The Virtual Laboratory

LPFG
Reference Manual

Last updated: November 30, 2021

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

Contents

1 Introduction

lpfg is a plant modeling program based on the formalism of L-systems. Models are defined using the L+C language, which extends the syntax of C++ to include constructs inherent in L-systems.

This is a reference manual with limited examples. Sample *vlab* objects are indicated in some sections to provide more detailed usage.

1.1 Running *lpfg*

lpfg is included with the *vlab* distribution, and is normally run from an object's menu within *vlab*. The command line below is defined in the object's *specification* file.

`lpfg [-a] [-b] [-c] [-cleanEA20] [-cn] [-d]` `[-dll` *filename.dll* `]` `[-ds] [-dtf] [-dtfes] [-lp` *path* `]` `[-o` *filename.dll* `]` `[-out` *filename*`] [-q] [-rmode` *mode* `]` `[-v] [-w` *w h*`] [-wnb] [-wp` *x y* `]` `[-wpr` *x y* `]` `[-wr` *w h* `]` [*animation*`.a`] [*colormap*`.map`] [*contour*`.con`] [*contourset*`.cset`] [*environment*`.e`] [*function*`.func`] [*functionset*`.fset`] [*material*`.mat`] [*parameter*`.vset`] [*timeline* `.tset`] [*view*`.v`] *Lsystem*`.l`

Command line options may appear in any order. The only mandatory parameter is the L-system file, *Lsystem*`.l`, which contains the L+C code for the model.

1.1.1 Command line options

Parameter	Description
`-a`	Start *lpfg* in animate mode, using the information in *animationfile.a*. Only `first frame` steps are performed, as opposed to `derivation length` steps.
`-b`	Start *lpfg* in batch mode: no window is created. The simulation is performed and the final content of the string is stored in the file specified by the `-out` option. Only module names are stored in the file. This mode cannot be combined with the `-a` option.
`-c`	Compile the L-system to the .dll file only. Do not run the simulation. There is no translation of L+C to C++, and the C++ compiler is not invoked. The default file is `lsys.dll`, but this can be changed with the `-dll` option.
`-cleanEA20`	Zero the array of (20) return values from an environmental program before the next iteration. This is used to ensure the array is clear if an environmental program returns an arbitrary number of values. See the *Environment Progams* manual for more information.
`-d`	Start *lpfg* in debug mode. Information regarding the execution of the program is sent to the standard output. This mode is intended to be used by developers.
`-dll` *filename.dll* `-o` *filename.dll*	Use *filename.dll* rather than the default .dll file (`lsys.dll`) with the `-c` option.
`-ds`	Output the current string to the console after each derivation step, before the `interpretation:` block.
`-dtf`	Output the final interpreted string to a file (i.e. after the `interpretation:` block). Uses the filename: *Lsystemfile*`.str`.
`-dtfes`	Output the interpreted string after each derivation step, to separate files. Uses the filename: *Lsystemfile*`.str`, but with an 8-digit suffix. For example the first string output would be in *Lsystemfile*`00000001.str`.

Parameter	Description
`-out` *filename*	In batch mode, use *filename* for the output string. This will be a text file. In regular (not batch) mode, run the model to the end and produce a single image in *filename*, based on the extension: bmp, jpg, pdf, png, tiff. The model window will close on completion. Also see the Save As... menu item (Section 1.2.2).
`-q`	Start *lpfg* in quiet mode. All messages, including warnings and errors, are suppressed.
`-rmode` *mode*	Define the method for re-reading input files. The values of *mode* are: `expl` = explicit `cont` = continuous `trig` = triggered The refresh mode may also be set with the Refresh mode menu item (Section 1.2.2), and within a *vlab* object's specification file (see the *Vlab Framework* manual).
`-v`	Start *lpfg* in verbose mode. Displays additional information/warning messages.
`-w` *w h*	Specify the width *w* and height *h* of the *lpfg* output window in pixels. Use either this option or `-wr` but not both.
`-wnb`	Create the *lpfg* window without borders or title bar, and do not display the output console window. This mode is useful for demonstration purposes.
`-wp` *x y*	Specify the *lpfg* window's top left corner position *(x,y)* in pixels relative to the top left corner of the screen. Use either this option or `-wpr` but not both.
`-wpr` *x y*	Specify the relative window position of the *lpfg* window: *x* and *y* are numbers between 0 and 1, and represent the position of the top left corner of the window relative to the top left corner of the screen. Use either this option or `-wp` but not both.
`-wr` *w h*	Specify the relative window size of the *lpfg* window: *w* and *h* parameters are numbers between 0 and 1 and specify the relative size of the *lpfg* output window with respect to the screen. Use either this option or `-w` but not both.

1.1.2 Input files

Input files are recognized based on their extension. The *lpfg*-specific input files, *animation*`.a` and *view*`.v`, are described in Sections 8.1 and 8.2 respectively. Other file types can be found in the *Vlab Tools* manual.

When the refresh mode is set to Triggered/Continuous, either from the command line (-rmode) or from the menu, *lpfg* turns on file monitoring to watch for changes in any of its input files. See Section 1.3 for more information on file monitoring.

Filename	Description
animation`.a`	Define the parameters for controlling animation of the model. See Section 8.1.
view`.v`	Define the drawing and viewing parameters, including setting the view, rendering, surfaces, etc. See Section 8.2.
colormap`.map` *material*`.mat`	Specify 256 colors or 256 materials, respectively. A colormap is generally used to create schematic images, whereas material files are used to create realistic images. If no colormap file or material file is specified, the default colormap is used. See Section 5.3.2 for information on how to use the colors within *lpfg*, and the *palette* and *medit* tools in the *Vlab Tools* manual.

Filename	Description
contour.con *contourset*.cset	Specify contours defined as planar B-spline curves. The curves are considered as cross-sections of generalized cylinders. There may be multiple *contour*.con files, each containing a single contour definition, but only one *contourset*.cset file containing multiple contour definitions. See Section 6.3 for information on how to access the contours within *lpfg*, and the *cuspy* and *gallery* tools in the *Vlab Tools* manual.
function.func *functionset*.fset *timeline*.tset	Specify functions of one variable. The functions are defined as B-spline curves constrained in such a way that they assign exactly one y to every x in the normalized function domain [0,1]. There may be multiple *function*.func files, each containing a single function definition, but only one *functionset*.fset file containing multiple function definitions, and one *timeline*.tset file containing functions constrained by a timeline rather than the normalized function domain. See Section 6.6 for information on how to access the functions within *lpfg*, and the *funcedit, gallery*, and *timeline* tools in the *Vlab Tools* manual.
parameter.vset	Define parameters that can be read from the L-system without recompiling. See Section 6.7 for the val function used to access the parameters within *lpfg*, as well as the file format.
environment.e	Specify the environmental program and its parameters. See the *Environment programs* manual for more information.

1.2 User Interface

When *lpfg* is opened, it normally runs the L-system and draws the final interpreted string. (Some command line options, such as -a and -b, produce different results.) Once the model is drawn it is possible to manipulate the view of the L-system, or make adjustments to it.

1.2.1 View manipulation

The view in the output window is manipulated using both the mouse buttons and the SHIFT and COMMAND keys within the *lpfg* window:

Action	Key & Mouse	Description
Rotation	Left mouse button	Rotate around the Y axis by moving the mouse horizontally, and around the X axis by moving the mouse vertically.
Roll	SHIFT key and middle mouse button	Roll clockwise around the Z axis by moving the mouse to the right, and roll counterclockwise by moving the mouse to the left.
Zoom	COMMAND key and left or middle mouse button	Zoom in by moving the mouse up, and zoom out by moving down.
Pan	SHIFT key and left mouse button	Move model in all directions using the mouse.
Change frustrum angle	COMMAND key and middle mouse button	Increase the angle by moving the mouse up, and decrease the angle by moving down. This operation has an effect only in perspective projection mode.

1.2.2 Main menu

A menu of options is displayed by clicking the right mouse button within the *lpfg* window. It includes the following menu items:

Menu item	Description
New model	Re-read all input files, recompile the L-system, reset the view, and run the simulation. This is equivalent to restarting the model from the object menu, but uses the existing *lpfg* window rather than opening a new one.
New L-system	Re-read all input files, except the view and animation files, and re-run the simulation. The view is not reset.
New run	Re-run the simulation without re-reading (and recompiling) the L-system file, or re-reading the view and animation files. Other parameter files (colors, functions, etc.) are re-read.
New view	Re-read the view file, along with the materials/colormap, surfaces, and textures, and reset the view without re-running the simulation.
New rendering	Re-read the same files as New view, but reset the rendering parameters only, without changing the view or re-running the simulation.
Save	Save the current state with the same name as the L-system file, with the given extension. The default is to save the image in PNG format. For other options, use Save as ...
Save as ...	Save the current state with the same name as the L-system file, in one of several image formats (BMP, JPG, PDF, PNG, TIF), or as Postscript, POV-Ray, Rayshade, or View parameters.
String > Input	Input the binary form of an L-system string from the file, *lsystemfile*`.strb`. Generally, this is a file created earlier by String > Output.
String > Output	Output the current string to the file, *lsystemfile*`.strb`, in binary form.
Animate	Switch to animate mode and re-run the model, stopping and drawing the interpreted string at the `first frame` as defined in the animation file. Additional menu items are added to the menu (Section 1.2.3).
Refresh mode	Set the mode used to refresh the input files. The default is Explicit, where the menu options above must be used to re-read each file. Triggered/Continuous mode monitors all files for changes (see Section 1.3).
Exit	Quit *lpfg*.

In summary, the New commands include the following actions:

Menu item	Re-read (& recompile) L-system	Re-read view	Reset view	Reset rendering	Re-read colors, surfaces, textures	Re-read functions, contours, timeline, parameters
New model	x	x	x	x	x	x
New L-system	x				x	x
New run					x	x
New view		x	x	x	x	
New rendering		x		x	x	

For *rayshade* output, the projection type must be set to `perspective` using the view file command `projection` (Section 8.2.1).

1.2.3 Animate menu

When Animate is selected from the menu, or the `-a` option is included on the command line (Section 1.1.1), the model is re-interpreted, stopping and drawing after the `first frame` defined in the animation file (Section 8.1), and the following menu items are added:

Menu item	Description	Keyboard shortcut
Step	Advance the simulation and redraw. This may correspond to more than one derivation step if the `step` parameter in the animation file is greater than 1.	Cmd-F
Run	Start or resume the animation.	Cmd-R
Forever	Start or resume the animation. After the last frame is reached the animation returns to the `first frame` and continues.	Cmd-V
Stop	Stop the animation.	Cmd-S
Rewind	Reset the animation to the `first frame`.	Cmd-W
Clear	Clear and redraw the latest frame. This is used if the `clear between frames:` parameter in the animation file is set to `no`.	
New animate	Re-read the animation file. Changes take effect when the simulation is re-run.	
Start recording	Record each frame of the animation as it is displayed, using the current file format specified in the Save option. To save each frame in a separate file, use the Save as option and set the `Numbering` checkbox.	
Don't animate	Stop the animation, and return to the original menu. Display the model at the `first frame` as defined in the *animationfile*.	

1.3 File monitoring

When Refresh mode is set to Triggered/Continuous, either from the command line (-rmode) or from the menu, *lpfg* turns on file monitoring to watch for changes in any of its input files. This allows changes to be made in the simulation as soon as a file is updated.

When a file change occurs, the following action is taken by *lpfg*:

File changed	Action
L-system	New L-system
View	New view
Animate	Rereads file only.
Colormap Material	New rendering
Surface Texture	New rendering
Function Contour Timeline Parameters	New run

2 The L-system file

L-system files use the L+C modeling language. It is a declarative language which combines L-system constructs (notably, modules and productions) within the general-purpose programming language C++. The principle advantage of this hybrid approach is that the expressive power of C++ can be used in L+C programs, making it easier to develop complex models.

