modules) as well as the name of communication files are fixed. These values do not limit the amount of transferred data, since the communication is done piecewise. For debugging purposes, it is possible to specify the maximum size of communication files as a parameter following the keyword *files*.

executable: *binary [field_params]* Specifies the executable of the environmental process and its optional command line parameters.

following module: *yes/no* Defines whether the module following the communication module is sent to the environmental process. The default is *no*[9].

turtle position: *format_string*
turtle heading: *format_string*
turtle left: *format_string*
turtle up: *format_string*
turtle line width: *format_string*
turtle scale factor: *format_string*

These commands define C-like format strings for those turtle parameters, which are sent to the environment (currently, only the parameters listed above can be transferred). Often, only the position and the heading vector are necessary, and the rest can be omitted. Since the information is being sent in a text format, it may be desirable to use only a few decimal places or to omit the z axis when possible. For example, commands:

```
turtle position:P:%.3f %.3f
turtle heading:H:%.1f %.5f
```

[9] In the current version of the program `cpfg`, the default value is *yes*.

specify that only the x and y coordinates of the turtle position and heading vector are transferred to the environment (as floating point numbers with the precision of 1, 3, or 5 decimal places). Letters in format strings are helpful for debugging purposes when using files for the communication but they are not mandatory.

interpreted modules: *all* or $M_1(n_1)$, $M_2(n_2)$, ..., $M_n(n_n)$ It is possible to include a set of polygons representing module *X* (following the communication module *?E*) with the data transferred to the environmental process. The module *X* is interpreted when: *hofs/environment/-MonteCarlo/test.runs*

- only the word *all* is specified,
- (n_i) is not present and $M_i = X$, or
- $M_i = X$ and X has n_i parameters (currently, it is not possible to specify $n_i > 6$).

If homomorphism or decomposition productions can be applied to the module *X* (Section 6.1.7), all geometry created by these productions is sent to the environment.

The geometry is transferred as a set of polygons in a text format (see the output format of OpenGL-like commands described in Section 7.4).

verbose: *on/off* Switches on or off the verbose mode, which informs the user about the details of the communication.

8.4.2 Environmental process

Flow of information

An environmental process communicating with the plant simulator operates as a slave, *i.e.* the communication is controlled by the plant simulator (the master). Generally, the environmental process waits for the data from the plant simulator. The data consists of a communication module, its address, and possibly turtle parameters or the module following the communication module. Afterwards, the process sends back the communication modules with modified parameters and waits for new input in a loop.

There are two possible modes of operation of an environmental process, (Figure 8):

1. immediate answer — the parameters of a communication module obtained from the plant simulator can be updated immediately, because the results depend on the local properties of the environment and do not depend on the other communication modules. This mode of operation is suitable, for example, for simulation of static environments that are too complex to be expressed in environmentally-sensitive L-systems.

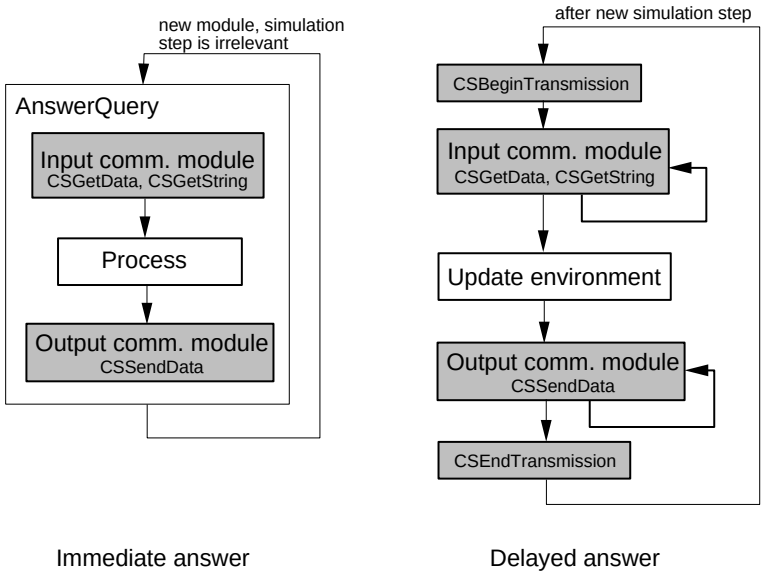


Figure 8: Two possible modes of operation of an environmental process

2. delayed answer — the reply depends on the information obtained from other communication modules in the string, due to the propagation of information through the environment. Thus all communication modules from the string have to be first input (and stored in internal data structures) before the parameters of the communication modules can be properly set. This mode of operation is usually used in the case the plant is affecting the environment, because the response then depends on changes in the environment, introduced by other communication modules.

The functions used to control the flow of information, to receive the data from the plant simulator, and to send data back (shown in Figure 8) are discussed below.

8.4.3 Data structures

Let us first overview the data structures used for the data exchange. The information about selected turtle parameters is received in the structure *CTURTLE*.

```
struct CTURTLE {
  float position[3];
  int positionC;     /* number of values sent for position */
  float heading[3];
  int headingC;      /* number of values sent for heading */
  float left[3];
```

```
  int leftC;         /* number of values sent for left */
  float up[3];
  int upC;           /* number of values sent for up */
  float line_width;
  int line_widthC;   /* number of values sent for width */
  float scale_factor;
  int scale_factorC; /* number of values sent for scale */
};
typedef struct CTURTLE CTURTLE;
```

It contains selected turtle parameters together with a parameter specifying how many values have been sent for a given parameter. Thus if a particular turtle parameter is not received by the environment (*i.e.* it is not listed in the communication specification file), the corresponding "count" parameter is set to 0. This allows the environmental process to check whether a required turtle parameter is available (see the examples below).

Structure *Cmodule_type* is used to store parameters of the communication module and the module immediately following it:

```
#define CMAXPARAMS   20 /* max. number of module parameters */
#define CMAXSYMBOLLEN 4 /* max. length of a module name */
struct module_type {
  char symbol[CMAXSYMBOLLEN+1];
  int num_params;
  struct param_type {
    float value;
    char  set;          /* if set=1, the value is sent back */
  } params[CMAXPARAMS];
};
typedef struct module_type Cmodule_type;
```

The structure consists of the module name (possibly a multisymbol module, such as $@Gs$, $@Gc$, $@Tx$, *etc.* — see Section 6.1.9), the number parameters, and an array of parameter values. Since the same structure is also used to inform the plant model about modified parameters of the communication module, the flag *set* associated with each parameter value specifies whether the parameter has been modified by the environment or not.

Both structures *Cmodule_type* and *CTURTLE* are defined in the library header file *comm_lib.h*.

8.4.4 Library functions

To facilitate the writing of an environmental process, the following functions are specified in the communication library (*comm*). The first two functions are used in both modes of operation (Section 8.4.2):

void **CSInitialize***(int *argc, char ***argv)*

Initializes the communication and parses necessary options. This call should be made as the very first operation in the function *main*(). The parameters of the function $CSInitialize()$ are pointers to the standard parameters of the function *main*(), specifying the number of command line options of the program and an array storing these options. Since the communication library may add some additional, internally used options to the command line, the function $CSInitialize()$ parses these options and updates the values of parameters $argc$ and $argv$ so that the user can process the options listed after the command $executable$ in the communication specification file (see Section 8.4.1).

void **CTerminate***(void)*

Ends the communication — this should be the last call in the function *main*().

If the parameters of a communication module can be modified immediately, the following function can be used.

void **CSMainLoop***(int (*Answer) (Cmodule_type *, CTURTLE *))*

The parameter of the function $CSMainLoop()$ is a mapping function $Answer()$. The mapping function modifies the parameters of the communication module, stored in a two-dimensional array (pointed to by the first function parameter), which also includes the module following the communication module. The second parameter of the function contains the received turtle parameters.

If the environmental program calls the function *CSMainLoop*() with a mapping function as the parameter, the communication is fully controlled by the communication part of the modeling system. The function *CSMainLoop*() returns when the plant simulator sends a message to terminate the environmental program. The environmental program can then clear its local data structures and call *CTerminate*() (see the first example in Section 8.4.5).

If the incoming query cannot be answered immediately, the following functions have to be called in a specific order (see Figure 8 and the code listing below):

int **CSBeginTransmission***(void)*

Starts transmission (of all communication modules in the string generated by the plant simulator). The process waits for the plant simulator to perform a simulation step and to send the first communication module. The function always returns a value of 1.

int **CSGetData***(int *master, unsigned long *module_id,*
*Cmodule_type *two_modules, CTURTLE *turtle)*

Obtains a communication module and possibly the following module from the plant simulator (if the second module is not present, its name is an empty string, *i.e.* $two\_modules[1].symbol[0]$ is equal to 0). The parameter *module_id* specifies a unique identification number of the communication module, the pointer

two_modules points to a two-dimensional array containing the communication module and the next module, and the pointer *turtle* points to the turtle structure (note that only some turtle parameters are sent, according to the specification file). The parameter *master* is set to the index of the calling master. This value is used only in a multiprocess environment (Section 8.5) and in the case of a two-process communication, it is always equal to 0.

The function returns 0 when there is no other module (at the end of the environmental pass). In this case, *module_id* is set to the number of the current simulation step.

int **CSGetString***(int *master, char *str, int length)*

Reads a string *str*, with maximum length *length*, sent by the plant simulator. According to the communication specification file, selected modules can be interpreted during an environmental step and the polygons representing the modules (or their homomorphic image) are sent as a set of strings following the communication module. Thus the function *CSGetString*() is used in a loop after each call to *CSGetData*() to retrieve these strings. It is recommended to always include a loop of calls to *CSGetString*() to receive possible strings from the incoming data (see examples below), because if the plant simulator sends some strings, which are not read by the environmental process, the communication would be interrupted.

The function returns 0 when there is no string coming.

void **CSSendData***(int master, unsigned long module_id,*
*Cmodule_type *comm_module)*

Sends the modified communication module back to the plant simulator. The original *module_id* must be specified. In the case of two-process communication, the value of *master* should be 0.

int **CSEndTransmission***(void)*

Ends a transmission (after all modified communication modules are sent back to the plant simulator). The function returns 1 when the process is requested to terminate. In this case, the communication loop should be exited, the process should free its data structures, and call *CTerminate()*.

Instead of calling the library function *CSMainLoop*(), the user has to define a function *MainLoop*(), which should have the following general form.

```
void MainLoop(void)
{
  Cmodule_type two_modules[2];
  int master, current_step;
  unsigned long module_id;
  CTURTLE turtle;
  char str[2048];
```

```
  for(;;) {
    CSBeginTransmission();
    while(CSGetData(&master, &module_id, two_modules, &turtle)) {
      StoreQuery(master, module_id, two_modules, &turtle)
      /* store all or some of the queries - do not forget
         to store values of 'master' and 'module_id'! */

      while(CSGetString(&master, str, sizeof(str))) {
        ProcessGraphics(master, str);
        /* process the graphical representation of the
           module following the communication module */
    }
    DetermineResponse(); /* determine the answers*/

    SendBackResponse();
    /* send back modified communication modules using
      CSSendData(master, module_id, &two_modules[0]); */

    if(CSEndTransmission()) break;
  }
}
```

Functions *StoreQuery*(), *ProcessGraphics*(), *DetermineResponse*(), and *SendBackResponse*() have to be defined by the user depending on the data structures chosen for storing and processing the incoming communication modules. In the case of a two-process communication, the parameter *master* may be ignored (and for function *CSSendData*() set to 0). To be able to use the program in a multiprocess environment (see Section 8.5), the parameter *master* should be stored as well.

The second example in the following section illustrates the use of the functions listed above.

8.4.5 Examples

Two simple examples of an environmental process are presented below. The first example illustrates the case when the parameters of the received communication module can be set immediately, thus the program uses the function $CSMainLoop()$.

```
#include <stdio.h>
#include "comm_lib.h"

int Answer(Cmodule_type *two_modules, CTURTLE *turtle)
{
 static float zero[3]={0,0,0};
```

```
 if(turtle->positionC < 3) {
   fprintf(stderr,"Turtle position not set!\n");
   return 0;
 }
 if(two_modules[0].num_params >= 1) {
   two_modules[0].params[0].set = 1; /* parameter modified */
   two_modules[0].params[0].value = Distance(turtle.position,zero)
     > two_modules[0].params[0].value ? 1 : 0;
 }
 return 1;
}

void main(int argc, char **argv)
{
 CSInitialize(&argc, &argv);
 CSMainLoop(Answer);
 CTerminate();
}
```

The function $Answer$ determines the distance of the turtle position from the point $(0, 0, 0)$ and if it is greater than the first parameter of the communication module, the parameter is set to 1. Otherwise it is set to 0.