A typical L+C program file has the following format:

```
#include <lpfgall.h>
//  data structure declarations
//  module declarations
//  function declarations
derivation length:  expression;
axiom:  module list;
//  productions
```

The three statements, `#include`, `derivation length` and `axiom` are mandatory, as well as declarations of all user-defined modules in the axiom and production(s).

All components of the program may appear in any order except for the following restrictions:

- The `#include` statement should be the first line in the file. It contains embedded header files with declarations and definitions used by *lpfg* and the L2C translator, including predefined types.[1]
- All elements referred to in a statement must be declared beforehand. This includes:
  - Types used as parameters of a module - must be declared before the module is declared.
  - Modules that appear in an `ignore` or `consider` statement - must be declared before the statement.
- Productions are matched in the order in which they are declared.

2.1 Derivation Length

This statement specifies the number of derivation steps in the L-system, and has the format:

```
derivation length: expression
```

There are no restrictions on *expression* except that it must evaluate to an integer. For example:

```
derivation length: 5*k+2
```

is valid assuming `k` has a predefined integer value.

Some care should be taken that the value is constant as the expression may be evaluated more than once and the behaviour of *lpfg* is undefined if the value changes.

2.2 Axiom statement

The syntax of an `axiom` statement is:

```
axiom: module list;
```

where *module list* is a sequence of modules. Examples of valid axioms are:

[1] Most predefined types are described in this manual. For additional information see the `lintrfc.h` file.

```
axiom: A(1,2) B() A(0,0);
axiom: A(idx*2,(int)(sin(x*M_PI));
```

There are no commas between the modules in the list. If a module has no parameters, the parentheses may be omitted. For example, the first axiom above could be written as:

```
axiom: A(1,2) B A(0,0);
```

All modules used in the axiom must be declared beforehand. See the next section for module declarations.

2.3 Module and type declarations

2.3.1 Modules

L+C requires that all modules be declared. Many standard modules are predefined (see Section 5) and, therefore, do not need to be declared. The syntax for declaring a new module is:

`module` *name*`(` *parameters* `);`

where *name* is the module name, and *parameters* is a list of the parameter types. For example:

```
module A(int, int);
module B();
module C(float, string);
```

If a module has no parameters, the parentheses can also be omitted. For example, module `B()` above can be declared as:

```
module B;
```

Note that, unlike function arguments, module parameters have no names. Thus the declaration `module A(int id, int age)` is illegal. However, comments may be used to note the parameter names to be used:

```
module A(int /*id*/, int /*age*/);
```

Also note that a module name cannot be used twice, even with different types or numbers of parameters.

2.3.2 Types

All user-defined types (such as `string` above) must be defined before being used in a module declaration. In addition, each type must be a single identifier; compound types such as `char*` or `unsigned int` are not allowed. To use these types, include a `typedef` statement to define a single name:

```
typedef char* string;
typedef unsigned int uint;
```

3 Productions

Productions define the structure of the L-system string over time by specifying the fate of modules with each derivation step. A production has two parts: the *predecessor* defines the module to be changed and the context it must be found in; and the *production body* defines how the predecessor will change in the next derivation step. The syntax of a production is:

predecessor: { *production body* }

3.1 The predecessor

3.1.1 The strict predecessor

The predecessor of a production contains, at a minimum, the *strict predecessor*. This is the module or sequence of modules which, if the production is applied, will be replaced by new modules in the next derivation step. Examples of valid productions containing only a strict predecessor include:

```
F(x): { ... }
A(age, length) B(): { ... }
```

Module parameters must be listed and given unique names, even if they are not used in the production body. Also, unlike module declarations and the `axiom`, a module with no parameters must be followed by parentheses `()`.

3.1.2 Left and right context

In addition to the strict predecessor, a production may also list a context to its left or right, or both. These contexts must also be matched within the string for the production to be applied, although only the strict predecessor will be replaced. The syntax is:

left context `<` *strict predecessor* `>` *right context*:

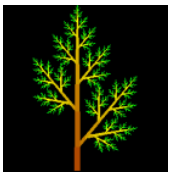
For example, the production

```
F(x) > G(y): { ... }
```

will replace `F(x)` in the next derivation step only if `G(y)` is to the right of `F(x)` in the string. However, `G(y)` is not replaced: it remains in the string unless another production has `G(y)` as the strict predecessor.

3.1.3 Left and right new context

The right and left context constructs above are matched to modules in the input string of the derivation. Since matching is done sequentially from one end of the string to another, it is also possible to match to the newly created modules in the output string. Normally, the string is matched from left to right ("forward") which enables matching to the left new context using the `<<` operator. For example:



See object: NewContext

```
B() << D(): { ... }
```

will replace `D()` in the next derivation step only if `B()` is the last module to be added to the new string so far.

The direction of the derivation can be controlled with the `Backward()` and `Forward()` statements, usually called within a control statement (see Section 4). When the string is matched from right to left ("backward"), the right new context can be used for matching with the operator `>>`. For example:

```
Start: { Backward(); }
E() >> F(): { ... }
```

Note that a production with a new context will never match if the derivation is going in the wrong direction: a new right context will not match if the direction is left to right ("forward"), and a new left context will not match if the direction is right to left ("backward").

"Old" and new contexts can be combined in a single predecessor. For example:

```
Age(age,length) << B() > B(): { ... }
```

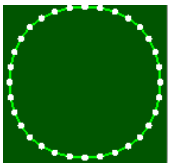
will match the module `B()` if the derivation is proceeding in the forward direction, the last module in the new string is `Age(age,length)`, and the old string has another `B()` to the right of the strict predecessor.

3.1.4 Ring L-systems

A ring L-system provides an alternate topology for context matching in the L-system string. Matching is performed as if the last module in the string and the first module in the string are adjacent, so that the string forms a ring.

For example:

```
Axiom: A B C;
C() < A() : { ... }
```



See object: B-spline

would match the `A` module in the axiom, because its left context is the `C` module at the end of the string.

To specify a ring L-system, include a statement before the `Axiom`:

`ring L-system:` *value*

where *value* is a non-zero number, or an expression returning a non-zero number.

3.2 Production body

If a production predecessor is matched successfully, *lpfg* executes the production body. This block may contain any valid C++ statement. The names given to module parameters in the predecessor act similar to function parameters in a C++ function.

3.2.1 The `produce` statement

The `produce` statement ends execution of the production body (like a `return` statement in a C++ function) and tells *lpfg* what the successor is. Its syntax is:

`produce` *successor*;

where *successor* is a sequence of modules. For example:

```
produce A(newAge,newLength);
produce B() A(x,length*12) B();
```

As with the axiom, there are no commas between modules and, if a module has not parameters, the parentheses may be omitted.

When the `produce` statement is reached the successor is added to the new string and the production ends. However, a production may also end without reaching a `produce` statement: by reaching the end of the production block or by a `return` statement. In that case, the production is considered *not applied*, and *lpfg* will continue to look for a production that does apply to the predecessor. For example, the production:

```
A(age,length):
{
  if (age < 10)
       produce A(age+1,length+dl);
}
```

will only be applied if the first parameter of module `A` is less than 10. Otherwise *lpfg* will continue to look for a production that matches `A(age,length)`. For example, there may be another production such as:

```
A(age,length):
{
  if (age >=10)
       produce B(length);
}
```

A `produce` statement may be found anywhere in the production body where a C++ statement is valid, and there may be multiple `produce` statements, similar to C++ `return` statements. For example the two productions above could be written as:

```
A(age,length):
{
  if (age < 10)
       produce A(age+1,length+dl);
  else
       produce B(length);
}
```

A `produce` statement may also be issued without a successor:

```
produce;
```

In this case the strict predecessor is removed from the string and not replaced.

Note the difference between ending a production with an empty `produce` statement, which removes the predecessor from the string, and a `return` statement, which continues to look for another production to match the predecessor.

3.2.2 The nproduce statement

It is sometimes useful to build a production's successor incrementally. The `nproduce` statement specifies part of a successor, but, critically, does not end the production. It syntax is like that of the `produce` statement:

`nproduce` *module(s)*;

The `nproduce` statement adds the listed modules to the currently defined successor, but does not end execution of the production. A subsequence `produce` statement will add its own argument to the successor, then add the entire successor to the string. If the production body ends without a `produce` statement, the production is not applied, and the partial successor is ignored. For example:

```
A(age,length):
{
   for (int i=0; i<age; i++)
        nproduce B;
   produce C(length);
}
```

will replace `A(age,length)` with a number of `B` modules equivalent to the value of the `age` parameter with a final `C(length)` module. If the predecessor is `A(3,1)`, it will be replaced with:

```
B B B C(1)
```

3.3 Testing context within a production body

The context of the strict predecessor can also be tested within the production body, using one of the four InContext expressions:

`InLeftContext (` *module list* `)`
`InRightContext (` *module list* `)`
`InNewLeftContext (` *module list* `)`
`InNewRightContext (` *module list* `)`

The expressions are of type `bool` and are true if the context matches and false otherwise.

For example, rather than defining the context in the predecessor of the production with:

```
F(x) < G(length) > H(y):  { ... }
```

the context can be tested within the production body as follows:

```
G(length):
{
   if (InLeftContext(F(x)) && InRightContext(H(y))
        { ... }
}
```

This applies to InNewContext expressions as well where

```
F(x) << G(length):  { ... }
```

is equivalent to:

```
G(length):
{
     if InNewLeftContext(F(x))
          { ... }
}
```

See object: InNewContext

Note the following:

- Modules within the InContext constructs are not separated by commas (these are not function calls). They are listed in the same manner as in the predecessor.
- The order in which modules are listed should be the same as in the predecessor.
- Module parameters must be declared beforehand and their types must match the module's declaration. This is different from checking context in the predecessor where the parameters are declared implicitly.
- All the rules of context matching are the same as when matching context in a production's predecessor (see Section 9).

It is possible to combine InContext constructs with a context-sensitive predecessor. The InContext expression will begin matching with the module preceding the left context (`InLeftContext`) or following the right context (`InRightContext`) in the production. For example, the production

```
F(x) < G(length) > H(y):  {
     if InLeftContext( F(x) )
          produce( G(x) );
     else
          produce( G(length+1) );
}
```

will match module `G(3)` in the string `E(1) F(2) G(3) H(4)`. However, the `InLeftContext` expression will then try to match the `E(1)` module. Since it does not find the `F(x)` module, the `else` clause will apply and `G(3)` will be replaced with `G(4)`.

Multiple InContext expressions that evaluate as true will continue to match modules further left (`InLeftContext`) or right (`InRightContext`). Consider the following example:

```
G(length):
{
   if ((InLeftContext(F(f1)) && InRightContext(R(a) F(fr))) ||
       (InLeftContext(F(f1)) && InRightContext(U(b) F(fr))))
           { ... }
}
```

The intention of this code is to consider two cases that have the same left context but different right contexts. However, if the first `InRightContext` expression returns false after evaluating the first `InLeftContext` expression, the second `InLeftContext` expression (after the `||` operator) will try to match the module to the left of the one matched by the first `InLeftContext`. To avoid this issue the production should be rewritten as:

```
G(length):
{
   if InLeftContext(F(f1))
   {
       if (InRightContext(R(a) F(fr)) || InRightContext(U(b) F(fr)))
           { ... }
   }
}
```

Note that the two `InRightContext` expressions will be attempting to match the same module since only one of them will evaluate as true.

In general InContext expressions should be treated as operations that read from a stream: as each expression evaluates as true, the next module in the stream will be available for matching.

4 Control statements

The following statements are used to control when specific productions, modules, and procedures are utilized in the derivation process.

4.1 Start and End statements

There are four statements that define procedures at specific points in an L-system derivation:

- `Start` - called before the first derivation step (i.e. before the output string is initialized from the axiom)
- `StartEach` - called before each derivation step
- `EndEach` - called after each derivation step
- `End` - called after the final derivation step

Each statement has the syntax:

statement name: { *C++ statements* };

For example, to maintain a global variable `steps` equal to the current derivation step, the following statements can be used:

```
int steps;
Start: { steps = 0; }
EndEach:{ steps++; }
```

Note the `End` statement is called after the final derivation step. Therefore, in Animate mode, if the animation is stopped or `Rewind` is used before it reaches the final derivation step, the `End` statement is never called. If the `End` statement runs a vital command (for instance, to close an output file), ensure that the animation is run to the final frame.

4.2 Ignore and Consider statements

By default, all modules are considered when matching contexts (more or less - see Section 9 on how productions are matched). However, there are cases where modules should not be included for the purposes of matching context. There are two statements that can be used for this:

`ignore:` *module list*;

or

`consider:` *module list*;

where *module list* is a sequence of module names. Use the `ignore` statement to list the modules that should be ignored when matching context, or the `consider` statement to list the only modules to be considered when matching context. For example, the code:

```
ignore: A B;
C(1) < D(2) > E(3):  { ... }
```

would be matched to the string: `C(1) A(10) D(2) B(5) E(3)`, since the `A` and `B` modules are ignored. The same effect can be achieved with a `consider` statement:

```
consider: C D E;
C(1) < D(2) > E(3): { ... }
```

In this case the same string would find a match because only the `C`, `D` and `E` modules are considered when matching.

Multiple `ignore` and `consider` statements are allowed within an L-system. Each statement applies to the subsequent productions until another `ignore` or `consider` statement is encountered. To cancel the effect of the last statement, use the empty `ignore` statement:

```
ignore: ;
```

The predefined modules `SB` and `EB` (Section 5.1) are **always** considered. Listing them in an `ignore` or `consider` statement has not effect.

4.3 Decomposition and Interpretation Rules

While productions are rules that specify how an L-system string evolves over time, decomposition rules are applied to decompose modules in the string into sub-modules, and interpretation rules are applied to provide information on how to display the L-system.

4.3.1 Decomposition Rules

In complex L-systems, productions can be used to define modules at a higher level of abstraction with more details specified in decomposition rules, similar to the use of function calls in C++. This provides a clear overview of the algorithm in the productions, with details to follow. Decomposition rules are applied to the L-system string in a *decomposition step* after the axiom and after each derivation step. The syntax is:

```
decomposition:
    predecessor : { successor }
    predecessor : { successor }
    ...
```

where each rule (predecessor/successor) follows the same standards as a production rule.

When the `decomposition:` statement is present in an L-system it indicates that all the following rules are decomposition rules, until the end of the source file, or until a `production:` or `interpretation:` statement is encountered.

For example, a decomposition rule may replace a module by its constituent parts:

```
M(t) : {
     produce I(t)
        SB() Right(45) A(t) EB()
        SB() Left(45) A(t) EB()
        I(t) ;
}
```

The module `M(t)` is replaced in the L-system string by all the modules in the `produce` statement. This successor will then be used in the interpretation step, and for the next derivation.

Decomposition rules can be recursive: the module in the strict predecessor can appear in the successor. However, the default maximum decomposition depth is 1. Therefore, to actually recursively use a decomposition rule, a `maximum depth` statement must be used. It has the syntax:

```
maximum depth:  n
```

where n must be an integer value. Decomposition is performed as long as the string does not contain any modules that can be further decomposed, or until `maximum depth` is reached. Only one instance of a `maximum depth` statement is allowed in an L-system. It is applied to all decomposition rules.

An example of a recursive decomposition rule is as follows:

```
decomposition:
maximum depth: 6;
   A(age):
   {
      if (age > 0)
         produce F(1) A(age-1);
   }
```

This rule will produce a series of `F(1)` modules equal to `age`, to a maximum of 6, ending with module `A`.

4.3.2 Interpretation rules

Interpretation rules are executed only during the interpretation of the string. Modules produced by interpretation rules are not inserted into the string for the next derivation step; they are only used as commands to the turtle when outputting the string. This provides a useful separation between the functional aspects of a model and its graphical interpretation.

An *interpretation step* is performed in the following cases:

- When drawing the model in a window.
- When generating an output file (e.g. a rayshade file).
- When calculating the (axes-aligned) bounding box of the model.
- After the axiom and each derivation step, if any of the production predecessors contain query or communication modules (see Section 5.3.3).

Syntactically, interpretation rules have the same format as decomposition rules, including a `maximum depth` statement for recursive rules:

```
interpretation:
maximum depth: expression;
    predecessor : { successor }
    predecessor : { successor }
    ...
```

Generally, interpretation rules are replacing conceptual modules with predefined modules for turtle interpretation (see Section 5). For example:

```
interpretation:
   A(age,length): { produce Sphere(age); }
```

interprets each module `A(age,length)` in the string to be a sphere of radius `age`.

4.3.3 Rule blocks

Generally, an L-system is written with an axiom, a block of productions, then decomposition rules, followed by interpretation rules.

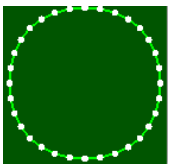
```
axiom: module list;
 predecessor : { successor }
 predecessor : { successor }
...
decomposition:
...
interpretation:
...
```

However, another possible organization, is to create a block of rules that apply to one type of module. For this, a `production:` statement is needed to return to regular productions after the first block. For example:

```
A() : { ... B() ... }
decomposition:
B() : { ... C() ... }
interpretation:
C() : { ... }

production:
X() : { ... Y() ... }
decomposition:
Y() : { ... Z() ... }
interpretation:
Z() : { ... }
```

4.4 Production groups



See object: B-spline

It is possible to specify alternate groups of productions and switch between them from one derivation step to the next. By default, all productions, decompositions, and interpretation rules belong to the default group, numbered 0. The default group has a special property: if no production in the current group can be applied to a symbol, the productions in the default group will be tried, even if it is not the current group.

To specify an additional group, use the statements:

```
group number:
...
endgroup
```

where *number* is an integer constant (not an expression or enumerated value) with a value greater than zero. The `endgroup` statement is not always required: a group also ends with another `group` statement, or with a `decomposition:` or `interpretation:` statement.

When *lpfg* is started, the default production group is used for the first derivation. To change to another group use the function:

```
UseGroup(grpid);
```

where *grpid* evaluates to an integer. It can be called at any time, but only takes effect on the next derivation step. It is often called at the beginning of each derivation step, in the `Start Each:` statement. For example, productions can alternate between two groups using the following statements:

```
Start: {n=0;}
StartEach: { UseGroup(((n++ % 2) == 0) ? 1 : 2) }
group 1:
...
group 2:
...
interpretation:
group 0:
...
```

In this case, the value of the `UseGroup` parameter is defined by a conditional statement: if the remainder when `n++` is divided by 2 is zero, then the group is 1, otherwise it is 2. The productions in the appropriate group will be apply in the next derivation step. Note that the `interpretation:` block returns to `group 0`; therefore, the productions in this block will always be used in the interpretation step.

There are also two specialized groups that are explained in greater detail later: Gillespie groups, `ggroup` (Section 7.2), and view groups, `vgroup` (Section 7.3).

5 Predefined modules

The following modules are predefined in the *lpfg* include files. The same names cannot be used for user-defined modules or global variables of any type. (The modules `f` and `g` cause name collisions particularly frequently.)

5.1 Branching structures

Module	Description	Equiv. in *cpfg*
`SB()`	Start new branch by pushing the current state onto the turtle stack.	`[`
`EB()`	End branch by popping the state from the turtle stack.	`]`
`Cut()`	Cut the remainder of the current branch, if the derivation direction is `Forward` (left to right). This module and all following modules are ignored up to the closest unmatched `EB` module, or the end of the string if no `EB` module is found. This module has no effect if the derivation direction is `Backward`.	`%`

5.2 Changing position and drawing

5.2.1 Turtle commands

Module	Description	Equiv. in *cpfg*
`F(float d)` `G(float d)`	Move forward a step of length `d` and draw a line segment from the original position to the new position. For `F` only: If the polygon flag is on (see Section 5.5), the final position is recorded as a vertex of the current polygon.	`F(d)` `G(d)`
`f(float d)` `g(float d)`	Move forward a step of length `d`. No line is drawn. For `f` only: If the polygon flag is on (see Section 5.5), the final position is recorded as a vertex of the current polygon.	`f(d)` `g(d)`
`MoveTo(float x, float y, float z)`	Move the turtle to point `(x,y,z)`	`@M(x,y,z)`

`MoveTo3f(V3f p)`
`MoveTo3d(V3d p)`
`MoveTo2f(V2f p)`
`MoveTo2d(V2d p)`
	Move the turtle to point `p`.	`@M`

`MoveRel3f(V3f p)`
`MoveRel3d(V3d p)`
`MoveRel2f(V2f p)`
`MoveRel2d(V2d p)`
	Move the turtle to the turtle's current position + `p`. The heading, left and up vectors are not changed.	

See Section 6.2.1 for a description of the predefined vector data types. For `V2d` and `V2f`: the z coordinate is assumed to be zero.

5.2.2 Affine geometry

Module	Description
`LineTo(float x, float y, float z)`	Draw a line from the turtle's current position to point `(x,y,z)`.
`LineTo3f(V3f p)` `LineTo3d(V3d p)` `LineTo2f(V2f p)` `LineTo2d(V2d p)`	Draw a line from the turtle's current position to point `p`. The turtle will be positioned at point `p`.
`LineRel3f(V3f p)` `LineRel3d(V3d p)` `LineRel2f(V2f p)` `LineRel2d(V2d p)`	Draw a line from the turtle's current position to its current position `+ p`. The turtle will be positioned at point `p`.
`Line3f(V3f p1, V3f p2)` `Line3d(V3d p1, V3d p2)` `Line2f(V2f p1, V2f p2)` `Line2d(V2d p1, V2d p2)`	Draw a line from point `p1` to point `p2`. The turtle will be positioned at point `p2`.
`SetCoordinateSystem (float s)`	Set the coordinate system affecting the above modules, using the turtle's current position and orientation and the global scaling factor `s`. The modules will be applied with respect to the modified coordinate system.

The turtle's heading, left and up vectors are not changed by these modules. If the distance between the two points is less than ϵ (a constant $= 10^{-5}$), these modules are ignored.

See Section 6.2.1 for a description of the predefined vector data types. For `V2d` and `V2f`: the z coordinate is assumed to be zero.

There are no *cpfg* equivalents for these modules.

5.3 Other turtle modules

5.3.1 Rotations

Module	Description	Equiv. in *cpfg*
`Left(float a)`	Turn left around the **U** axis by angle `a`	`+(a)`
`Right(float a)`	Turn right around the **U** axis by angle `a`	`-(a)`
`Up(float a)`	Pitch up around the **L** axis by angle `a`	`^(a)`
`Down(float a)`	Pitch down around the **L** axis by angle `a`	`&(a)`
`RollL(float a)`	Roll left around the **H** axis by angle `a`	`\(a)`
`RollR(float a)`	Roll right around the **H** axis by angle `a`	`/(a)`
`RollToVert()`	Roll around the **H** axis so that **H** and **U** lie on a common vertical plane, with **U** closer to up than down.	`@v`
`RotateXYZ (V3f axis, float angle)`	Rotate by `angle` around `axis` in global XYZ coordinates. The axis will be normalized. If its length is less than ϵ, no rotation will occur.	
`RotateHLU (V3f axis, float angle)`	Rotate by `angle` around `axis` in local turtle (HLU) coordinates. The axis will be normalized. If its length is less than ϵ, no rotation will occur.	