The following example illustrates the second mode of operation, when the incoming communication modules (queries) have to be stored before their parameters can be modified. The environmental program detects whether a communication module collides with another one. The program has been used in the model of Sierpinski's gasket from Section 8.1.

The communication is defined by the following communication specification file.

```
executable: point_collision
communication type: pipes
turtle position:%.5g %.5g
```

The environmental program is given below.

```
/****  Environmental process - testing point overlapping  ****/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "comm_lib.h"

#define EPSILON    0.001 /* precision of comparisons */
#define MAXQUERIES 1000 /* maximum number of queries */
struct item_type {
```

```
  float position[2];
  float query;
  unsigned long id;
  int master;
} queries[MAXQUERIES];  /* queries */

int num_queries;        /* actual number of stored queries */

/*************************************************************/
void StoreQuery(int master, unsigned long module_id,
                Cmodule_type *comm_symbol, CTURTLE *tu)
{
  if(tu->positionC < 2) {
    /* do not write to stdout, because it is used for pipes */
    fprintf(stderr,"environment: turtle position missing.\n");
    return;
  }
  if(num_queries >= MAXQUERIES) {
    fprintf(stderr, "environment: too many queries!\n");
    return;
  }
  queries[num_queries].position[0] = tu->position[0];
  queries[num_queries].position[1] = tu->position[1];
  queries[num_queries].query = comm_symbol->num_params >= 1;
            /* answer only if ?E has one or more parameters */
  queries[num_queries].master = master;
  queries[num_queries].id = module_id;
  num_queries++;
}

/*************************************************************/
void DetermineResponse(void)
{
 int i, j;
 Cmodule_type comm_symbol;

 comm_symbol.num_params = 1;
 comm_symbol.params[0].set = 1;
 comm_symbol.params[0].value = 0; /* report only collisions */

 for(i=0; i< num_queries; i++)    /* for all queries */
  if(queries[i].query) {   /* don't answer if no parameter */
   for(j=0; j< num_queries; j++)
    if(i!=j)
```

```
     if(fabs(queries[i].position[0]-queries[j].position[0])
                 < EPSILON &&
        fabs(queries[i].position[1]-queries[j].position[1])
                 < EPSILON) {
        CSSendData(queries[i].master,queries[i].id,
                   &comm_symbol);
        break;
     }
  }
}

/*************************************************************/
void MainLoop(void)
{ /* controls the loop of data exchange */
  Cmodule_type two_modules[2];
  unsigned long module_id;
  int master;
  CTURTLE turtle;

  /* infinite loop - until message 'exit' comes */
  for(;;) {
    CSBeginTransmission();
    num_queries = 0;

    while(CSGetData(&master,&module_id,two_modules,&turtle))
      StoreQuery(master, module_id, two_modules, &turtle);

    DetermineResponse();

    /* EndTransmission returns 1 when the process is
       requested to exit */
    if(CSEndTransmission()) break;
  }
}
/*************************************************************/
int main(int argc, char **argv)
{
  /* initialize the communication as the very first thing */
  CSInitialize(&argc, &argv);
  MainLoop();
  CTerminate(); /* should be the last function called */
  return 1;
}
```

Each incoming query is stored in a one-dimensional array of a fixed size. To determine the response for queries with more than one parameter, the coordinates of the query point are compared with the coordinates of all other points. If there is another point with the same coordinates, the environmental process sends the value 0 to the plant model. Otherwise, the parameter of the communication module stays unchanged and there is no reply by the environment.

8.4.6 Troubleshooting

During the design of a model, it may be necessary to find out whether proper data are transferred between the environment and the plant simulator. To view the exchanged data, it is possible to use the file communication and to display the content of files *.to_fieldXXXX.0* and *.to_fieldXXXX.0*.

If the amount of transferred data is too large though, the data are transferred from one process to the other in several chunks (each stored in a file with the same name). Thus the user can access only the last chunk of data. The maximum size (in the number of modules) of the data file is predefined, but it is possible to increase it to a value large enough so that there is only one communication file used during the data exchange (by adding a number behind the keyword *file* in the specification file).

Often it is also necessary to debug the environmental program. The debugging is much easier if the program is running in a stand-alone mode, without the plant simulator. To achieve this, it is possible to use the pipe communication and to run only the environmental program, while inputing the data to the standard input and receiving the response on the standard output. The input data can also be redirected from a file. This file can be either created in a text editor or obtained from files exchanged between the processes during a regular simulation (which uses the file communication).

Running the environmental process in the stand-alone mode then follows these steps:

1. The simulation is first run with the plant simulator, using the file communication. Each time the simulation is stopped, it is possible to concatenate the data file *.to_fieldXXXX.0* (the data sent to the environment in the last data exchange) to a file *to_field*, which is of zero length at the beginning of the simulation. It is also possible to choose the data file from only one simulation step.

2. The user may change the content of the file (*e.g.* edit some values) or add a message terminating the environmental process, by including a line containing the string "*Control: 8*" to the end of the file *to_field*.

3. The environmental process can then be run separately, by setting the communication type to *pipes* and redirecting the file *to_field* to the standard input:

   environment -e comm_spec_file $<$ *to_field*

   The program writes all modified communication modules to the standard output in a text format.

8.5 Distributed system

In a distributed system, several plant models can communicate with different environmental processes and then send the graphical interpretation of the models into a single drawing window.

Each program in the system has to be compiled with the provided communication library. Since the programs can be running on different machines, they exchange data using Unix sockets. The connection to other processes is specified in the command line of a process, following the switch *–C*:

hofs/cpfg3.4.examples/Distr.environ

```
-C -c,confirm_socket,start_machine;-m:spec_file1,socket1...
...;-s:spec_fileK,socketK,master_machineK
```

The single string of switches specifies:

-c the number of a socket and a machine name to which the confirmation about a successful execution should be sent. After the confirmation, the process monitors the socket for a possible request to terminate.

-m a master connection. The process operates as a master: it sends data specified in the communication specification file *spec_file1* to the defined socket (*socket1*) and expects the reply on a socket with number *socket1+1*. There can be several master connections.

-s a slave connection. The process acts as a slave: it expects data (defined in the communication specification file *spec_fileK*) from a given socket on a specified machine, processes the incoming data and responds back through a socket with number *socketK+1*. If there are more slave connections the data from the sockets are processed in the order given in the command line.

The delimiters in the command string can be characters ',' or ';'.

The communication specification file contains the same commands as in the case of two-process communication with following modifications:

- command *executable* is ignored.
- the type of communication (*communication type*) is also ignored; it is always set to *sockets*.
- two new commands have been added:

  strings only: on/off the data exchanged between the processes contain only text strings. No L-system modules are transferred. This switch is used, for example, for transferring a list of primitives from plant models to the drawing program.

  binary data: on/off binary data can be exchanged between processes. The receiving process must be aware of the coming binary data, thus often the data are preceded by a special command string.

The following section lists the functions provided by the communication library.

8.5.1 Communication library functions

This sections lists all functions of the communication library, which can be used by processes communicating with each other in a distributed system.

First, there are three functions used by all processes regardless their role in the communication (*i.e.* slave, master, or both):

void **CInitialize***(char *program_name, char *command_string)*
: Initializes the communication. The first parameter specifies the name of the process, which is used to distinguish messages from different processes displayed on the same terminal. The second parameter contains the string following the switch *-C* containing the specification of all connections. The format of the string is described above.

void **CSInitialize***(int *argc, char ***argv)*
: An optional function to the previous one. This function retrieves all the necessary information from the process' command line. In this case, the switch *-C* has to be the first one on the command line.

void **CTerminate***(void)*
: Ends the communication — this should be the last function called.

int **CShouldTerminate***(void)*
: This function returns 1 if the process is requested to terminate — when a special character is sent to the process' confirmation socket by a control process (see Section 8.5.2).

A process can operate as a slave, master or both. The latter means that the process waits for an input from its masters and then can require some data from its slaves. Communicating with its masters, the process can use functions as in the case of a two-process communication:

void **CSMainLoop***(int (*AnswerQuery) (Cmodule_type *, CTURTLE *))*

int **CSBeginTransmission***(void)*

int **CSEndTransmission***(void)*

int **CSGetString***(int *master, char *str, int length)*

int **CSGetData***(int *master, unsigned long *module_id,*
*Cmodule_type *two_modules, CTURTLE *turtle)*
: This function is generally used only by environmental processes directly communicating with the plant simulator (`cpfg`).

void **CSSendData***(int master, unsigned long module_id,*
*Cmodule_type *comm_module)*
: This function is generally used only by environmental processes directly communicating with the lant simulator (`cpfg`).

In addition there are few new functions:

int **CSGetNumberOfMasters***(void)*
Returns the number of connections to a master specified on the command line.

int **CSSendString***(int master, char *item)*
Sends a string to the specified master (with index *master*).

int **CSSendBinaryData***(int master, char *item, int item_size, int nitems)*
Sends a binary data to the specified master. This function is used, for example, for sending images with a depth information to a drawing program (see Section 8.5.3). The function returns 0 if the data have not been sent.

To communicate with its slaves, a process can call functions:

int **CMBeginTransmission***(void)*
Initializes connections to all slave processes for a single data exchange. Currently, the function always returns 1.

int **CMEndTransmission***(int current_step)*
Terminates the data sent by the master in a single data exchange. The parameter *current_step* is used by `cpfg` to send the number of the current simulation step to the environment. This number is returned as the *module_id* parameter of the function *CSGetData* (see above). Currently, the function always returns 1.

int **CMTerminate***(void)*
Terminates all slave processes. The function returns 1, if all processes are successfully terminated.

int **CMGetNumberOfSlaves***(void)*
Returns the number of slaves communicating with the process.

int **CMSendString***(int slave, char *item)*
Sends a string to the specified slave.

int **CMGetString***(int slave, char *str, int length)*
Receives a string from the specified slave. Returns 0 if there is no string coming.

int **CMSendBinaryData***(int slave, char *item, int item_size, int nitems)*
Transfers a binary data to a given slave.

int **CMGetBinaryData***(int slave, char *data, int item_size, int nitems)*
Receives a binary data from a given slave. Returns 0 if there is no data coming.

int **CMSendCommSymbol***(int slave, unsigned long module_id,*
*Cmodule_type *two_modules, CTURTLE *turtle)*

Sends two modules (a communication module with the following module) with

their identification number to a given slave. The function returns 1 if the second module should be graphically interpreted and the resulting set of triangles transferred to the slave. This function is used mainly by the plant simulator `cpfg`.

int **CMGetCommunicationModule***(int slave, unsigned long *module_id, Cmodule_type *comm_module)*

Receives a communication module with its identification number from a specified slave. The function returns 0, if there are no more modules coming from the slave.

Unless specified, the functions return a value of 1, if they finish successfully. Note that unlike for a slave, in the case of a master the incoming data are fetched from a specified slave. This allows the program to process the response from the same slave as the one to which the data from the master has been transferred.

Example. Following example illustrates a program *hub* that acts as a common interface between several plant simulators (masters of the *hub*) and models of the environment (slaves of the *hub*). The program establishes all connections to its masters and slaves. Then for each communication module, it receives from a given master, it sends the module to a slave with an index specified as the first parameter of the communication module. The index of the master is stored as the first parameter of the symbol sent to the slave.

After each module is transferred to a specific slave, the response from a slave is checked and if there is one, the communication modules sent by the slave are transferred to the proper master. Just in case the slave processes are not responding immediately, the input from all of them is again checked at the end of a single transmission.

The full listing of the program follows.

```
#include "comm_lib.h"

void MainLoop(void)
{
 Cmodule_type two_modules[2], comm_module;
 unsigned long module_id;
 CTURTLE turtle;
 char string[2048];
 int slave, master;

 /* infinite loop - until signal 'exit' comes */
 for(;;) {
  CSBeginTransmission();

  /* begin transmission to all slaves */
  for(i=0;i<CMGetNumberOfSlaves();i++)
    CMBeginTransmission(i);
```

```
/* process the data */
while(CSGetData(&master, &module_id, two_modules, &turtle)) {
 if(two_modules[0].num_params>0 &&
    two_modules[0].params[0].value > 0 &&
    two_modules[0].params[0].value <= CMGetNumberOfSlaves()) {

    slave = two_modules[0].params[0].value-1;

    /* store the index of the master */
    two_modules[0].params[0].value = master;

    if(CMSendCommSymbol(slave,
                        module_id, two_modules, &turtle)) {
      /* send graphics */
      while(CSGetString(&master, string, sizeof(string)))
            CMSendString(slave, string);
    }

    /* check for possible response */
    while(CMGetCommunicationModule(slave, &module_id,
                                   &comm_module)) {
      /* retrieve the master */
      master = comm_module.params[0].value;
      /* do not change the first parameter */
      comm_module.params[0].set = 0;

      CSSendData(master, module_id, comm_module);
    }
 }

CMEndTransmission(module_id);

/* process the rest */
for(slave = 0; slave < CMGetNumberOfSlaves(); slave++)
  while(CMGetCommunicationModule(slave,&module_id,&comm_module)){
    /* retrieve the master */
    master = comm_module.params[0].value;
    /* do not change the first parameter */
    comm_module.params[0].set = 0;

    CSSendData(master, module_id, comm_module);
  }
```

```
  if(CSEndTransmission()) break;
  }
}

/*************************************************************/
void main(int argc, char **argv)
{
  /* establishes all connections according to -C parameter */
  CSInitialize(&argc, &argv);

  MainLoop();

  CTerminate();
}
```

The two sections below describes the function of an initialization program *start* and a simple drawing program *draw*.

8.5.2 Initialization program

hofs/cpfg3.4.examples/Distr.environ/-with_start

Program *start* reads in a text specification file and executes a set of communicating processes. The program takes as a parameter the name of the specification file and an optional switch *-v* to run the program in a verbose mode with detailed report displayed in the terminal window.

The specification file contains the definition of processes and connections between couples of processes. The specification of a process starts with command **processes:** followed by a group of lines, not separated by an empty line, with commands:

name: *process_name* Defines a unique process name.

host: *machine_name* Defines the name of the machine on which the process is executed. If the command is omitted, the local machine is used.

files: *list of filenames* Specifies files which have to be copied to the remote machine. This command can be repeated several times to specify more files.

command: *binary with parameters* Defines the executable of the process. The full path does not have to be specified if the process path is set in the system variable $PATH$.

display: *machine* Specifies the machine to which the display is redirected. If not specified, the system variable $DISPLAY$ is set to the local machine. Since plant simulator `cpfg` needs the connection to an X-server, it is necessary to specify a valid display variable, especially if the program is executed on a remote machine. The X-server connection is required in order to use calls to graphics

library OpenGL. If the plant simulator is connected to the X-server on a different machine, a lot of data have to be transferred between these machines to create the image of the plant. Thus if it is not possible to connect to the display on the same machine as the one of the plant simulator, either the plant simulator should generate only a list of primitives or L-system strings or it should be compiled using a public domain graphics library equivalent to OpenGL[10], which does not need the connection to an X-server.

Note that command *setenv* used for modifying the variable *DISPLAY* may not work in every operating system. In that case it is possible to add a corresponding variant of the command *setenv* to the command line of the process.

Each command can be replaced by one or more letters, because only the first character of a command is considered. A line starting with symbol '#' is ignored. An empty line separates definitions of different processes.

Every process specified in the list has to be compiled with the same version of the communication library and has to start with call to function *CIntialize()* or *CSInitialize()* and finish with a call to *CTerminate()* (see the previous section).

All communicating processes have to be specified before defining connections. The specification of connections follows the command *connections:*. Each line defines a set of connections. For example, the line:

```
master1,...,masterN -> slave1,...,slaveM: spec_file common_parameters
```

initiates $M.N$ connections between each of the N masters and all M slaves using the same type of communication (specified in the communication specification file). In addition, common command-line parameters added to process executables (*common_parameters*) can be defined. Processes are referred to by their name.

In the following example, a drawing process *draw_server* (named *draw*) communicates with two modeling programs `cpfg` (named *tree1* and *tree2*). Both modeling programs are masters to an environmental process *chiba* (named *light*) simulating the local light environment shared by the two trees. The specification file is listed below.

```
processes:
name: draw
files: specs.e
command: draw_server -r shaded

name: tree1
host: ik
files: specs.e tree.e tree.mat tree1.l tree.v tree.a leaf.s
command: cpfg -a -M tree.mat tree1.l tree.v tree.a
display: shere
```

[10] Library MESA by Brian Paul available at http://www.ssec.wisc.edu/~brianp/Mesa.html.

```
name: tree2
host: shere
files: specs.e tree.e tree.mat tree2.l tree.v tree.a leaf.s
command: cpfg -a -M tree.mat tree2.l tree.v tree.a

name: light
host: ip
files: tree.e light.spec
command: chiba2 light.spec

connections:
draw -> tree1,tree2: specs.e -g -w 640 480
tree1,tree2 -> light: tree.e
```

The program *start* operates as follows. First, it processes the specification file. For each process run on a remote machine, a unique directory */tmp/cpfg_tmp_XXXXXX* is created on the specified host and all required files are copied into it (using system command *rcp*).

After all connection are read, all processes are started one by one using system command *rsh*:

rsh host "cd /tmp/cpfg_tmp_XXXXXX; setenv DISPLAY host:0.0;
path/binary connection_params common_params specified_params &" &

The process path *path* is either specified or obtained using system command *which*. The connection is specified by connection parameters *connection_params* described at the beginning of this section:

```
-C -c,confirm_socket,host_of_start;-m:spec_file1,socket1,...
...;-s:spec_fileK,socketK,master_hostK
```

Each process obtains a socket *confirm_socket* to which it should confirm execution and send a character 'T' upon termination. Each master connection is defined by a socket and a specification file. Each slave connection in addition needs the machine name of the master that sets up the socket. Parameters *common_params* are parameters shared by both processes communicating with each other (specified with each connection, in the example above *-g -w 640 480*) and *specified_params* are these defined with the process.

The program *start* waits up to 20 seconds for the confirmation of a successful execution of a process. If it is confirmed, the next process is spawned. Otherwise all previously started processes are terminated by sending character 'K' on their confirm socket. This sockets is automatically monitored during calls to functions of the communication library. Any ongoing communication is terminated and the process is forced to terminate.

After all processes are successfully started the program waits for a signal from processes about their termination. After all signals are received, the program removes all files and directories created in */tmp* on all used hosts.

8.5.3 Drawing program

hofs/cpfg3.4.examples/Distr.environ/-with_start

In the example from the previous section, two processes simulating two trees are communicating with a drawing program *draw_server* that displays both trees in a single window.

The program *draw_server* communicates with plant simulators by sending text commands, such as *new view* or *step*, and receiving the graphical information about the simulated structure. Currently, the program recognizes two forms of graphical data. First, it is an array of values representing the color and depth for each pixel. The depth value is necessary in case images from several programs are combined into one window. The second format consists of a list of primitives (a sequence of OpenGL-like commands — see Section 7.4) describing the geometry of the modeled structure.

The program *draw_server* is linked with the communication library *comm*, thus connections between the program and plant models are defined using command line switch *-C*. This switch is set automatically if the distributing program *start* is used (see the previous section). Other command line parameters include:

–r mode Sets the rendering mode. Currently, *shaded*, *flat*, and *wireframe* mode is supported. The default is *shaded*.

–c num Defines the number of polygons around a cylinder.

–w xsize ysize Specifies the size of the window. Since the program is drawing the models into an pixmap, which is then copied into the window when necessary and it does not store the information coming from plant models, it is not possible to resize the window.

Switches *-r* and *-c* are considered only if the interpretation of OpenGL-like commands is performed.

Although the program *draw_server* is designed to communicate with the plant simulator `cpfg`, it is possible to use a simple process *draw_client* that sends all the data from its standard input to the drawing process specified by a socket.

Let us consider an example, in which first the drawing programs is executed with following switches:

```
draw_server -C -m:specs.e,1244 -w 640 480
```

setting the window size and a socket for the communication. The communication is specified by file *specs.e*:

```
strings only: on
```

which allows strings being sent between the processes (not communication modules $?E$).

After its execution, the drawing process is waiting for the first transfer of data through the specified socket. If the *draw_server* is running on machine *shere*, for example, the data can be sent by calling:

```
draw_client -C -m:specs.e,1244,shere
```

and typing in following commands:

```
clear 1 1 1   /* background color */
material
 0.5 0.5 0 1  /* ambient color (r,g,b,alpha) */
 1 1 0 1      /* diffuse color */
 0 0 0 1      /* specular color */
 0 0 0 1      /* emissive color */
 0            /* transparency */

polygon
 0 0 0
 0 1 0
 1 0 0
```

When the last line is typed in and the input is terminated by pressing keys *Control* and *D*, the data are transferred to the drawing process that draws a yellow triangle on a white background.

Thus it is possible to add the drawing client into the distributed system and interactively add primitives to the visualized scene.

9 Miscellaneous features

9.1 Rayshade instantiation

hofs/cpfg3.4.examples/homomorphism/rayshade.instancing

Homomorphism productions generally produce the same geometry for a given module with a given set of parameters. It is then convenient to take advantage of this information during the output into a rayshade file format, since this format supports instantiation.

It is possible to mark selected homomorphism productions (using a delimiter `-o>` instead of the standard `-->`). During the rayshade output an object with a name given by the predecessor of the productions and the values of its parameters is created. Each time such a module is encountered during the interpretation of the string (while creating the rayshade file), only a reference to the given object is included in the file, not the geometry representing the module.

The output consists of three stages:

Stage 1. The string is parsed left to right with the full update of the turtle parameters, but no output file is created yet. Each time a homomorphism production with the delimiter `-o>` is applied to a module, this module with its parameters is searched in a hash table. If the hash table does not contain the same module with the same set of parameters, the module is added to the table (together with selected turtle parameters — the line width, the scale factor, the color index, the color index for the back side of a surface, and the texture index). Otherwise the encountered module is an instance of the already stored module and it is not necessary to add it to the hash table.

It is possible to control the precision with which the parameters of the encountered module are compared with parameters of stored modules. To this end, a view file command `rayshade objects:` $format$ defines the printf style format string (*e.g.* $\%.3f$), used for specifying the precision of the compared parameters. The default value is $\%g$ (the full precision is used), but the number of instances of a single module can be increased, and consequently the size of the rayshade file reduced, if the precision is decreased to a few decimal points. In this case a module can be represented by a module with slightly different parameters, but the resulting structure may be still acceptably close to the original.

In some instances, the same modules with the same parameters can result in different structures, because some turtle parameters, such as the current line width or color index are different when the second instance of the module is interpreted. It is possible to include also turtle parameters (namely the line width, the scale factor, the color index, the color index for the back side of a surface, and the texture index) to the comparison between two modules with the same parameters. To do so, words *turtle considered* have to be added after the format string to the view file command `rayshade object`.

At the end of this stage, the hash table contains all the modules whose geometry has to be specified at the beginning of the rayshade file (since rayshade format does not allow backward referencing). Each module is also linked to the module that previously

occurred during the interpretation, to be able to process the modules in the opposite order than the order in which they appeared (used in the following stage).

Stage 2. The modules stored in the hash table are interpreted, in the order given by the linked list (from right to left in the string). For each module M or $M(a_1, a_2, ..., a_n)$, a rayshade object with the name M or $M\_a_1\_a_2\_...\_a_n\_$ is created and all the geometry resulting from the interpretation of the module is stored within this object. If the module is not a letter, the name starts with symbol c followed by the ascii code of the character (for example, module ';' would be represented as $c073$). In the case that also the turtle parameters are used for the differentiating between the same instances, a symbol i followed by a unique index of the module is added to the object name. The index differentiate between the same modules with the same parameters.

At the beginning of interpretation of each module, the turtle position and orientation is set to the default values (positioned at 0, and pointing upwards). Other turtle parameters, such as the current color or material index or the current line width are set to the values stored with the module in the hash table. Thus if the turtle is ignored during the stage 1, all instances will use the same turtle parameters as the ones at the first occurrence of the module (during the stage 1). If the turtle is considered for the module comparisons, each instance will have correct turtle parameters.

If during the interpretation of a module another module that can be found in the hash table is encountered, the encountered module is not interpreted. Instead, a reference to its object name is included in the rayshade file, followed by a transformation matrix capturing the current turtle position and orientation. Processing of the modules in the opposite order than the order in which they appear during the interpretation guarantees that the object is already defined in the file.

Stage 3. After all instantiated modules are output to the file, the L-system string is interpreted again. If a module is not found in the hash table, its geometry is output to the rayshade file. In the case, that the turtle is considered for the comparisons, even if the turtle parameters of the interpreted and stored module differ the module's geometry is output to the file. If the module is found in the hash table, only the reference to the predefined object is included in the rayshade file. To properly position and orient the object, the reference to the object is followed by a transformation matrix capturing the current turtle position and orientation.

9.2 Sending commands to cpfg through sockets

hoofs/cpfg3.4.examples/socket.commands

It is possible to control the interactive operation of the plant simulator *cpfg* by sending the commands through sockets. Each menu item has a corresponding command. This functionality allows the user, for example, to modify the L-system file, view file, or any other input file by an external program and then send a command corresponding to the cpfg menu items *New L-system* or *New View*. Thus the displayed model can be updated without interactive participation of the user.