Module	Description	Equiv. in *cpfg*
`SetHead (float hx, float hy, float hz, float ux, float uy, float uz)`	Set the heading vector of the turtle to `hx,hy,hz`, the up vector to `ux,uy,uz`, and the left vector to the cross product of the new **H** and **U**. Normalized vectors do not need to be specified. The module is ignored if any of the three settings is less than ϵ.	`@R(hx, hy,hz, ux,uy, uz)`
`SetHead3f(V3f h)`	Set the heading vector of the turtle to vector `h`. The turtle frame is rotated by the smallest rotation necessary to align the old and new heading vectors (i.e. parallel transport transformation).	

NOTE: There was a bug in the previous implementation of `Up, Down, RollL,` and `RollR` which caused the turtle to rotate in the opposite direction. This has been fixed; however, in order to keep compatibility with existing models, the view file parameter `corrected rotation` can be used to turn off the corrected behaviour (see Section 8.2.6).

See Section 6.2.1 for a description of the predefined vector data type, `V3f`.

5.3.2 Display parameters

Module	Description	Equiv. in *cpfg*
`IncColor()`	Increase the current colour index or material index by one.	`;`
`DecColor()`	Decrease the current colour index or material index by one.	`,`
`SetColor(int n)`	Set the current colour index or material index to `n`. If `n` <1 or $>$ 255, the module is ignored.	`;(n)` `,(n)`
`SetWidth(float v)`	Set the line width to `v`. If `v` $\leq$ `0`, the module is ignored.	`#(n)` `!(n)`

5.3.3 Turtle queries

If any of the following query modules are present in the predecessor of any production in the L-system, an interpretation step is performed after each derivation step even if no drawing occurs. The turtle is "moved" and all positions are calculated in case they are needed by the query modules.

Module	Description	Equiv. in *cpfg*
`GetPos(float x, float y, float z)`	Query the `x, y,` and `z` coordinates of the current turtle position.	`?P(x,y,z)`
`GetHead(float x, float y, float z)`	Query the `x, y,` and `z` coordinates of the current turtle heading vector.	`?H(x,y,z)`
`GetLeft(float x, float y, float z)`	Query the `x, y,` and `z` coordinates of the current turtle left vector.	`?L(x,y,z)`
`GetUp(float x, float y, float z)`	Query the `x, y,` and `z` coordinates of the current turtle up vector.	`?U(x,y,z)`

If there are multiple views (Section 7.3), the interpretation rules in `vgroup 0` will be used.

5.3.4 View and labels

Module	Description	Equiv. in *cpfg*
`Camera()`	Change the view parameters such that the camera is located at the position of the turtle, with the same orientation. See also the `new view between frames:` parameter in the animation file (Section 8.1).	
`Label(Text str)`	Print the string `str` at the current turtle position. `Text` is a predefined data type: `typedef const char __lc_Text`	`@L(str)`

5.4 Circles and spheres

Module	Description	Equiv. in *cpfg*
`Circle()`	Draw a circle, with diameter equal to the current line width, in the HL plane.	`@o`
`CircleFront0()`	Draw a circle, with diameter equal to the current line width, in the screen plane.	
`Circle(float r)`	Draw a circle of radius `r` in the HL plane, centred at the current turtle position.	`@o(d)` where `d` is the diameter, not the radius.
`CircleFront (float r)`	Draw a circle, with radius `r`, in the screen plane.	
`CircleB(float r)`	Draw a circle outline in the HL plane, with inner radius = `r - width/2` and outer radius = `r + width/2`, where `width` is the current line width.	`@bc(r)`
`CircleFrontB (float r)`	Draw a circle outline in the screen plane, with inner radius = `r - width/2` and outer radius = `r + width/2`, where `width` is the current line width.	`@bo(r)`
`Sphere0()`	Draw a sphere, with diameter equal to the current line width.	`@O`
`Sphere(float r)`	Draw a sphere of radius `r` at the current turtle position.	`@O(d)` where `d` is the diameter, not the radius.

The number of sides in the circle approximation is controlled by the `ContourSides` module (Section 5.7), or the `contour sides` command in the view file (Section 8.2.2). For spheres, there will be `contour sides` longitudinal sections and `(contour sides+1)/2` transversal sections.

5.5 Polygons, rhombus, and isosceles triangles

Module	Description	Equiv. in *cpfg*
`SP()`	Start a polygon.	{

Module	Description	Equiv. in *cpfg*
`EP()`	End a polygon.	}
`PP()`	Set a polygon vertex.	.
`Rhombus(float length, float width)`	Draw a rhombus in the HL plane. The turtle is at the center of the bottom edge.	
`Triangle(float width, float height)`	Draw an isosceles triangle in the HL plane. The turtle is at the center of the bottom edge.	

5.6 Surfaces and Meshes

Predefined surfaces and meshes are specified in the view file (Section 8.2.3), where the first surface in the file has `id`=0.

Module	Description	Equiv. in *cpfg*
`Surface(int id, float scale)`	Draw the predefined Bézier surface `id` at the current location and orientation. The surface will be uniformly scaled by the factor `scale`.	~
`Surface3(int id, float xscale, float yscale, float zscale)`	Draw the predefined Bézier surface `id` at the current location and orientation. The surface will be scaled independently along the *x*, *y* and *z* axes by `xscale, yscale,` and `zscale`, respectively.	
`Mesh(int id, float scale)`	Draw the predefined mesh at the current location and orientation. The mesh will be uniformly scaled by the factor `scale`.	
`Mesh3(int id, float xscale, float yscale, float zscale)`	Draw the predefined mesh at the current location and orientation. The mesh will be scaled independently along the *x*, *y* and *z* axes by `xscale, yscale,` and `zscale`, respectively.	
`SetUPrecision (float p)`	Set the drawing precision of bicubic surfaces to `p` in the U direction. If set to zero, the U precision is reset to the surface default, defined in the view file.	
`SetVPrecision (float p)`	Set the drawing precision of bicubic surfaces to `p` in the V direction. If set to zero, the V precision is reset to the surface default, defined in the view file.	
`InitSurface(int id)`	Initialize an L-system-define surface. Currently there is only one surface allowed, so the parameter is ignored.	`@PS`
`SurfacePoint (int id, int p, int q)`	Set the `(p,q)` control point of the L-system-defined surface to the current turtle position. The `id` parameter is ignored.	`@PC`
`DrawSurface(int id)`	Draw the L-system-defined surface. The `id` parameter is ignored.	`@PD`
`DSurface (SurfaceObj s)`	Draw the dynamic Bézier surface `s`. See Section 7.1.	

5.7 Generalized cylinders

Generalized cylinders are specified as contours, which can be defined using the *cuspy* tool (see the *Vlab Tools* manual), and listed on the command line (Section 1.1.2). In addition, the cylinder can

be texture mapped using an image file specified in the view file (Section 8.2.3). Both contours and textures are referenced sequentially by an `id` in the order in which they were listed.

Module	Description	Equiv. in *cpfg*
`StartGC()`	Start a generalized cylinder at the current turtle position.	`@Gs`
`PointGC()`	Specify a control point on the central line of the generalized cylinder.	Similar to `@Gc(n)`
`EndGC()`	End the current generalized cylinder.	`@Ge`
`CurrentContour (int id)`	Set contour `id` as the current contour for generalized cylinders. If `id`=0, the default contour (a circle) is used.	`@#(id)`
`BlendedContour (int id1, ind id2, float blend)`	Interpolate the contour between `id1` and `id2` using the interpolating coefficient `blend`. At `blend`=0 the contour is `id1`; at `blend`=1 the contour is `id2`.	
`ScaleContour (float p, float q)`	Scale the contour independently by `p` (left) and `q` (up).	
`ContourSides (int sides)`	Specify the number of sides all subsequent generalized cylinders will have. This module should be placed before the `StartGC` module; it has no effect within a generalized cylinder (i.e. between `StartGC` and `EndGC`).	
`CurrentTexture (int texid)`	Use texture `txtid` to texture map the generalized cylinders. If `txtid=-1`, texture mapping is turned off.	
`TextureVCoeff (float v)`	Set the texture's scaling factor, where `v` is the portion of texture that will be mapped to the cylinder as the turtle moves forward one unit. For example, to map the texture to a cylinder that is 10 units long, set `v` to 0.1. If `v` > 1, the texture wraps.	

5.8 Tropisms

Tropisms are defined in the view file (Section 8.2.4). They are numbered sequential with an `id` as they appear in the file.

Module	Description	Equiv. in *cpfg*
`SetElasticity (int id, float v)`	Set the elasticity parameter of tropism `id` to `v`. This is equivalent to the `S:` parameter of the `tropism` and `torque` commands in the view file.	`@Ts`
`IncElasticity (int id)`	Increment the elasticity parameter of tropism `id` by the value defined by `SetElasticity`.	`@Ti`
`DecElasticity (int id)`	Decrement the elasticity parameter of tropism `id` by the value defined by `SetElasticity`.	`@Td`
`Elasticity (float v)`	Set the elasticity to `v`.	_ (underscore)

5.9 Mouse interaction modules

The following two modules are used to interactively identify a component of the model using the mouse and a combination of key strokes. The module is inserted into the string before the object identified

by the mouse. If no object is identified (i.e. the mouse is clicked outside of the model components), no module is inserted.

Module	Description
`MouseIns()`	Inserted into the string when the user holds down the Shift and Command keys (or the `1` key) and clicks the left mouse button on a component of the model.
`MouseInsPos (MouseStatus)`	Inserted into the string when the user holds down the Alt and Command keys (or the `2` key) and clicks the left mouse button on a component of the model. A `MouseStatus` structure is included with the insertion of this module.

See Section 7.4 for more details, including the definition of the `MouseStatus` data type.

5.10 Environment modules

Module	Description	Equiv. in *cpfg*

`E1(float v)`
 	
`E2 (float v1, float v2)`
 	
`EA20(EA20Array a)`
	Send or receive environmental information, using the individual parameters, `v`, or `v1` and `v2`, or the array `a`.	`?E(v)`

See the *Environment Programs* manual for more details including the definition of `EA20Array`.

6 Predefined functions

6.1 Controlling the L-system derivation

6.1.1 Forward and Backward functions

Function	Description
`void Forward()`	Perform the next derivation step from left to right. This is the default.
`void Backward()`	Perform the next derivation step from right to left.
`bool IsForward()`	Returns the last derivation direction. Note that this function returns the value of the last `Forward` or `Backward` statement but may not reflect the current derivation direction if it is changed *during* a derivation step.

See the section on new context (Section 3.1.3) for use of these functions.

6.1.2 Group functions

Function	Description
`void UseGroup(int)`	Use the group from the specified `group` or `ggroup` in the next derivation step.
`int CurrentGroup()`	Return the number of the current group.

See the sections on groups (Section 4.4) and Gillespie groups (Section 7.2) for use of these functions.

6.2 Vectors

6.2.1 Predefined vector structures

Vector functions are used with a set of pre-defined structures.

```
struct V2f { float x,y; };
struct V2d { double x,y; };
struct V3f { float x,y,z; };
struct V3d { double x,y,z; };
```

If the preprocessor symbol `NOAUTOOVERLOAD` is not defined before `#include lpfgall.h`, these structures receive additional functionality including operators for addition and subtraction of two structures of the same type, unary negation, multiplication and division of a vector by a scalar, dot product, and the assignment operators `+=, -=, *=`, and `/=`. In addition, the cross product is defined on `V3f` and `V3d` with operator `%`. Some examples are:

```
V2f a(1.5, 2,0), b(0, 0.5);
V2f c = a * 2.5 + b;
float x = a * b;

v3f d(1.2, 2.3, 0), e(0, 0,5, 0,1);
V3f f = d % e;
```

6.2.2 Vector methods

Most functionality associated with vectors are actually methods:

Method	Description
`Length()`	Return the vector's length as float or double, depending on the structure.
`Normalize()`	Normalize the vector.
`Normalized()`	Return a normalized form of the vector.
`Set(x,y)` `Set(x,y,z)`	Set the components of a vector.

Examples of these methods are:

```
float x = a.Length();

a.Normalize();              //  Vector a is normalized
b = a.Normalize();          //  Both vectors a and b are normalized
b = a.Normalized();         //  Vector b is normalized only

V2f a;
a.Set(7,5);
```

6.2.3 Vector functions

There is only one type of vector function:

Function	Description

`V2d normalize(V2d v)`
`V2f normalize(V2f v)`
`V3d normalize(V3d v)`
`V3f normalize(V3f v)`
	Normalize vector `v`, and return a copy of this vector.

6.3 Curve and surface functions

Curves are predefined B-spline contours, which can be defined using the *cuspy* tool (see the *Vlab Tools* manual), and are listed on the command line (Section 1.1.2). Predefined surfaces can be defined using either the *bezieredit* or *stedit* tool and are specified in the view file (Section 8.2.3). Both contours and surfaces are referenced sequentially by an `id` in the order in which they were listed.

Function	Description

`float curveX(int id, float t)`
`float curvey(int id, float t)`
`float curveZ(int id, float t)`
`V2f curveXY(int id, float t)`
`V3f curveXYZ(int id, float t)`
	Return the coordinates of of curve `id` defined in the contour-set file, where `t` is the arc-length parameter.

`void curveScale (int id,` `float x, float y, float z)`	Scales curve `id` by the factors `x, y,` and `z`.
`void curveSetPoint(int id, int p,` `float x, float y, float z)`	Assign control point `p` in curve `id` to position `(x,y,z)`. The curve must be recalculated using `curveRecalculate` in order for the `curve` functions to return proper values.

Function	Description
`void curveRecalculate(int id)`	Recalculate curve `id` after assigning a control point with `curveSetPoint`.
`void curveReset(int id)`	Reset curve `id` to the state define in the contour-set file. The file is not re-read.
`SurfaceObj GetSurface(int id)`	Return the control points of the predefined Bézier surface specified in the view file as `id`. If the surface contains more than one patch, only the first patch is returned. Used to dynamically manipulate a surface (see Section 7.1).

See Section 5.7 for the use of contours to create generalized cylinders, and Section 5.6 for modules related to surfaces.

6.4 View functions

Function	Description
`void UseView(int id)`	Activate view number `id` from the view file.

`Float vvXmin(int id)`
`Float vvYmin(int id)`
`Float vvZmin(int id)`
`Float vvXmax(int id)`
`Float vvYmax(int id)`
`Float vvZmax(int id)`
	Return the coordinate of the bounding box of view number `id`.

`float vvScale(int id)`	Return the current projection scaling factor of view number `id`.
`CameraPosition GetCameraPosition(0)`	Get the current position of the camera.

See Section 7.3 for a description of all the components of multi-view mode including `UseView`. `CameraPosition` is a predefined data type:

```
struct CameraPosition {
  V3tf position, lookat;
  V3tf head, left, up;
  float scale;
};
```

6.5 Animation functions

The following functions are available in Animate mode only; they are ignored outside of this mode.

Function	Description
`void DisplayFrame()`	Display a frame of the animation at the current derivation step, if the `display on request` parameter is set to `on` in the animation file (Section 8.1). If it is `off`, this function has no effect.

Function	Description
`void OutputFrame("`*filename.ext*`")`	Output a frame of the animation at the end of the current derivation step, as an image, postscript, or OBJ file, depending on *ext*. If the `display on request` parameter is set to `on` in the animation file (Section 8.1), this call must be preceded by a `DisplayFrame()` function so that the frame buffer is updated.
`void RunSimulation()`	Run the simulation. Only executed when a derivation step is performed; thus, at least a single step may be required if the simulation is already paused.
`void PauseSimulation()`	Pause the simulation.
`void Stop()`	Stop the simulation. The `End` statement is executed after the current derivation step.

6.6 Calling an external function

External functions are defined in three types of input files: *function*`.func`, *functionset*`.fset`, and *timeline*`.tset` (see Section 1.1.2). The functions are numbered in the order they are read from the files, beginning with `1`. There is also an all-caps version of the `name` parameter defined.

The functions are called within the L-system using one of the following forms. If a function number (`id`) is outside the number of functions, or the function name (`fname`) is not found, the value 0 is returned.

Function	Description
`float func(int id, float x)` `float func(char* fname, float x)`	Return the value of a function defined in a function file, specified by its order number (`id`) or its name (`fname`). The parameter `x` must be in the range [0,1].

`float pfunc(int id,`
`float x, float min, float max)`
 	
`float pfunc(char* fname,`
`float x, float min, float max)`
	Return the value of a function defined in a function file, specified by its order number (`id`), or its name (`fname`). The function is evaluated over the range [`min`,`max`]. The parameter `x` must be within that range.

`float tfunc(int id, float x)` `float tfunc(char* fname, float x)`	Return the value of a function defined in a timeline (`.tset`) file specified by its order number (`id`), or its name (`fname`). The function is evaluated over the range specified in the timeline file. The parameter `x` must be within the specified range.

6.7 External parameters

Parameters can be defined in a `.vset` file (Section 1.1.2) to explore the parameter space of a model without editing and re-reading the L-system file, which requires that the L+C code be re-compiled before generating the new image. The following function is used to retrieve the value of a parameter from the file:

Function	Description
`float val(char* pname)`	Return the value of parameter `pname` from the *parameter*`.vset` file.

The *parameter*`.vset` file contains `#define` statements, one per line, in the format:

`#define` *pname value*

where *pname* is the parameter name, and *value* is its initial value.

For example, if *parameter*.`vset` contains:

```
#define LENGTH 10
#define ANGLE 60
```

the L-system retrieve the parameters with the statements:

```
len = val(LENGTH);
a = value(ANGLE);
```

Note that the return variable cannot have the same name as the parameter. The variable must be declared, but the parameter is not.

To see the results of parameter changes immediately, ensure the refresh mode is set to Continuous. This can be done on the command line (`-rmode cont`) or from the menu (Refresh mode > Triggered/Continuous).

6.8 Mouse and menu functions

Function	Description
`struct MouseStatus GetMouseStatus()`	Return the state of the mouse. See Section 7.4 for examples of its usage.
`void UserMenuItem (char* label, int code)`	Add the menu item `label` to the user menu, and return `code` when it is selected. The menu is accessed by holding down two of the CMD, SHIFT, and ALT keys and clicking the right mouse button.
`int UserMenuChoice()`	Return the `code` associated with the last selection made from the user menu since the previous call to this function.
`void UserMenuClear()`	Clear the user menu.

6.9 Input and output functions

Function	Description
`void Printf(const char*, ...)`	Print message to the `lpfg.log` file, and to the console if it is open. Recommended over the standard C function *printf* since *lpfg* may not be connected to a console.
`void OutputString (const char* filename)`	Write the current string to the specified file in binary format (`.strb`), similar to the String > Output menu item.
`void LoadString (const char* filename)`	Overwrite the current string with the string in the specified binary file (`.strb`), similar to the String > Input menu item. Normally this is a string created by the `OutputString` function, or the String > Output menu item. This function should be called in a control block, not within a production.

6.10 Random number functions

Function	Description
`float ran(float range)`	Generate a peseudo-random number uniformly distributed in the range (0, range).

Function	Description
`void sran(long seed)`	Seed the pseudo-random number generator used by `ran`. Use `sran` in the `Start` block to ensure every run is identical, even after rewinding.
`void SeedGillespie(long seed)`	Seed the pseudo-random number generator used by the Gillespie engine (see Section 7.2).

6.11 Environmental functions

Function	Description
`void Environment()`	Perform environment interpretation *after* the `EndEach` block. Environment information will be available in the *next* derivation step.
`void NoEnvironment()`	Turns environment interpretation off unconditionally.

See the *Environment Programs* manual for more information.

7 Advanced topics

7.1 Dynamic surfaces

Single-patch Bézier surfaces that can be dynamically created and/or manipulated from within the L-system. This are useful, for example, when creating an animation with the use of "keyframe" surfaces, or when building a family of similar surfaces that are modifications of a predefined set of base surfaces.

The manipulations that can be performed on a surface include:

- Non-uniform scaling
- Linear interpolation between surfaces
- Manipulation of individual the control points

7.1.1 Creating dynamic surfaces

A dynamic surface can be initialized for further manipulation by:

- Using the `GetSurface` function (Section 6.3) to get the control point coordinates of a predefined surface specified in the view file (Section 8.2)
- Initializing the coordinates of individual control points within the L-system

To explicitly initialize the coordinates of a control point use one of the Set methods:

```
void SurfaceObj::Set(int id, const float* arr)
void SurfaceObj::Set(int id, const V3f& v)
```

See Section 6.2.1 for a description of the predefined vector data type, `V3f`.

A similar method is available to get the coordinates of a control point:

```
V3f SurfaceObj::Get(int id) const
```

7.1.2 Manipulating dynamic surfaces

Scalar multiplication operators can be used to scale a surface object by a real number:

```
const SurfaceObj SurfaceObj::operator*(float r)
friend SurfaceObj operator*(float r, const SurfaceObj& obj)
```

To scale the surface non-uniformly (by a different factor in each direction), make the scaling factors coordinates of a `V3f` vector and use the method:

```
void SurfaceObj::Scale(V3f scale)
```

The addition operator combines two surfaces by pointwise adding their control points:

```
friend SurfaceObj operator+(const SurfaceObj& l,const SurfaceObj& r)
```

The addition operator, along with the scalar multiplication operator, defines a vector space over patches. This can be used to interpolate between surfaces. For example:

```
SurfaceObj s1, s2;
float weight;
...
SurfaceObj interpolated = s1*weight + s2*(1-weight);
```

7.1.3 Drawing dynamic surfaces

To draw a dynamic surface, use the predefined `DSurface` module (Section 5.6). For example, a surface can be initialized and drawn with:

```
SurfaceObj leaf_surface = GetSurface(LEAF);
...
produce DSurface(leaf_surface);
```

7.2 Gillespie groups

Gillespie groups are a special case of production groups (Section 4.4), with a different derivation strategy. They are designed for modeling chemical reactions as stochastic processes. The specification of a Gillespie group begins with

`ggroup` *number*:

where *number* is an integer and part of a shared numbering system with regular production groups; therefore, a regular group and a Gillespie group cannot have the same number. Gillespie groups end with the standard `endgroup` statement, and are called using the standard `UseGroup` function.

Unlike a regular derivation step where every module in the string can produce a successor, a derivation step using a Gillespie group will have only *one* module in the entire string produce a successor, chosen using *Gillespie's method* [1]. All other modules will remain the same.

Each module defined in a Gillespie group specifies the reactions that may occur within the module and the likelihood (or propensity) of each reaction. For example, if the `Cell` module specifies the Michaelis-Menten reactions:

$$S + E \underset{c_1}{\rightarrow} ES, \quad ES \underset{c_2}{\rightarrow} S + E, \quad ES \underset{c_3}{\rightarrow} P + E$$

then the production for the `Cell` module in the Gillespie group would be:

```
Cell(S,E,ES,P):
{
   propensity c1*S*E produce Cell(S-1,E-1,ES+1, P);
   propensity c2*ES produce Cell(S+1,E+1,ES-1,P);
   propensity c3*ES produce Cell(S,E+1,ES-1,P+1);
}
```

In each derivation step, *lpfg* will randomly choose the next reaction to take place based on the propensities of *all* the modules in the Gillespie group such that the reaction with the greatest propensity is more likely to be chosen. For example, if there are ten `Cell` modules with the three reactions above, *lpfg* will pick one reaction out of 30. It will also calculate the time τ to the next reaction as $\tau = ln(1-\chi)/p$, where χ is a uniform random number in (0,1) and p is the sum of the propensities of all modules. To access τ, call the function:

```
float GillespieTime();
```

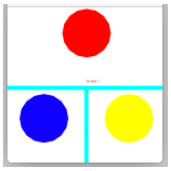
There are two restrictions when using Gillespie groups:

- Ring L-systems are ignored.
- New context is not supported.