Note that this functionality is available only in the interactive mode of operation.

To be able to send the commands to *cpfg*, it is necessary to execute it with a command line switch `-S` followed by an arbitrary number specifying the socket:

$$cpfg\text{ -}M\ plant.mat\text{ -}S\ 3000\ plant.l\ plant.v.$$

Afterwards, the user can send an arbitrary command representing a menu item to the plant simulator by using a program *command_client*. The program has two parameters, the first one specifies the network name of the machine *cpfg* is running on (thus the command can be send also from a remote machine) and the number of the corresponding socket. The first parameter can be omitted in which case the program *command_client* is trying to access a socket on the local machine.

The commands sent to *cpfg* contain the text of the desired menu items (in case of submenus, also the text of the upper menu is included, separated by '—'). The program *command_client* reads the commands from the standard input (one command per line), but it is often more convenient to pipe the commands to the program. For example:

$$echo\ "New\ view"\ |\ command\_client\ machine\ 3000.$$

or

$$echo\ "Output|Image|RGB|Save\ as\ ..."\ |\ command\_client\ 3000.$$

The command can be all in lower case, because the matching is not case sensitive.

If the menu item with a predefined filename is to be selected, the file has to be replaced by a dot ('.'). Thus

$$echo\ "Input|String|binary|Input from."\ |\ command\_client\ 3000$$

inputs a binary string to *cpfg* from the default file name, unless another file name is specified on the command line:

$$cpfg\text{ -}M\ plant.mat\text{ -}strb\ my\_string.strb\text{ -}S\ 3000\ plant.l\ plant.v.$$

10 Limitations

This section addresses some limitations of the current version of the plant simulator *cpfg*.

10.1 Using the hardware colormap

When using a hardware colormap, the program *cpfg* checks whether a colormap of size 4096 already exists. Usually it does, as indicated by an X-root variable *_SGI_DEFAULT_COLORMAP* and *cpfg* uses this colormap (the id if the colormap is again accessible from the X-root). Sometimes, though, the colormap does not exists or is not big enough and an external program *install_map* is called.

This program creates and installs a colormap of the size 4096 (if possible), creates a new X-root variable called *_OPENGL_INDEX_COLORMAP*, and stores the colormap

index there. Unfortunately, to properly install the colormap the program has to be terminated. Thus this cannot be done by *cpfg*.

The program *install_map* is located together with other utilities in the same directory as *cpfg*. This directory should be included in your *PATH* variable, otherwise the program will not be executed and the colormap allocation fails.

10.2 Using cpfg on less than 24-bit screens

If your hardware does not support true color visuals (*i.e.* it has less than 24-bits per pixel), the index mode may not work properly. To determine the number of bits per pixel of the screen buffer, run command *ginv* (on SGIs only) and sum the number of bitplanes for a single buffered alpha, red, green, and blue channels.

For example, on 8-bit screen you will be able to use only a colormap of size 256 (not 16×256 as is usual on on 24-bit screen) and in the index mode, `cpfg` will automatically switch to this 256 entries even if the default colormap uses the second 256 entries in the hardware colormap of a bigger size. Unfortunately, the current versions of utility programs `loadmap` and `savemap` do not recognize the type of the screen and try to load or save the second 256 entries. Thus it maybe necessary to use command line parameter -c0 both with `loadmap` and `savemap`.

You can also use command line switch `-m` with a colormap file or switch `-M` with a material file but in this case, the low number of bits per pixel will significantly reduce the quality of the output (the image will be dithered). At least use a single buffer mode (command line switch `-sb`) to increase the number of bit planes allocated for each pixel (in double-buffered mode, the number is divided by two — *e.g.* 4 bits per front and 4 bits per back buffer as compared with 8 bits in a ingle-buffer mode).

10.3 Use of symbol # in the L-system file

Make sure that the symbol # does not appear as the first symbol on the line in an L-system file or the first symbol after tabs or spaces. Otherwise, the preprocessor tries to recognize it as its command and the reading of the file fails. If you would like to use a production with # as the predecessor add an empty left context, such as in:

$$* < \#(wid) \rightarrow \#(wid * 0.9).$$

Also make sure that in the case of multiple-line successors the new line does not starts with # and move the module to the previous line.

10.4 Transparent objects

The support of transparent objects is not very strong in OpenGL. To render transparent objects correctly, it is necessary to perform two passes through the objects, first draw the opaque objects, then to sort all transparent or semitransparent objects according to their position with respect to the viewer, and draw them in that order (with the depth

buffer switched off). This is a very time consuming process. Consequently, all objects resulting from the interpretation of the L-system string are drawn opaque (even if the material has transparency set to a value above 0).

Nevertheless, it is possible to define transparent object in the background scene (used, for example, to visualize the concentration contour in the three-dimensional model of roots). The only limitation is that the objects are not sorted for the second drawing pass and the resulting image may be incorrect.

hofs/cpfg3.0.features/-interpretation/-gen_cylinders/-background_scene

Note that the transparent objects are output to rayshade or inventor even if they are not transparent on the screen.

11 Things to do

11.1 Problems

- On April 21 1998, Jim discovered a problem when using new homomorphisms and the instance stuff; it causes a crash. He wanted to look at it. I am not sure what is the current status.
- When using stochastic productions, `cpfg` requires stochastic values for ALL productions, whether they are stochastic or deterministic. This should not be necessary. Also, the seed is not set! This should be looked at soon.
- you cannot have a variable with the same name as an array. It used to be possible, but now it does not work. An error message is printed if such a variable or array is defined, but it should be fixed so that the programs allows both.
- There's an error message that is given in the new subLsystem code when a recursive call is attempted to a sub-Lsystem:

  ```
  ERROR: Recursive call to Sub-L-system #
  ```

  Right now it exits the program, but it should do like other errors do, and leave the process running for future rereads.
- One thing Jim have noticed is that when running the binary on an O2 there are strange things happening with the buffers. There seems to be a one pixel margin around the edge of the window that gets cleared ok, but when the image is finally drawn apparently random colours appear in that margin, giving a pulsing effect. Very disconcerting. He does not have the O2 any more, but he should be getting his Octane early in 1998 and he will check it again, and try compiling to see if it fixes the problem.
- The memory allocated by `cpfg`'s (the resident size) increases with each New View. This increase may be significant in animations in which the new view is invoked after each animation step. This problem has not been traced yet, it may be something related to OpenGL or X. It is not anything directly caused by `cpfg` calling malloc, strdump, or realloc, because these calls can be monitored if `cpfg` is compiled after running `make heapcheck`. Maybe related to textures? Maybe `cpfg` does not call some cleaning functions of X or OpenGL.
- The Jim's changes related to variables local to each sub L-system do not take account of cut strings appended at the end of the L-system string.
- Sometimes, the buffers are not switched properly, if you resize the window.
- The size of the rgb image output by `cpfg` is wrong, but ras output is fine (on IRIX 6.2).

- The normals seem to be wrong when `@Gr` is set to -90.
- The bounding box is wrong for all off-screen generated rayshade outputs (used when defined the optional object consisting of only the bounding box — see Section7.1), unless the user includes also the view volume (using `-vv /dev/null`). Since determining the view volume takes some time and the bounding object is often not used, it would not be a good idea to compute the bounding box as a default, but maybe some switch would be nice.
- when mapping textures on generalized cylinders, the aspect ratio of the texture image is preserved. To do this, the length of the contour has to be computed. Right now it is done approximately by computing the distances between $10 \cdot n$ points lying on the contour (n is the number of control points specifying the contour). It would be better to compute the real length of the contour.
- During instancing of homomorphism productions in rayshade output (Section 7.1), it may happen that an empty object is defined and rayshade would core dump on the file. Currently, a tiny transparent sphere is defined in such cases. This could be better solved by noting which instances are empty and not using them in other places.
- If a contour includes some singularity (*e.g.* a sharp edge created by having three control points at the same location), the normals are not correct.
- `Cpfg` should start with a reasonable colormap in the index mode to avoid a black window if the user forgets to run `loadmap`.
- Spheres are not textured. What mapping to use?
- `Cpfg` generated Inventor files have sometimes too big memory requirements (possibly related to too many items in a group).
- Directional lights do not convert properly to Inventor output (they are defined as a very distant point source).
- If a predefined surface is not included in the view file, the created input file cannot be viewed (the viewer does not read such a file). Either remove the references to such surfaces or at least print a warning message.
- Currently, `cpfg` sends only recognized blackbox modules to the environment (*e.g.* $@C$ is not passed to the environment but $@O$ is). This is a bigger problem related to multiple-symbol modules. I would suggest to send only a single module after $?E$, because a homomorphism production can be used to replace this module with a multiple-symbol module.
- `Cpfg` displays the final image without showing the drawing process — a departure from `cpfg2.7`; visible, in particular in lilacs.

- Maybe input string should not reread the view. A good question — what should and what should not be reread?
- The materials with textures sure gives a lot more scope for making images... but its slow on old machines. It'd be nice to have textures off during rotations or something...
- It would be nice to enhance rotation speed by allowing for a different mode while the object is being rotated.
- Sub L-systems should have names, instead of the cryptic numbers.

11.2 Fixes to the manual

- Section 7.4 need to say under what conditions which of the statements appear, assumedly by reference to the view file parameters. Although currently, `cpfg` outputs the viewing parameters in a single projection matrix.
- Format strings are used in a few places. It would be nice to have a section explaining the general setup, along with usage examples.
- It would be nice to include an example of rayshade instancing (`-o>` productions) where turtle's parameters are considered.
- Regarding the manual describing environmental programs, it may be nice to include an example of the environment argument file in the description (this is optional, because an example is a part of the vlab object that you point to an example anyway).

11.3 Suggestions for future extensions or improvements

- It would be nice to have an option for having interpretation step both before and after the environmental step, only before it or only after it.
- Global homomorphism is not implemented. Right now, each L-system has its own homomorphism productions (page 24).
- Extend the programming language by incorporating structures, user-defined functions, or typed variables.
- It would be really good if the extent of labels was included in the bounding box computed by `cpfg`.
- It would be helpful to be able to define a command that would be run for files before input and after output (*e.g.* gzip).

- In case of textures, do not limit the size to power of two (*e.g.* some new machines can handle a size of a multiple of 2). Include a switch or create a bigger texture image with black boundaries and scale the texel coordinates.
- Adding of depth test to postscript output.
- Textures in the background scene.
- It would be better if it was possible to avoid menus by pressing keys. Especially when the menu causes expose event after it is closed. At least to have a stop animation button.
- Create an HTML version of this manual.
- Enable user-defined functions in `cpfg`.
- Is there a way how to allow the user to specify a blackbox functions?
- Add antialiasing.
- For a more efficient visualization of environments add the possibility to send the visualization information (images or GLS files) through additional data stream or using the current communication process (at the end after all `?E`s are sent back to `cpfg`).
- Use the OpenGL shared display lists for a more efficient displaying of predefined surfaces.
- Switch between sub L-systems (or tables) on a flag. (this could be done even now by using a global variable and having a production which would switch the sub L-system if this variable is changed).
- Switch off environmental step on a flag. (as in the previous point, modules $?E$ can be introduced just before they are needed, using global variables, *e.g.* a step counter — although this solution would not eliminate the environmental pass, only no data would be transferred between `cpfg` and the environment).
- enable different homomorphisms, one for the environment, one for the screen (actually there could be one for each type of output as well — string, rayshade, postscript, *etc.*). Switching from one to another could be controlled by some variable
- Add a smooth interpolation of colors, *e.g.* along the stems (even in the shaded mode).
- Would it be possible to use shared libraries to add mathematical and blackbox functions without recompiling?

- How about the ability to put the labels in screen space?? maybe in a separate $@L$ parameter 0-1 for each dimension and scaled to fit?
- Might be nice if fonts could be specified per label.
- Allow the user to change format for parameters in string output; currently scientific notation.
- Should there be a window opened when a warning message is sent rather than just to the console? The console is not necessarily open.
- Allow for different timing within sub L-systems.
- In perspective viewing, can the image be automatically scaled to properly fit the window (and if this is the case, how should the parameters be passed to rayshade)?
- How about having the light stay fixed when the object is rotated?
- How about having the system call incorporate additional variable values using an sprintf?
- Make it possible to access view parameters from productions.

Part II
Examples

This section contains examples of many models created by `cpfg`. The input files for these examples are included with `cpfg` and can be conveniently examined and experimented with using the Virtual Laboratory framework, `vlab`. Instructions for getting the Virtual Laboratory distribution are given in Section 3.

12 Quadratic Koch island

Figure 9 shows several approximations of the *quadratic Koch island* from *The Algorithmic Beauty of Plants* [7] page 8. They were generated with the command:

```
cpfg koch.l koch.v
```

The files' contents are detailed in the following sections.

12.1 koch.l

```
lsystem: 0
derivation length: 3

axiom: F-F-F-F

F --> F+F-F-FF+F+F-F

endlsystem
```

This L-system introduces three turtle symbols: `F`,`+`, and `-`. The `F` symbol causes the turtle to move forward, and draw a straight line. The `+` and `-` symbols cause the turtle to turn counter-clockwise and clockwise respectively. The amount that the turtle turns (90° in this example) is specified in the viewing file (Section 12.2).

The axiom `F-F-F-F` draws a square. The production:

```
F --> F+F-F-FF+F+F-F
```

replaces each line segment with a shape as shown in Figure 10. Note that there are no productions for the `+` and `-` symbols. Symbols with no replacement productions are replaced with themselves. In other words, `cpfg` treats this L-system as if it contained these productions:

```
+ --> +
- --> -
```

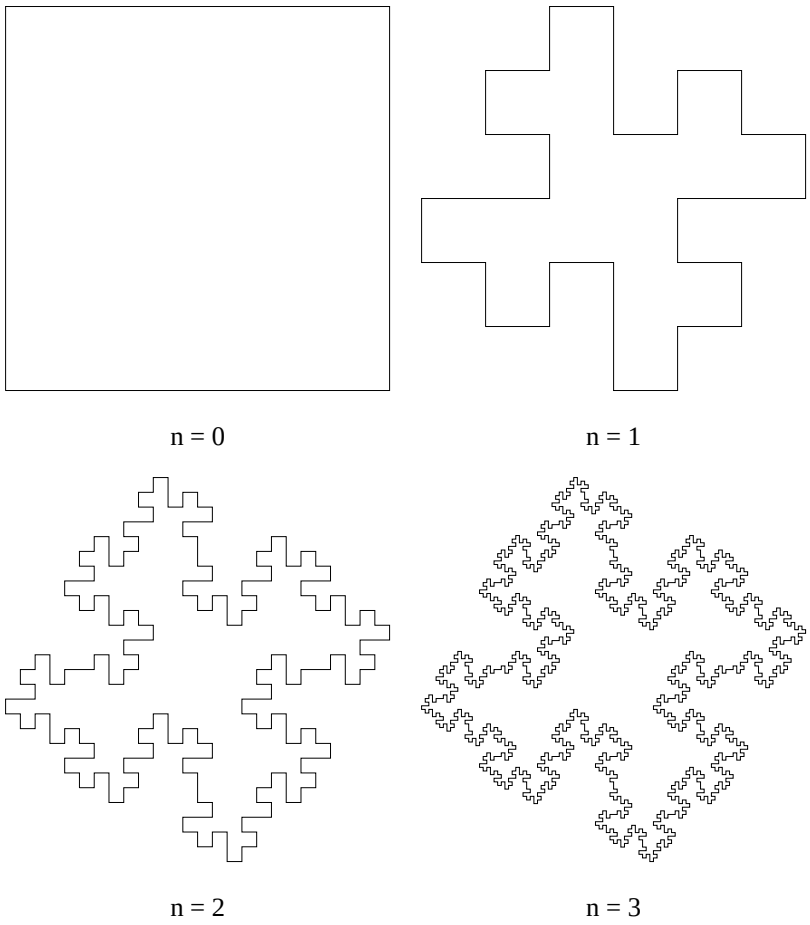


Figure 9: Koch Islands generated in $n = 0$,1,2,and 3 derivation steps

12.2 koch.v

```
angle factor: 4
initial color: 1
color increment: 0
initial line width: 2
line width increment: 0
viewpoint: 0,0,1
view reference point: 0,0,0
twist: 0
```

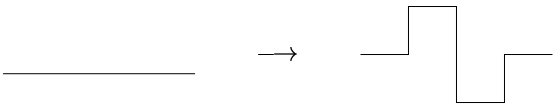


Figure 10: The production F $\longrightarrow$ F+F-F-FF+F+F-F

```
projection:
parallel front distance: -100000.0
back distance: 100000.0
scale factor: 0.9
z buffer: off
cue range: 0
shade mode: 7
light direction: 1.0, 1.0, 1.0
diffuse reflection: 0
tropism direction: 0.0,1.0,0.0
initial elasticity: 0.0
elasticity increment: 0.0
```

This is a fairly typical viewing file. The most important value for this fractal is:

```
angle increment: 90
```

This tells `cpfg` that the angle increment used with the `+` and `-` commands is equal to $90°$.

13 Koch snowflake curve

Figure 11 shows several derivations of the *Koch snowflake*. They were generated with the command:

```
cpfg snowflake.l snowflake.v
```

The files' contents are detailed in the following sections.

13.1 snowflake.l

```
lsystem: 0
derivation length:  3
```

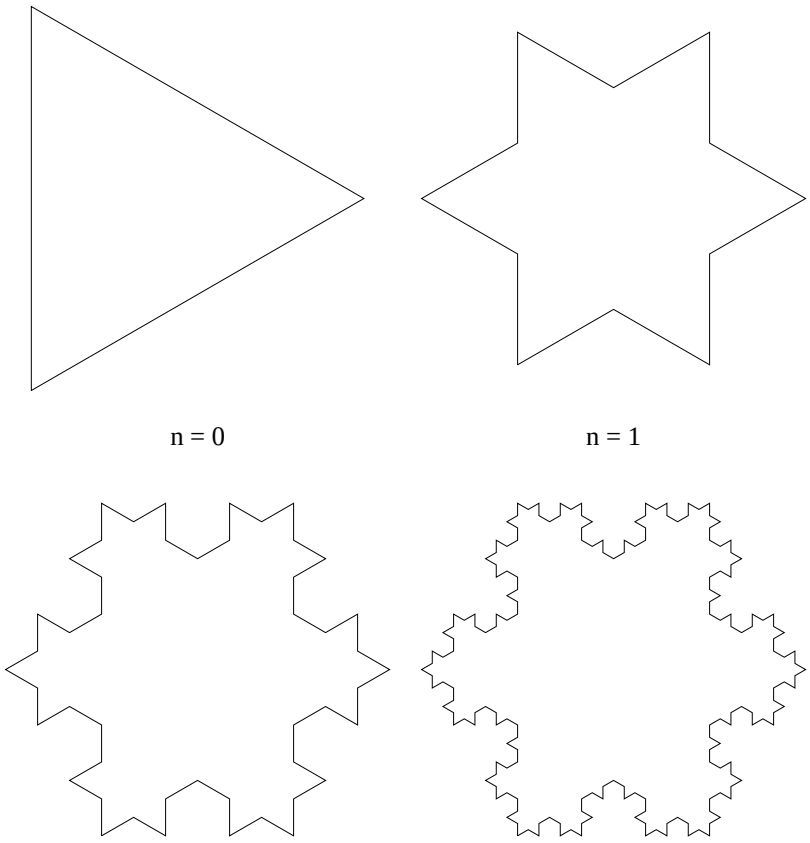


Figure 11: Snowflake curves

```
axiom: F-F-F
F --> F+F--F+F
endlsystem
```

The axiom `F-F-F` draws a triangle. The production:

```
F --> F+F--F+F
```

replaces each line segment with the shape shown in Figure 12.

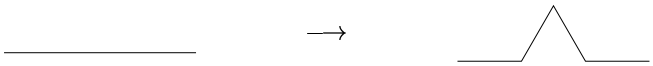


Figure 12: The production F $\longrightarrow$ F+F–F+F

13.2 snowflake.v

```
angle increment: 60
      .
      .
      .
```

The + and - commands for this L-system rotate the turtle by 60°. The viewing file for the snowflake is identical to that for the Quadratic Koch island except for the different angle increment.

14 Combination of islands and lakes

Figure 13 illustrates an application of the turtle symbol: `f`, which moves the turtle forward, but does not draw a line. The L-system used to generate this image is shown below.

14.1 lakes.l

```
lsystem: 0
derivation length:  2

axiom: F+F+F+F

F --> F+f-FF+F+FF+Ff+FF-f+FF-F-FF-Ff-FFF
f --> ffffff

endlsystem
```

The axiom `F+F+F+F` draws a square. The production:

```
F --> F+f-FF+F+FF+Ff+FF-f+FF-F-FF-Ff-FFF
```

replaces each line segment with the shape shown in Figure 14.

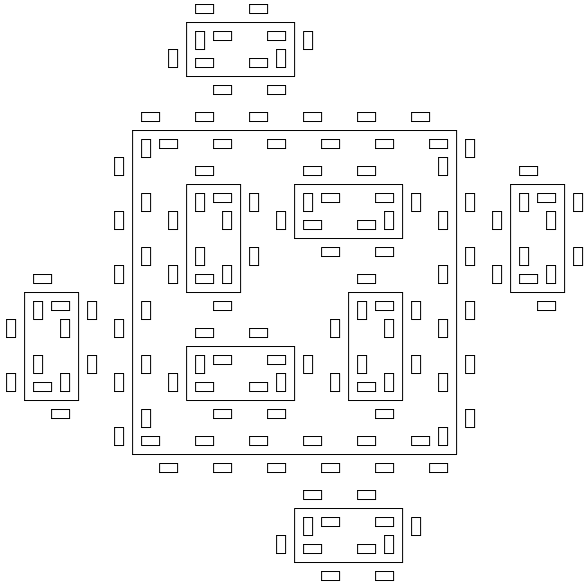


Figure 13: Islands and Lakes

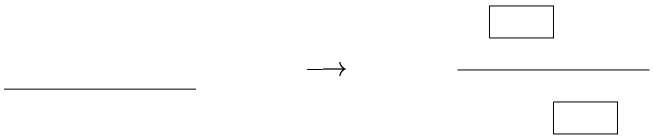


Figure 14: The production F $\longrightarrow$ F+f-FF+F+FF+Ff+FF-f+FF-F-FF-Ff-FFF

15 Dragon curve

Figure 15 shows several generations of the *dragon curve*. The L-system used to generate this image is shown below.

15.1 dragon.l

```
lsystem: 0
derivation length:  12
axiom: FL
```

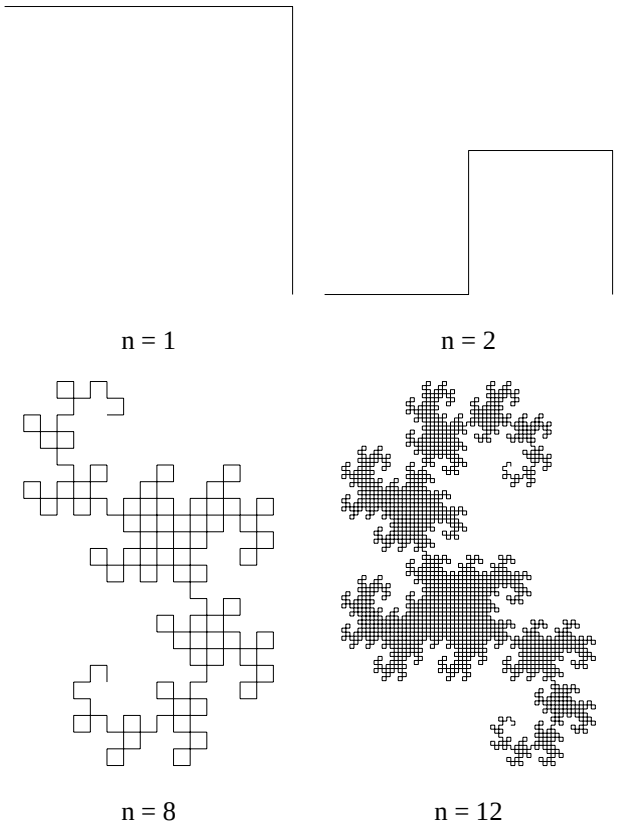


Figure 15: Dragon Curves

```
L --> L+RF+
R--> -FL-R
endlsystem
```

The dragon curve consists of two types of edges, “left” and “right”. The L-system which generates the dragon curve is based on an L-system with two symbols for edges F_l and F_r:

$$\begin{array}{lcl} \text{axiom:} & & F_l \\ p_1 & : & F_l \rightarrow F_l + F_r + \\ p_2 & : & F_r \rightarrow -F_l - F_r \end{array}$$

Figure 16 shows the replacements made by this L-system. We can convert this L-system to one which uses only one type of edge symbol as follows.