7.3 Multi-view mode

lpfg allows multiple views to be displayed simultaneously. The location of each view within the main window is defined in the view file (Section 8.2.1) using the `window` command. For example, to create two views that use the left and right halves of the *lpfg* window, the commands would be:

```
window: leftview  0.0  0.0  0.5  1.0
window: rightview 0.5  0.0  1.0  1.0
```



See object: Multiview

The default border between the views is a black line, one pixel wide. This can be altered with the `window border` command. Note that the view area is cut on the side with a border. This is especially noticeable if a wide border is used.

To activate these views within the L-system, they must be defined with the `UseView` function (Section 6.4). The function is normally called within the `Start:` statement. For example:

```
Start: {
     UseView(leftview);
     UseView(rightview);
}
```

The actual content of each view is defined in the `interpretation:` section of the L-system using the `vgroup` command. For example, for the two views defined above, there would be two `vgroup` subsections within the `interpretation` section:

```
interpretation:
vgroup leftview:
     ...
     produce ... ;
     ...
vgroup rightview:
     ...
     produce ... ;
     ...
```

7.4 Mouse interactions

The status of the mouse can be obtained using the `GetMouseStatus()` function, which returns a `MouseStatus` structure defined as:

```
struct MouseStatus {
  int viewNum;          // currently active view
  int viewX,viewY;      // x,y pixel positions of mouse cursor

  V3d atFront,atRear,atMiddle;
                  // Intersection of cursor ray with viewplane
                  // Front or back viewplane, or halfway between them
```

```
                    // Independent of any keys

    bool lbDown;          // Left button currently down

    bool lbPushed, lbReleased;
                    // Left button pressed/released
                    // since last call to GetMouseStatus
};
```

The left button values, `lbDown` and `lbPushed`, are only set when the left button is pushed with a combination of keys. These key combinations are also used to determine which mouse module is inserted when the left mouse button identifies a component of the model:

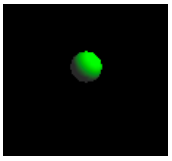
Key combination	Alternate key	Module inserted
Shift+Command Shift+Alt+Command	1	`MouseIns()`
Alt+Command	2	`MouseInsPos(MouseStatus)`
Shift+Alt	3	*No module inserted*

Therefore, using any combination of the keys above, and the left mouse button, it is possible to draw a line:

```
MouseStatus ms;
...
ms = GetMouseStatus();
if(ms.lbPushed) // start a line
     produce MoveTo3d(ms.atMiddle) Cursor();
if (ms.lbDown)  // continue drawing while button is down
     produce LineTo3d(ms.atMiddle) Cursor();
```

See object:
DrawLine

The following code draws a sphere when the left mouse button is pushed along with one of the key combinations for `MouseIns()`. The sphere can then be selected and moved.



See object:
MoveSphere

```
module AddSphere();
module PosSphere(V3d, int);
...
MouseStatus ms;
...
StartEach: { ms = GetMouseStatus(); }

production:

// Draw sphere when button first pushed
AddSphere():
{
     if(ms.lbPushed) { produce PosSphere(ms.atMiddle, 1); }
}

// Existing sphere selected
// (MouseIns() module has been inserted into the string)
MouseIns() PosSphere(pos, selected) :
```

```
{
     produce PosSphere(ms.atMiddle, 1) ;
}

// Move the sphere as long as the mouse button is not released
PosSphere(pos, selected) :
{
     if (selected && !ms.lbReleased)
          produce PosSphere(ms.atMiddle, 1);
     if (selected)
          produce PosSphere(ms.atMiddle, 0) ;
}
...

interpretation:
PosSphere(pos, selected) :
     { produce MoveTo3d(pos) SB Sphere(1) EB ; }
```

8 *Lpfg*-specific input files

8.1 Animation file

The animation file is identified by its extension (*filename*`.a`), and may contain the following commands:

Command	Description	Default
`first frame:` *n*	Interpret derivation step *n* as the first frame of an animation.	`0`
`last frame:` *n*	Interpret derivation step *n* as the last frame of an animation.	`Derivation length`
`step:` *n*	Set the number of derivation steps between frames to *n*.	`1`
`swap interval:` *t*	Set the time interval between frames to *t*.	
`double buffer:` *flag*	Set double buffer flag `on` or `off`.	`on`
`clear between frames:` *flag*	Clear between frames (*flag* = `on`).	`on`
`hcenter between frames:` *flag*	Horizontally center the model between frames (*flag* = `on`).	`off`
`scale between frames:` *flag*	Scale the model to fit the view window between frames (*flag* = `on`).	`off`
`new view between frames:` *flag*	Reset the view between frames (*flag* = `on`). This command is most useful when using the `Camera()` module (Section 5.3.4) to dynamically position the camera.	`off`
`display on request:` *flag*	Display frames only on request. When *flag* = `on`, only the first and last fame are displayed automatically. The `DisplayFrame()` function (Section 6.5) must be called to display intermediate frames. This makes it possible to skip frames that do not advance time but perform other calculations. If *flag* = `off`, frames are displayed according to the `step` parameter.	`off`

Note that in *lpfg* the Rewind command on the pop-up menu returns to the axiom (whereas in *cpfg* it returns to the first derivation step), and the first frame defaults to 1, not 0).

8.2 View file

Viewing and drawing parameters are stored in the view file, identified by its extension (*filename*`.v`)[2].

The view file is read by the C++ preprocessor; therefore, the use of comments (both C style `/* ... */` and C++ style `//`), as well all other standard preprocessor directives such as `#define` and `#if` statements are allowed.

The commands in the file are interpreted in the order in which they appear in the file. If there are two or more commands that specify the same parameter, the last one takes precedence. This does not apply to commands that specify new set of parameters every time they appear (e.g. `lights, tropisms`). Every command must be contained on a single line.

[2]Some older models may use *filename*`.dr`

8.2.1 Setting the view

Command	Description	Default
`projection:` *pvalue*	Set the projection to `parallel` or `perspective`.	`parallel`
`scale:` *s* `scale factor:` *s*	Set the size of the final image on the screen. For full size, set $s = 1.0$. The two commands are equivalent.	`0.9`
`min zoom:` *zmin*	Set the minimum value of the zooming factor.	`0.05`
`max zoom:` *zmax*	Set the maximum value of the zooming factor.	`50`
`line style:` *lstyle*	Set the line style to `pixel, polygon` or `cylinder`.	`pixel`
`front distance:` *d1* `back distance:` *d2*	Set the distance to the front (*d1*) and back (*d2*) clipping planes, from the viewer in `perspective` projection, or from the position of the clipping plane with respect to the centre of the object's bounding box in `parallel` projection. Thus in `parallel` projection the `front distance` should be a negative number and the `back distance` should be positive. Both commands must be specified in order to have an effect.	No clipping plane
`generate on view change:` *vchange*	Regenerate the L-system string (the simulator rewinds to the axiom and performs the derivations again) to: - `on` - every time the view changes through rotation, zoom, or pan - `triggered` - after the user releases the mouse button - `off` - never	`off`
`view:` *id* `dir:` *dx dy dz* `up:` *ux uy uz* `pan:` *px py pz* `fov:` *val* `shift:` *val* `scale:` *val*	Define the view transformations to be used for view window *id*. All the transformation commands are optional. - `dir` and `up`: the view direction and up direction. - `pan`: the point that is the center of the view, relative to the center of the bounding box. - `fov`: the angle of the field of view in the y direction. - `shift`: the distance between the camera and the point being looked at. - `scale`: the scale of objects.	
`box:` *id xmin xmax ymin ymax zmin zmax*	Define the default bounding box for view window *id*.	
`window:` *vname left top width height*	Define the location of view *vname* within the *lpfg* window. The parameters *left, top, right* and *bottom* are the relative position of the view within the main window where 0,0 is the upper left corner and 1,1 is the bottom right. See Section 7.3 for a description of all components of multi-view.	
`window border:` *size r g b*	Define the size and color of the border between multiple views, where *size* is in pixels, and *r, g, b* are integers between 0 and 255. See Section 7.3 for a description of all components of multi-view.	*size* = `1` *r*=*g*=*b*=`0`

8.2.2 Rendering commands

Command	Description	Default
`z buffer:` *zflag*	Turn z buffering `on` and `off`.	`off`
`render mode:` *rvalue*	Set the rendering mode to `filled`, `wireframe`, `shaded`, or `shadows`.	`filled`
`light:` `O:` *x y z* `V:` *x y z* `P:` *x y z e c* `A:` *r g b* `D:` *r g b* `S:` *r g b* `T:` *c l q*	Define a light source as one of: - `O`: origin of point light source - `V`: vector of directional source - `P`: spotlight with direction *(x,y,z)*, exponent *e*, and cutoff angle *c* And, optionally, the characteristics of the source: - `A`: ambient color of light source - `D`: diffuse color of light source - `S`: specular color of light source - `T`: attenuation factors It is possible to define up to 8 light sources, one per line.	`V: 0 0 1` (corresponds to the default view direction)
`shadow map:` `size:` *n* `color:` *r g b* `offset:` *factor units*	Define parameters for shadow mapping when the `render mode` parameter is set to `shadows`. The shadow map will be generated using the first directional or spot light source specified with the `light` parameter. The following parameters are optional: - `size`: width and height of the shadow map (*n* x *n*), where *n* must be an even number. Values that are too small ($n < 100$) or too large (dependent on graphics card) may cause shadows not to displayed. - `color`: shadow color in *rgb* components. - `offset`: polygon offset for a generating depth map used to reduce shadow acne (erroneous self-shadowing). To reduce shadow acne, try increasing these values.	*n* = 1024 *r* = 0.2 *g* = 0.2 *b* = 0.4 *factor* = 5 *units* = 10
`stationary lights:` *flag*	Enable stationary light sources (*flag* = `on`) keeping the position of all light sources fixed.	`on`
`contour sides:` *n*	Set the number of sides *n* that will be drawn on generalized cylinders. Affects all generalized cylinders, but can be overridden by either the `ContourSides` module (Section 5.7), or the contour-specific `samples` parameter.	
`backface culling:` *flag*	Specify that backward-facing polygons should not be drawn (*flag* = `on`). This may speed up rendering or improve the rendering of transparent objects.	`off`
`concave polygons:` *flag*	Enable the OpenGL tesselator (*flag* = `on`), which divides polygons into triangles. This allows for more complex concave polygon shapes, but will cause *lpfg* to run slower.	

Command	Description	Default
`gradient:` *direction magnitude*	Define whether gradient shading should be applied to surfaces when exporting to postscript. *direction*=`0`: gradient off *direction*=`1`: gradient left to right *direction*=`2`: gradient bottom to top *magnitude*: percentage change from near to far edge (where 1.0 represents 100%). May be positive or negative.	

8.2.3 External files

Command	Description	Default
`surface:` *filename*`.s` *scale sdiv tdiv txid*	Declare the predefined Bezier surface in *filename*`.s`. The remaining parameters are optional: - *scale*: a file-specific scaling parameter which is multiplied by the scaling parameter in the `Surface` module to produce a total scaling factor. - *sdiv* and *tdiv*: the number of subdivisions to draw along the s and t axes. These parameters must be used together. - *txid*: the texture associated with the surface.	*scale* `= 1`
`mesh:` *filename* `S:`*scale* `T:`*txid* `C:`$x\ y\ z$	Declare a predefined mesh in *filename* with optional scaling (`S:`), texture (`T:`), and contact point coordinates (`C:`). The mesh file can be in OBJ or PLY format.	*scale* `= 1` $x\ y\ z$ `= 0 0 0`
`texture:` *filename*	Declare a texture in image file *filename*. Textures are assigned identifiers in the order given, starting at 0. Both the width and height of the image must be less than 4096. Only RGB files are supported.	

See Section 5.6 for surface and mesh modules.

8.2.4 Tropism commands

Command	Description	Default

`tropism:`
`T:` $x\ y\ z$
`A:` a
`I:` x
`E:` e
`S:` de
	Set tropism parameters. The tropism vector (`T`) is required. The remaining parameters are optional:
	- `A:` angle (in degrees) that segments are trying to reach, with respect to the tropism vector
	- `I:` (global) intensity of the tropism
	- `E:` initial elasticity
	- `S:` elasticity step
		`A: 0`
		`I: 1`
		`E: 0`
		`S: 0`

`torque:`
`T:` $x\ y\ z$
`I:` x
`E:` e
`S:` de
	Set parameters for rotating segments around their heading without modifying the heading orientation. The tropism vector (`T`) is required. The remaining parameters are optional, and are the same as for `tropism`, except that `A` is not required.	`I: 1`
		`E: 0`
		`S: 0`

`stropism:` $x\ y\ z,\ e$	Define a simple tropism, specifying the tropism vector *(x,y,z)* and the elasticity e.	

There may be multiple tropisms in the view file. Tropisms can be manipulated using the modules in Section 5.8.

8.2.5 Fonts

Command	Description	Default
`font:` *Xfont*	Define the font type to be used in @L interpretation, using the Xfont specification.	`-*-courier-bold-r-*-*-12-*-*-*-*-*-*-*`
`winfont:` *font size* `bi`	Define the font for the `Label` module. - *font*: the font name. Enclose in quotation marks if multiple words (e.g. `"Times New Roman"`) - *size*: the font size in pixels. - `bi`: optional flags to specify bold and/or italics respectively.	`Ariel 12`

8.2.6 Correction

Old versions of *lpfg* had a bug which caused all rotations by the modules `Up, Down, RollL,` and `RollR` to be in the wrong direction. This was fixed, but in order to run old models without corrections, this command is needed.

Command	Description	Default
`corrected rotation: on|off`	Use the corrected rotations (`on`). Turn `off` for older L-systems created before the bug was fixed.	`on`

8.2.7 Deprecated commands

The following commands have been replaced but may still exist in older models.

Command	Description	See new command
`view:` *id px py pz scale ux uy uz*	Define the view transformations to be used for view window *id*: - *px py pz*: The center of the view (pan) relative to the center of the bounding box. - *scale*: The scale of objects. - *ux uy uz*: The up direction.	`view:` (Section 8.2.1)

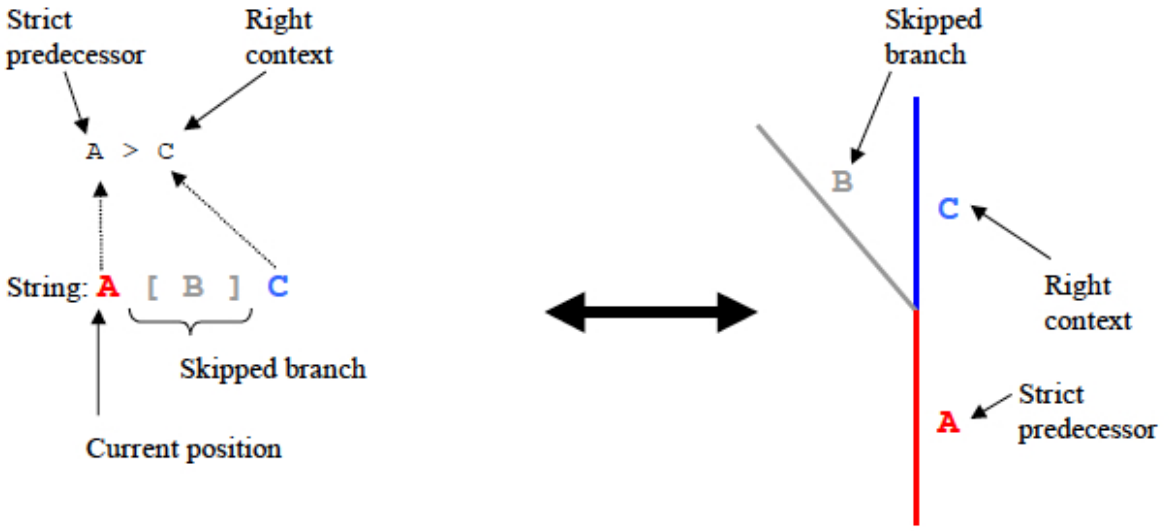


Figure 1: Matching right context. Lateral branches are implicitly ignored.

9 Appendix: Production Matching

When rewriting the string it is necessary to determine which production must be applied to each module in the string. The process of determining the applicable production is called *production matching*. For every module in the string, productions are checked for matching. The productions are checked in the order in which they are specified in the L-system. For a production to match, all three components of the predecessor (left context, strict predecessor and right context) must match. The rules for matching each of these components are different. This is because the L-system string is a means of representing branching structures and symmetric operations on the string do not (in general) correspond to symmetric operations on the branching structure.

This section contains a detailed explanation of rules that control the process of production matching. The notation used here utilizes symbols [and] to denote the beginning of a branch and the end of a branch (modules `SB` and `EB` in *lpfg*).

When the strict predecessor is compared with the module(s) at the current position in the string, they must match exactly.

When matching the right context, if a module in the context is not the same as the module in the string the following rules apply:

- If a module in the string is [and the module expected is not [then the branch is skipped. This rule reflects the fact that modules may be topologically adjacent, even though in the string representation of the structure the two modules may be separated by modules representing a lateral branch `B` (see Figure 1).
- When a branch in the right context ends (with a right bracket) then the rest of the branch in the string is ignored by skipping to the first unmatched]. This rule also reflects the topology of the branching structure, not its string representation. For example in Figure 2, module `C` is closer to `A` than `D`.
- If multiple lateral branches start at a given branching point, then the predecessor in Figure 2 would check the first branch (see Figure 3). To skip a branch it is necessary to specify explicitly which branch at the branching point should be tested (see Figure 4). This notation is a simple consequence of the rule presented in Figure 2. In the current L-system notation there is no

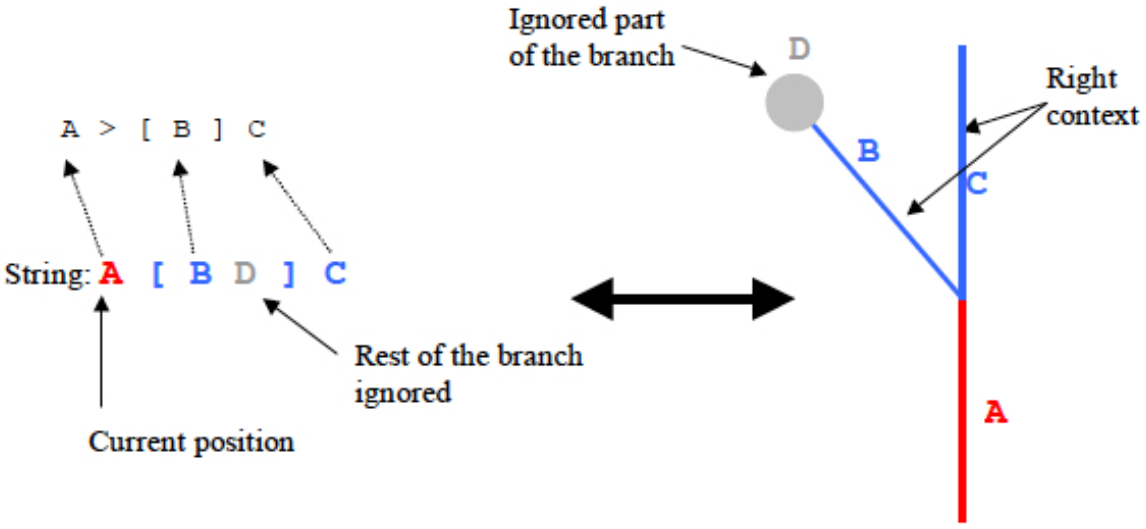


Figure 2: Matching right context. Remainder of lateral branch is implicitly ignored.

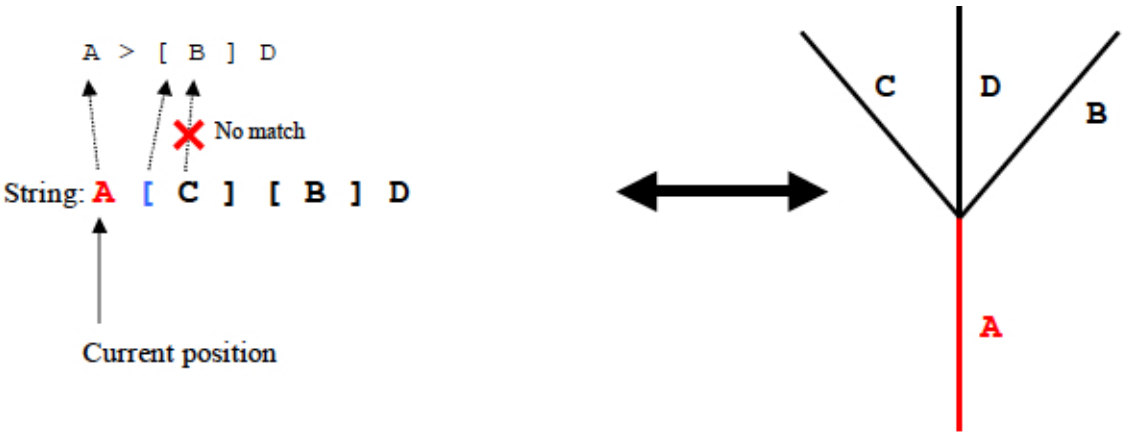


Figure 3: Problem with multiple lateral branches when matching the right context.

shortcut to specify the second, third etc. lateral branch in a branching point without explicitly including pairs of [ ] in the production predecessor. There is also no way to specify "any of the lateral branches".

When matching the left context the following rules apply:

- Module [is always skipped, since the preceding module will be topologically adjacent (see Figure 5).
- If the module in the string indicates the end of a branch then the entire branch is skipped (Figure 6).

The rule illustrated in Figure 5 is a pronounced manifestation of the asymmetry in the left-context / right-context relationship: module `C` is the left context of both `A` and `B`. But the right context of `C` is `B` (unless [ ] delimiters are used explicitly). The left context may be thought of as the parent module: the module before (below) the branching point. It is then natural to say that `C` is the parent module of both `A` and `B`.

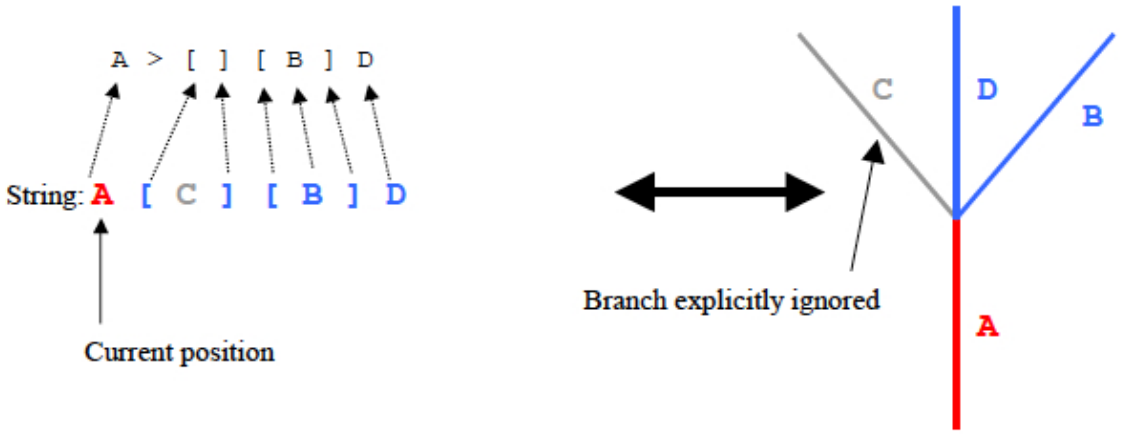


Figure 4: Explicit enumeration of lateral branches in the right context.