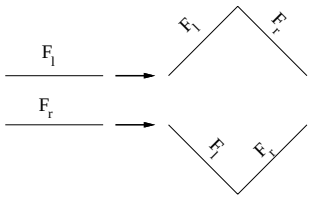


Figure 16: The productions $F_l \rightarrow F_l + F_r +$ and $F_r \rightarrow -F_l - F_r$

Assume temporarily that a production predecessor can contain more than one letter; thus an entire subword can be replaced by the successor of a single production (a formalization of this concept is termed a *pseudo-L-system* and is discussed in *The Algorithmic Beauty of Plants* [7]). The dragon-generating L-system can be rewritten as:

$$\begin{array}{lcl} \text{axiom}: & & Fl \\ p_1 & : & Fl \rightarrow Fl + rF + \\ p_2 & : & rF \rightarrow -Fl - rF \end{array}$$

where the symbols l and r are not interpreted by the turtle. Production p_1 replaces the letter l by the string $l + rF -$ while the leading letter F is left intact. In a similar way, production p_2 replaces the letter r by the string $-Fl - r$ and leaves the trailing F intact. Thus, the L-system can be transformed as follows:

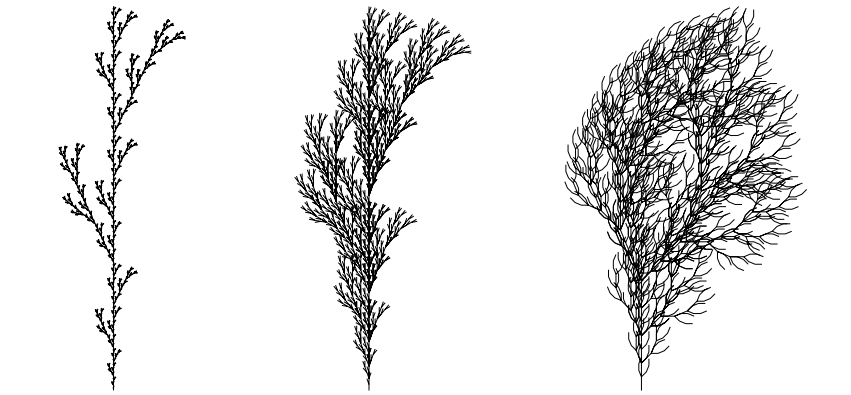
$$\begin{array}{lcl} \text{axiom}: & & Fl \\ p_1 & : & l \rightarrow l + rF + \\ p_2 & : & r \rightarrow -Fl - r \end{array}$$

16 Branching structures

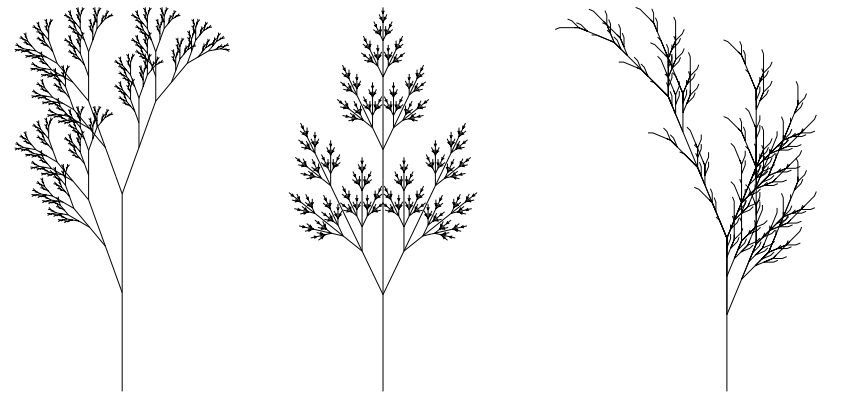
Branches in structures such as those shown in Figure 17 are delimited by the turtle symbols `[` and `]`. The turtle saves its state at the start of a branch, and restores it when the turtle reaches the end. See page 24 of *The Algorithmic Beauty of Plants* [7] for more details. The following section shows the L-system file for the first "plant". Productions for the remaining structures are indicated in Figure 17.

16.1 plant.l

```
lsystem: 0
derivation length: 5
axiom: F
F--> F[+F]F[-F]F
endlsystem
```



F $\longrightarrow$ F[+F]F[-F]F F $\longrightarrow$ F[+F]F[-F][F] F $\longrightarrow$ FF-[-F+F+F]+[+F-F-F]



X $\longrightarrow$ F[+X]F[-X]+X
F $\longrightarrow$ FF

X $\longrightarrow$ F[+X][-X]FX
F $\longrightarrow$ FF

X $\longrightarrow$ F-[[X]+X]+F[+FX]-X
F $\longrightarrow$ FF

Figure 17: Examples of plant-like branching structures

17 Stochastic L-systems

All plants generated by the same deterministic L-system are identical. An attempt to combine them in the same picture would produce a striking, artificial regularity. Stochastic L-systems provide for random variations that preserve the general aspects of a type of plant, but modify the details.

Figure 18 shows several plants generated with the same stochastic L-system (except

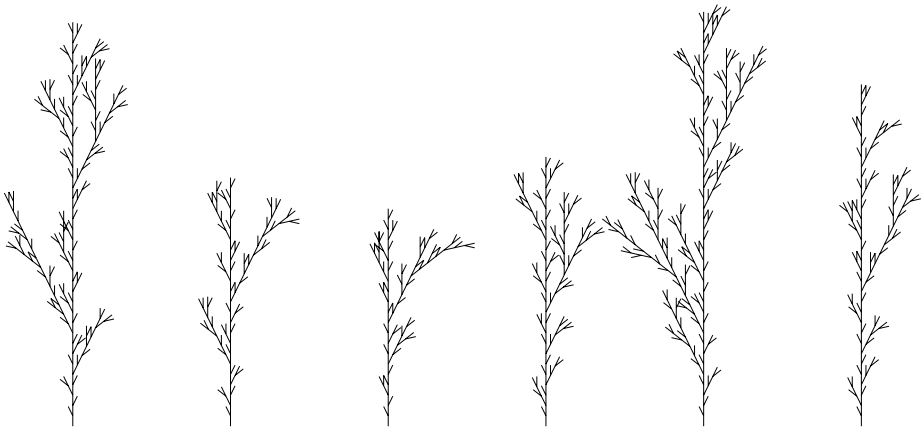


Figure 18: Stochastic branching structures

for different seed values)[11].

17.1 plants.l

```
lsystem:
seed: 2454
derivation length:  3
axiom: F
F--> F[+F]F[-F]F : 1/3
F--> F[+F]F      : 1/3
F--> F[-F]F      : 1/3
endlsystem
```

This is a stochastic L-system. There are three possible successors for the `F` symbol. For each `F` symbol in the string, `cpfg` randomly picks one of the three available productions. The probabilities for each production are given as $1/3$, so they are equally likely to be applied. Note that expressions can be used for the probability. The `seed` keyword specifies a seed for the random number generator.

18 Context sensitive L-systems

Figure 19 shows a plant generated with a context-sensitive L-system.

[11] Note that the **New Model** and **New L-system** menu options do not reset the seed value, so a different structure will be generated each time one of these items is selected.

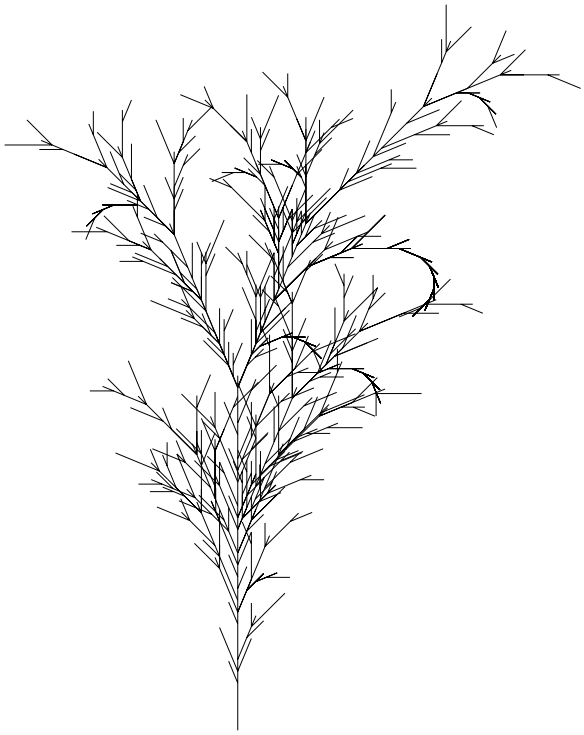


Figure 19: A plant generated with a context-sensitive L-system

18.1 context.l

```
lsystem: 0
derivation length: 30
ignore: +-F

axiom: F1F1F1

1 < 1 > 1 --> 0
1 < 1 > 0 --> 1
1 < 0 > 1 --> 1F1
1 < 0 > 0 --> 0
0 < 1 > 1 --> 1
0 < 1 > 0 --> 1
0 < 0 > 1 --> 1[-F1F1]
0 < 0 > 0 --> 0
* < - > * --> +
* < + > * --> -
```

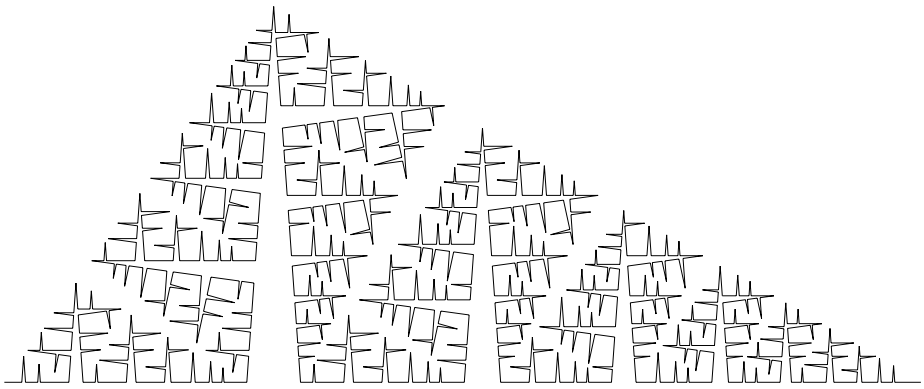


Figure 20: “Row of Trees” generated using a parametric L-system

```
endlsystem
```

The productions of this L-system have the following structure:

```
      1       <       1       >       1       -->      0
Left Context        Predecessor        Right Context          Successor
```

This production will replace a given `1` with `0` only if it is preceded by `1` and followed by `1`. The command `ignore: +-F` tells `cpfg` not to consider the `+`, `-` and `F` symbols when matching contexts. Other examples of context sensitive L-systems are given in Section 1.8 of *The Algorithmic Beauty of Plants* [7].

The production:

```
* < - > * --> +
```

lists `*` for both the left and right context, and consequently, will match a `-` symbol with *any* context. The `*` is not required. The following productions are equivalent:

```
* < - > * --> +
    - > * --> +
* < -     --> +
    -     --> +
```

19 Parametric L-systems

Figure 20 shows a fractal generated with a parametric L-system.

19.1 rowoftrees.l

```
#define STEPS 7
#define a 86
#define p 0.3
#define d1 2
#define d2 1
#define d3 0

#define q (1-p)
#define h ((p*q)^0.5)

lsystem: 0
derivation length: STEPS
axiom: -(90)F(1)

F(x) : x>0.05 --> F(x*p)+(a)F(x*h)-(a+a)F(x*h)+(a)F(x*q)

endlsystem
```

This L-system makes use of parameters to control the distance moved by the turtle. The initiator (production predecessor) is the hypotenuse AB of a right triangle ABC (Figure 21). The first and the fourth edge of the generator subdivide AB into segments AD and DB, while the remaining two edges traverse the altitude CD in opposite directions. From elementary geometry it follows that the lengths of these segments satisfy the equations

$$q = c - p \qquad \text{and} \qquad h = \sqrt{pq}.$$

In the next derivation step, the four edges of the generator can be associated with four triangles that are similar to ABC.

20 Global variables in parametric L-systems

Figure 22 shows a fractal generated with a parametric L-system.