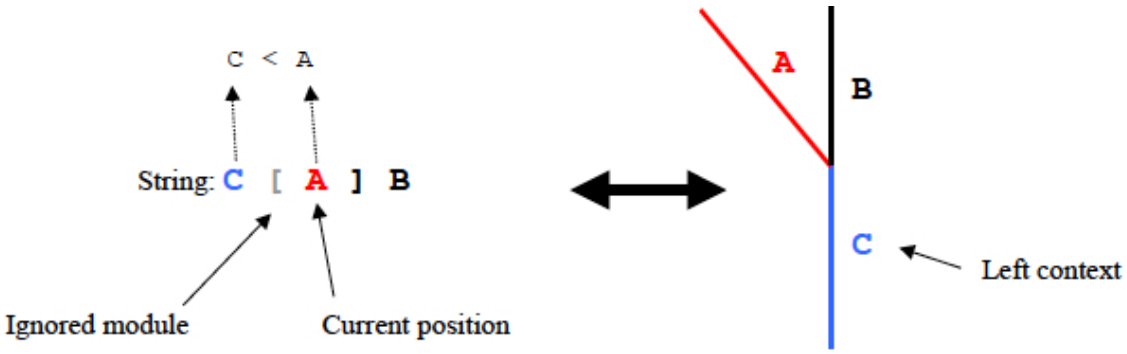


Figure 5: Matching left context. The beginning of the branch is implicitly ignored..

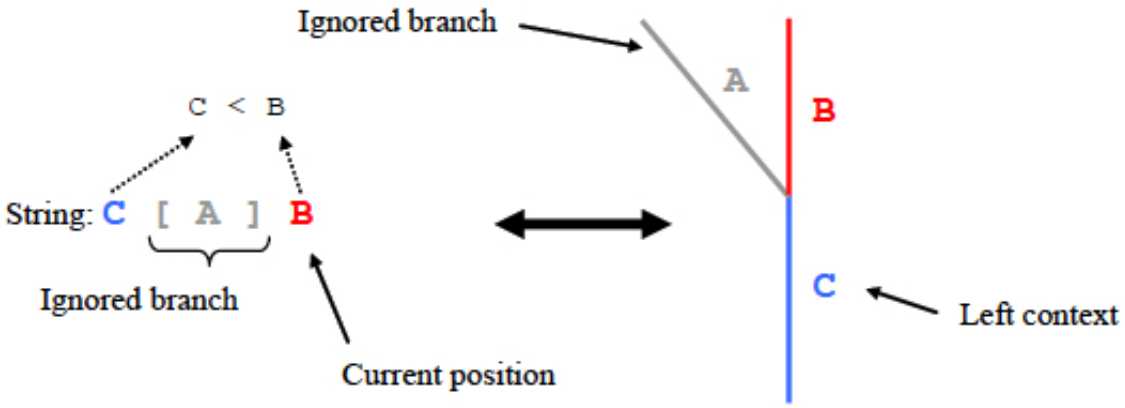


Figure 6: Matching left context. The lateral branches are implicitly ignored..

10 Appendix: Deprecated / Undocumented features

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

10.1 B-spline surfaces

B-spline surfaces and the editor for them, *splineEdit*, are no longer supported. All surfaces should be defined as Bézier patches (see the *bezieredit* and *stedit* tools in the *Vlab Tools* manual). However, the constructs used to manipulate B-spline surfaces still exist within *lpfg*.

10.1.1 Defining and drawing B-spline surfaces

Predefined B-spline surfaces are specified in the view file using the command:

`bsurface:` *filename*`.s` *scale sdiv tdiv txid*

where the parameters are defined the same as for the `surface` command used to specify Bézier surfaces (see Section 8.2.3). The surface is drawn using the module:

```
BSurface(int id, float scale)
```

where *id* is the surface file number, and *scale* is a uniform scaling factor. The surface is drawn at the current location and orientation of the turtle.

10.1.2 Dynamic B-spline surfaces

B-spline surfaces can also be manipulated from within the L-system using constructs equivalent to their Bézier surface counterparts. See Section 7.1 for more details on dynamic surfaces.

The B-spline surface classes are:

`BsurfaceObjS` for surfaces with up to 10x10 control points
`BsurfaceObjM` for surfaces with up to 32x32 control points

and the following methods are available for each class:

Method	Description
`Set(int i, int j, const V3f& v)`	Initialize the coordinates of a control point.
`Get(int i, int j) const`	Get the coordinates of a control point.
`Scale(V3f scale)`	Non-uniformly scale the surface by a different factor in each direction.

In addition, there are functions and modules to get and draw dynamic B-spline surfaces, similar to the Bézier surface function (Section 6.3) and module (Section 5.6):

Module	Description
`BsurfaceObjS GetSurface(int id)` `BsurfaceObjM GetSurface(int id)`	Return the control points of the predefined B-spline surface specified in the view file as `id`. If the surface contains more than one patch, only the first patch is returned.
`DBSurfaceS(BsurfaceObjS s)` `DBSurfaceM(BsurfaceObjM s)`	Draw the dynamic B-spline surface `s`.

10.2 Tablet interaction

Support for a tablet has not been tested with the newest versions of iOS or the newest tablet drivers. It consists of:

- a command line argument: `-tablet`
- the function: `struct TabletStatus GetTabletStatus()`

When `-tablet` is specified on the command line, tablet input is consider separately from mouse input, providing two separate input devices. With this option the tablet pointer is input using the `GetTabletStatus()` function, not the `GetMouseStatus()` function. However, the tablet cannot be used to change the view or insert `MouseIns` modules.

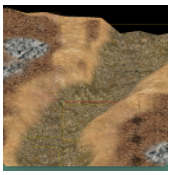
The `GetTabletStatus()` function returns the state of the tablet pointer, similar to `GetMouseStatus()`, including tablet pressure and pen angle if the tablet supports it. `TabletStatus` is a predefined data type:

```
struct TabletStatus {
  bool connected;

  int viewX, viewY;
  float azimuth, altitude;
  double pressure;
  unsigned int cursorT, buttonState;

  V3d atFront, atRear;
};
```

10.3 Terrain



See object: TerrainDemo

Functionality related to terrain is kept to provide backward compatibility with older models, in particular models of plants in the Rhynie chert. Unfortunately, no further documentation (including `.patch` file format) is available, and the terrain editor no longer works. However, files used by this editor are kept in the example object, in case the editor is "revived" in the future.

Terrain functionality begins with the inclusion of a texture and a terrain in the view file using the commands:

`texture:` *filename*`.rgb`
`terrain:` *filename*`.patch` *levels scale offset grid txid UTiling VTiling*

The texture must be in RGB format. The parameters to the `terrain:` command are:

Parameter	Description	Default
filename`.patch`	A predefined terrain file.	
levels	The number of levels to be used in the LOD system, where 1 is the lowest level. Must not exceed the `Number of Resolutions to Export` field in the Terrain Editor program at the time of export. This parameter is required.	
scale	The value that should be multiplied to the position of every point in the terrain when the file is loaded.	1
offset	The distance the camera must be to a patch of the terrain before it changes its level of detail. A value of 1 is conservative and will work well on slower systems, while 50 will generally display the highest level of resolution.	1

Parameter	Description	Default
grid	Display the terrain LOD system on the screen as yellow rectangles, when set to `on`.	`off`
UTiling *VTiling*	The number of times the texture will be tiled in the u and v directions.	*UTiling* = 1 *VTiling* = 1

Parameters for drawing the terrain can be set with predefined functions:

Function	Description
`bool terrainHeightAt (V3f pointInWorldSpace, V3f &pointOnTerrain)`	Project a ray along the Y axis, (0,1,0), from the `pointInWorldSpace`, and return the `pointOnTerrain` at which the ray intersects the terrain. If the ray intersects the terrain mesh, return `true`, otherwise return `false`.
`void terrainVisibilityAll (VisibilityMode mode)`	Set the visibility of all terrain to `Shaded, Hidden` or `Wireframe`
`void terrainVisibilityPatch (VisibilityMode mode, int level, V3f point)`	Set the visibility of a single patch of terrain to to `Shaded, Hidden` or `Wireframe`. The patch of terrain is selected by casting a ray along the Y axis at `point`, and choosing the visible patch that the ray intersects. All child patches are also set to this `mode`. The `level` parameter is no longer used.
`void scaleTerrainBy (float value)`	Multiplying the x, y and z components of each point of the Terrain by `value`.

See Section 6.2.1 for a description of the predefined data type, `V3f`.

The terrain mesh is drawn using the predefined module:

```
Terrain(CameraPosition)
```

which draws the terrain using the current position and orientation of the turtle, and the current color. To ensure the most current camera position is used, it is generally defined just before the `Terrain` module:

```
CamerPosition cameraPos;
...
cameraPos = GetCameraPosition(0);
produce Terrain(cameraPos);
```

See Section 6.4 for a description of `GetCameraPosition()`.

10.4 String verification

A mechanism was developed for verifying the main elements of the L-system string during the derivation process. It was available only in batch mode, using the statement:

`VerifyString:` *module list*;

where the *module list* contained module names only, not parameter values. After deriving the string, *lpfg* would compare it with the *module list*. If the modules matched, the message *Verify: Success* would be printed to the standard output. Otherwise, the message would be *Verify: Fail*, and two files would be created, one containing the *module list* and one containing the derived string:

```
Verify_[lsystem]_expected.txt
Verify_[lsystem]_actual.txt
```

where `[lsystem]` is the name of the L-system file specified in the *lpfg* command line.

11 Credits

The original implementation of the L+C language was by Radoslaw Karwowski in the scope of his Ph.D. thesis [2], and published in [3]. Further extensions have been made by Brendan Lane [4], Thomas Burt, Mikolaj Cieslak, and Pascal Ferraro.

Vlab uses a modified version of the rendering program *rayshade* written by Craig Kolb[5] for the Save as Rayshade option.

12 Document revision history

Date	Description	By
2002	First version in Microsoft Word	Radoslaw Karwowski
2010 & 2014	Updates made to Microsoft Word version	Radoslaw Karwowski Brendan Lane
2021	Updated and re-written in LaTex	Lynn Mercer Przemyslaw Prusinkiewicz Pascal Ferraro Mikolaj Cieslak

References

[1] D. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22:403–434, 1976.

[2] Radoslaw Karwowski. *Improving the Process of Plant Modeling: The L+C Modeling Language*. PhD thesis, University of Calgary, 2002.

[3] Radoslaw Karwowski and Przemyslaw Prusinkiewicz. Design and implementation of the L+C modeling language. *Electronic Notes in Theoretical Computer Science*, 86(2):19pp, 2003.

[4] Przemyslaw Prusinkiewicz, Radoslaw Karwowski, and Brendan Lane. The L+C plant modelling language. In J. Vos, L.F.M. Marcelis, P.H.B. de Visser, P.C. Struik, and J.B. Evers, editors, *Functional-Structural Plant Modeling in Crop Production*, pages 27–42. Springer, 2007.

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