20.1 flake.l

```
#define STEPS 5
#define af 1.08
#define hf 0.41

lsystem: 0
Start: {h = 20; a = 66;}
EndEach: {h = h*hf; a = a*af;}
derivation length: STEPS
axiom: -(90)F(60)
```

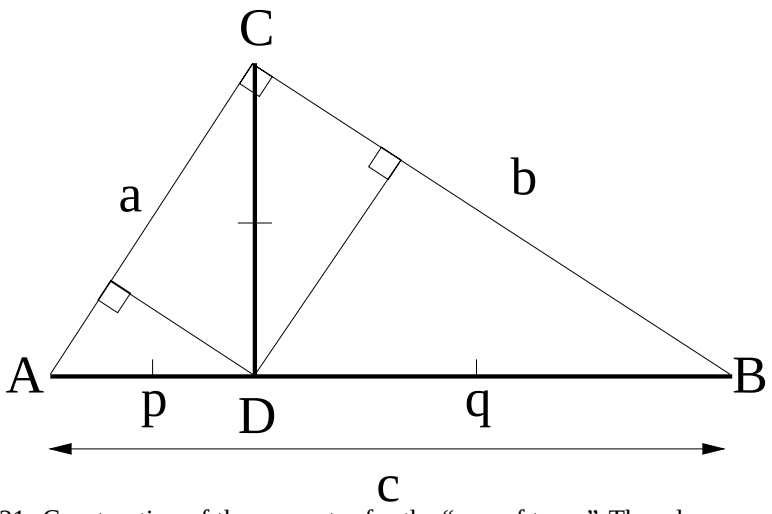


Figure 21: Construction of the generator for the "row of trees." The edges are associated with triangles indicated by ticks.

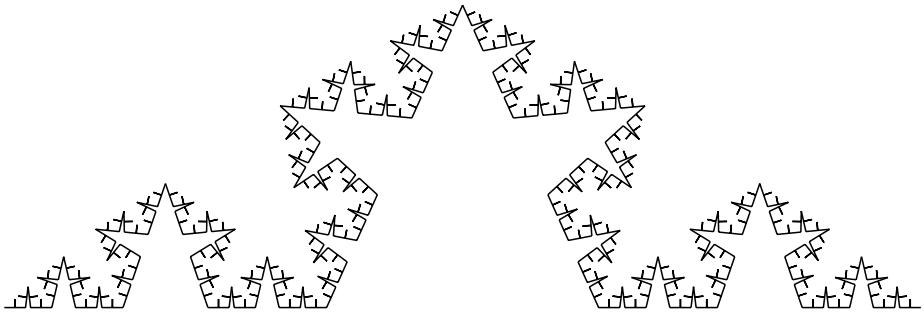


Figure 22: "Snowflake" generated using a parametric L-system

```
F(x) --> F(x/2-h/tan(a))+(a)
         F(h/sin(a))-(2*a)
         F(h/sin(a))+(a)
         F(x/2-h/tan(a))

endlsystem
```

This L-system makes use of two global variables, h and a. The line:

```
Start: {h = 20; a = 66;}
```

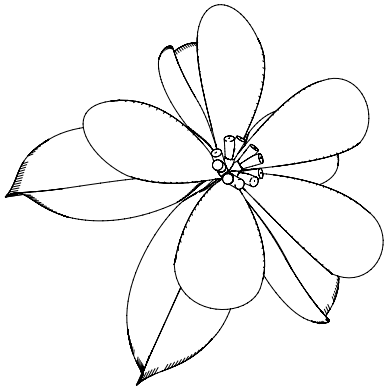


Figure 23: A stylized apple blossom

sets the initial values at the begining of the derivation, and:

```
EndEach: {h = h*hf; a = a*af;}
```

updates the values at the end of each step.

21 Incorporation of predefined surfaces

Figure 23 shows a model which uses predefined surfaces.

21.1 blossom.l

```
#define SIZE 100
lsystem: 0
derivation length: 3
axiom: /(154)B
B --> [&(72)#;;F(5*SIZE),,!K]
K --> [,S/(72)S/(72)S/(72)S/(72)S]
S --> [^(103)~c][^(72)~p][,^(34)F(SIZE)#,[-F(SIZE)][+F(SIZE)]]
endlsystem
```

21.2 blossom.v

```
angle factor: 21
initial color: 120
color increment: 32
```

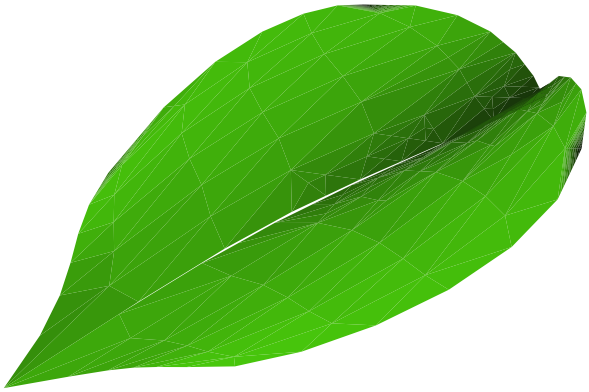


Figure 24: Apple Leaf

```
initial line width: 20.0
line width increment: 3.0
projection: parallel
front distance: -26000.0
back distance: 26000.0
scale factor: 0.7
z buffer: on
cue range: 0
shade mode: 3
light direction: 1.0,0.0,0.0
diffuse reflection: 25
tropism direction: 0.0,2.0,0.0
initial elasticity: 0.02
elasticity increment: -0.02
surface ambient: .15
surface diffuse: .85
line: ~ line.s 1.0
surface: c leaf.s 350
surface: p petal.s 500
```

21.3 leaf.s

This section specifies the surface of a leaf such as the one shown in Figure 24. It consists of two patches, but there is no interpolation of shading between them.

```
-28.72 25.68   -20.64 81.29   -7.50 21.85
CONTACT POINT  X: 0.00 Y: -20.00 Z: 0.00
```

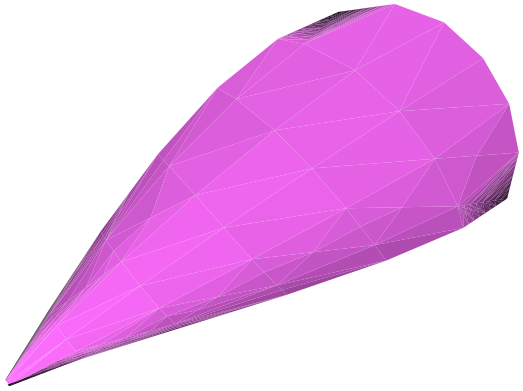


Figure 25: Apple Petal

```
END POINT  X: 0.00 Y: -19.61 Z: 0.00
HEADING  X: 0.00 Y: 0.99 Z: -0.14
UP  X: 0.02 Y: -0.14 Z: -0.99
SIZE: 101.93
Patch_1
TOP COLOR: 298 DIFFUSE: 0.75 BOTTOM COLOR: 298 DIFFUSE: 0.75
AL: ~ A: ~ AR: ~
L: ~ R: ~
BL: ~ B: ~ BR: ~
-15.51 0.84 -5.00  -22.12 13.03 -5.00  -28.72 42.48 -5.00  -17.84 60.08 -2.50
-7.35 -10.67 -5.00  -20.00 10.00 -7.50  -20.00 36.05 -5.00  -8.90 75.60 0.00
-10.00 -5.00 -5.00  -10.00 15.00 -5.00  -10.00 40.00 -2.50  -1.52 71.71 2.50
0.00 -20.64 0.00  0.00 16.75 0.00  0.00 16.75 0.00  0.00 81.29 21.85
Patch_2
TOP COLOR: 298 DIFFUSE: 0.75 BOTTOM COLOR: 298 DIFFUSE: 0.75
AL: ~ A: ~ AR: ~
L: ~ R: ~
BL: ~ B: ~ BR: ~
0.00 -20.64 0.00  0.00 25.00 0.00  0.00 16.41 0.00  0.00 81.29 21.85
4.31 -5.00 -5.00  10.00 22.17 -5.00  10.00 40.00 -2.50  1.20 70.24 2.50
8.59 -7.50 -7.50  15.74 10.00 -7.50  20.00 25.00 -5.00  12.08 68.89 0.00
14.80 4.23 -5.00  25.68 28.94 -5.00  21.80 45.19 -5.00  19.08 51.62 -2.50
```

21.4 petal.s

This section specifies the surface of a leaf such as the one shown in Figure 25.

```
-5.93 34.60   8.43 50.90   -1.47 7.91
CONTACT POINT  X: 23.63 Y: 8.49 Z: 0.13
END POINT  X: 23.63 Y: 8.70 Z: 0.37
HEADING  X: 0.00 Y: 1.00 Z: 0.00
UP  X: 0.01 Y: 0.00 Z: 1.00
SIZE: 39.59
Petal_1
TOP COLOR: 0 DIFFUSE: 0.00 BOTTOM COLOR: 0 DIFFUSE: 0.00
AL: ~ A: ~ AR: ~
L: ~ R: ~
BL: ~ B: ~ BR: ~
22.64 8.49 0.00  20.53 15.22 4.03  13.10 30.26 5.80  15.07 40.29 4.35
22.60 8.49 0.00  19.67 15.00 1.97  19.67 30.00 2.14  16.65 49.99 4.83
23.42 8.49 0.00  26.53 15.00 1.80  26.19 30.00 1.97  28.94 50.90 -1.47
22.96 8.43 0.00  26.03 15.00 3.69  34.60 30.75 7.91  32.18 41.26 1.28
```

22 More predefined surfaces

Figure 26 shows a lilac inflorescence, incorporating predefined surfaces. The specification files can be found in `$VLABHOME/oofs/ext/examples/ext/lilac`.

23 Use of sub-L-systems

Figure 27 shows a model of the sedge *Carex laevigata*. In this model, sub-L-systems are used to generate the male and female spikes. The main L-system, shown in Section 23.1 uses the `?` symbol to incorporate the productions from the sub-L-systems included from the files `female.l`, `male.l` and `leaf.l`.

23.1 sedge.l

```
/* internode growth rate */
#define RATE 1.02
/*  For a doubling in branch length we want 1.26 times the width */
/*  The exponent is equivalent to log(1.26)/log(2) approximately */
/*  for 1.1 we use an exponent of .1375 */
/*  for 1.2 we use an exponent of .2630 */
/*  for 1.26 we use an exponent of .3334 */
/*  for 1.3 we use an exponent of .3785 */
#define STEMRATE 1.06
/* width of stem at start of internode */
#define STEMWIDTH .0075
/* Sub L-systems for female spike, male spike and leaf */
```

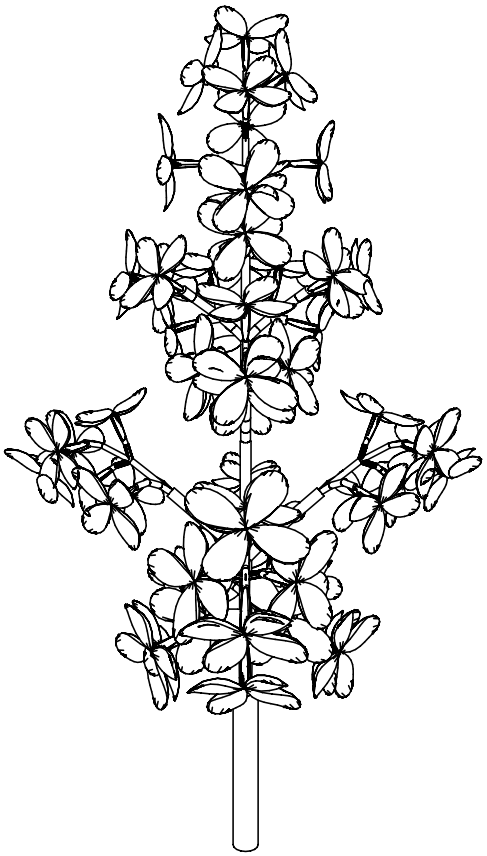


Figure 26: A lilac inflorescence

```
#define F_SPIKE ?(2,1.25)axiom$
#define M_SPIKE ?(3,1.25)axiom$
/* leaf L-system parameters: starting delay, time to turn, and new elasticity */
#define LEAF ?(4,1)axiom((a-10),a,(a-13)/100)$

lsystem: 1
/* nice derivation length 90+ (95?) */
derivation length: 95
axiom: /(30)+(10)#(STEMWIDTH)A(4,4)
* < A(a,t) > * : a==30 --> F(1)/(137.5)M_SPIKE
* < A(a,t) > * : t<10 --> F(1)A(a+1,t+1)
```

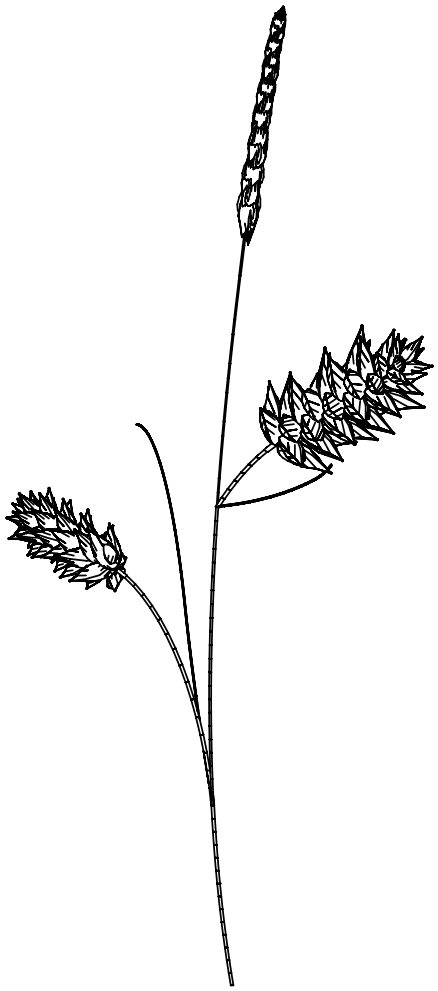


Figure 27: *Carex laevigata*

```
* < A(a,t) > * : t==10 --> F(1)/(137.5)[L(a)][S(a)]#(STEMWIDTH)A(a+1,0)
* < #(d) > * : d<200 --> #(d*STEMRATE)
/* ! is used here so that width won't be increased */
* < S(a) > * : * --> [^(25)_(0-.1)!(.3)F((30-a)/5)F((30-a)/5)_(0)F_SPIKE]
* < L(a) > * : * --> [^(60)!(.1)LEAF]
* < F(t) > * : t<2 --> F(t*RATE)
* < F(t) > * : !(t<2) --> F(t*RATE/2)F(t*RATE/2)
```

```
endlsystem
#include "female.l"
#include "male.l"
#include "leaf.l"
```

23.2 female.l

This L-system contains the line:

```
lsystem: 2
```

and generates the image shown in Figure 28. It is included into the main L-system with the turtle symbols `?(2,1.25)`, specifying that L-system 2 is to be included and scaled by a factor of 1.25.

```
#define I_RATE 1.01
/* internode growth rate */
#define S_RATE 1.05
/* seed growth rate */
lsystem: 2
derivation length: 76
axiom: ////F(5)axiom
* < A(t) > * : t<75 --> F(.2)[B]/(137.5)A(t+1)
* < B > * : * --> &(35)[~f(1)]/(180)[~f(1)][~c(1)#(.1)F(.5)]
* < F(t) > * : t<1 --> F(t*I_RATE)
* < &(a) > * : a<50 --> &(a*S_RATE)
* < ~f(t) > * : t<2 --> ~f(t*S_RATE)
* < ~c(t) > * : t<2 --> ~c(t*S_RATE)
* < axiom > * : * --> [&(30)/(180)~f(2.25)#(.1)F(.5)]F(.1)/(180)
                       [&(30)/(180)~f(2.25)#(.1)F(.5)]/(137.5)A(0)
endlsystem
```

24 L-System defined surfaces

Figure 29 shows several stages in the development of a *Lychnis coronaria* flower. The specification files can be found in:

```
$VLABHOME/oofs/ext/examples/ext/lychnis
```

This model uses Bezier surfaces specified by the L-system using the `@PD(`*i*,*s*,*t*`)` symbols. The file `lychnis.l` contains many parameters controlling both the timing of the development, and the angles and sizes of various components.

Figure 28: Female spike

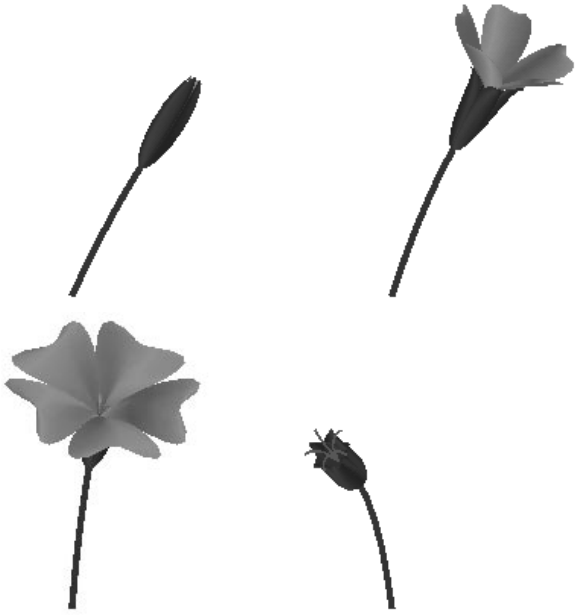


Figure 29: Lychnis

25 Other examples

Other examples, imllustrating homomorphism, decomposition, the use of generalized cylinders, and various other features can be found either on the system (see the objects

noted at the margins in places where various features are described) or in [2, 3].

References

[1] HANAN, J. S. *Parametric L-systems*. PhD thesis, University of Regina, Regina, Saskatchewan, Canada, 1992.

[2] MĚCH, R. *Modeling and Simulation of the Interaction of Plants with the Environment using L-systems and their Extensions*. PhD thesis, The University of Calgary, Calgary, Canada, November 1997.

[3] MĚCH, R., PRUSINKIEWICS, P., AND HANAN, J. Extensions to the graphical interpretation of L-systems based on turtle geometry. Tech. Rep. 97/599/01, Dept. of Computer Science, The University of Calgary, Calgary, Canada, 1997.

[4] MĚCH, R., AND PRUSINKIEWICZ, P. Visual models of plants interacting with their environment. *Computer Graphics (SIGGRAPH '96 Conference Proceedings)* (August 1996), 397–410.

[5] PRUSINKIEWICZ, P., AND HANAN, J. L-systems: From formalism to programming languages. In *Lindenmayer systems: Impact on theoretical computer science, computer graphics, and developmental biology*, G. Rozenberg and A. Salomaa, Eds. Springer-Verlag, Berlin, 1992, pp. 193–211.

[6] PRUSINKIEWICZ, P., JAMES, M., AND MĚCH, R. Synthetic topiary. *Computer Graphics (SIGGRAPH '94 Conference Proceedings) 38* (July 1994), 351–358.

[7] PRUSINKIEWICZ, P., AND LINDENMAYER, A. *The algorithmic beauty of plants*. Springer-Verlag, New York, 1990 (second printing 1996). With J. S. Hanan, F. D. Fracchia, D. R. Fowler, M. J. M. de Boer, and L. Mercer.

[8] WOO, M., NEIDER, J., AND DAVIS, T. *The OpenGL Programming Guide, Second Edition*. Addison-Wesley.

A L-system Input Grammar

Lfile → $Lsystems\ BlankLines$

$Lsystems$ → $Lsystems\ Lsystem$
| */* empty */*

Lsystem → *Header Productions Decomposition Homomorphism* `endlsystem` *<newline>*

$Homomorphism$ → `homomorphism` $HomoWarning < newline > HomoItems\ Productions$
| */* empty */*

$HomoItems$ → $HomoItems\ HomoItem$
| */* empty */*

$HomoItem$ → $HomoSeed$
| $ProdDepth$

$HomoWarning$ → `: warnings`
| `: no warnings`
|

$HomoSeed$ → `seed` $Expression < newline >$

$Decomposition$ → `decomposition` $DecompWarning < newline > DecompItems\ Productions$
| */* empty */*

$DecompWarning$ → `: warnings`
| `: no warnings`
|

$DecompItems$ → $ProdDepth$
| */* empty */*

$ProdDepth$ → `depth` $Expression < newline >$

Header → *BlankLines Label Items Axiom*

BlankLines → {*BlankLine*}

BlankLine → *<newline>*

Label → `lsystem:` *Characters <newline>*

Items → {*Item*}

Item → *Seed*
| *Dlength*
| *Ignore*
| *Consider*
| *BlankLine*
| *Defines*
| *Startblock*
| *Endblock*
| *Starteach*
| *Endeach*

Seed → `seed:` *Characters* <*newline*>

Dlength → `derivation length:` *Expression* <*newline*>

Consider → `consider:` *Characters* <*newline*>

Ignore → `ignore:` *Characters* <*newline*>

Characters → {<*character*>}

Startblock → `start:` *Block* <*newline*>

Endblock → `end:` *Block* <*newline*>

Starteach → `start each:` *Block* <*newline*>

Endeach → `end each:` *Block* <*newline*>

Block → { *Statements* }

Statements → {*Statement*}

Statement → *Assignment*
| $Procedure$
| $IfStatement$
| $WhileStatement$
| $DoStatement$

Assignment → *LHS* = *Expression* `;` $BlankLines$

LHS → <*identifier*>
| <*identifier*> $ArrayRef$

$ArrayRefs$ → { $ArrayRef$ }

$ArrayRef$ → `[` $Expression$ `]`

$Procedure \rightarrow Expression$ `;` $BlankLines$

$IfStatement \rightarrow$ `if (` $Expression$ `)` $BlankLines\ Block\ BlankLines$
| `if (` $Expression$ `)` $BlankLines\ Block$ `else` $BlankLines\ Block\ BlankLines$

$WhileStatement \rightarrow$ `while (` $Expression$ `)` $BlankLines\ Block\ BlankLines$

$DoStatement \rightarrow$ `do` $BlankLines\ Block$ `while (` $Expression$ `) ;` $BlankLines$

$Defines \rightarrow$ `define` $DefineBlock < newline >$

$DefineBlock \rightarrow$ { $BlankLines\ DefStatements$ }

$DefStatements \rightarrow \{DefStatement\}$

$DefStatement \rightarrow ArrayDefStatement$
| $ExternalDefStatement$

$ArrayDefStatement \rightarrow$ `array` $ArrayDefs$ `;` $BlankLines$

$ArrayDefs \rightarrow ArrayDef$ { , $ArrayDef$ }

$ArrayDef \rightarrow$ <identifier> $ArrayDims$
| <identifier> $ArrayDims$ = $ArrayInitBlock$

$ArrayDims \rightarrow \{ArrayDim\}$

$ArrayDim \rightarrow$ `[` $Expression$ `]`

$ExternalDefStatement \rightarrow$ `external`$ExternalDefs$ `;` $BlankLines$

$ExternalDefs \rightarrow ExternalDef$ { `,` $ExternalDef$}
| $ExternalDef$

$ExternalDef \rightarrow$ <identifier>
| <identifier> $ArrayDims$

Axiom → `axiom:` *Modules* <*newline*>

Productions → {*Production*}

Production → *BlankLine*
| *Predecessor* [*Conditional*] `-->` *Successor* <*newline*>
| *Predecessor* [*Conditional*] `-o>` *Successor* <*newline*>

Predecessor → *Strictpred*
| *Lcontext* < *Strictpred*
| *Strictpred* > *Rcontext*
| *Lcontext* < *Strictpred* > *Rcontext*

Lcontext → *
| *FormalModules*

Strictpred → *FormalModules*

Rcontext → *
| *FormalModules*

Conditional → `:` *Condition*
| `:` *Precondition Condition*
| `:` *Condition Postcondition*
| `:` *Precondition Condition Postcondition*
|

Precondition → *Block*

Postcondition → *Block*

Condition → *
| *Expression*

Successor → *StrictSucc*
| *StrictSucc Probability*

StrictSucc → *
| *Modules*

Probability → : *Expression*

FormalModules → {*FormalModule*}

FormalModule → *Symbol*
| *Symbol* (*FormalParameters*)

Modules → {*Module*}

Module → *Symbol*
| *Symbol* (*Parameters*)

Symbol → <*character*>

FormalParameters → *FormalParameters* {, *FormalParameter*}

FormalParameter → <*identifier*>

Parameters → *Expression* {, *Expression*}

Expression → *Expression* || *Expression*
| *Expression* && *Expression*
| *Expression* == *Expression*
| *Expression* != *Expression*
| *Expression* <> *Expression*
| *Expression* < *Expression*
| *Expression* <= *Expression*
| *Expression* > *Expression*
| *Expression* >= *Expression*
| *Expression* + *Expression*
| *Expression* - *Expression*
| *Expression* * *Expression*
| *Expression* / *Expression*
| *Expression* % *Expression*
| *Expression* ^ *Expression*
| - *Expression*
| ! *Expression*
| (*Expression*)
| *Function*
| *Name*
| *Value*
| *LValue*
| *String*

Function → *FunctionName* (*Expression*)

FunctionName → `tan`
| `sin`
| `cos`
| `atan`
| `asin`
| `acos`
| `ran`
| `nran`
| `bran`
| `biran`
| `srand`
| `exp`
| `log`
| `floor`
| `ceil`
| `trunc`
| `fabs`
| `sign`
| `stop`
| `sqrt`
| `printf`
| `fprintf`
| `fopen`
| `fclose`
| `fflush`
| `fscanf`

Value → <*number*>

Name → <*identifier*>

$LValue$ → & <*identifier*>
| & <*identifier*> $ArrayRefs$

String → " <*string*> "

Index