vlab

The Virtual Laboratory

# LPFG
# Reference Manual

Last updated: December 11, 2022

# Contents

## 1 Introduction

*lpfg* is a plant modeling program based on the formalism of L-systems but also has many other applications, such as the generation of fractals. Models are defined using the L+C language, which extends the syntax of C++ to include constructs inherent in L-systems. It is assumed that the reader is familiar with C/C++ and with the basic concepts of modeling with L-systems as introduced in [1] and the **CPFG Reference** manual. While simple L-system models may be created more compactly using *cpfg*, the powerful programming constructs in *lpfg* make it preferable for the development of complex models [2; 3].

This is a reference manual with limited examples. However, sample *vlab* objects are indicated in some sections to illustrate 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] [-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. In general, the only mandatory argument is the L-system file, *lsystem*`.l`, which contains the L+C code for the model. (See the `-dll`/`-o` options for an exception.)

#### 1.1.1 Command line options

| Option | 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 into object code, and create a shared library file called `lsys.so`. This library can subsequently be used by *lpfg* to run the L-system model without recompiling (see the `-dll` and `-o` options). Do not run the simulation. |
| `-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 ***vlab* Environmental Progams** manual for more information. |
| `-cn` | Check for numerical errors in the arguments of turtle movement modules. When this option is included, *lpfg* checks that the arguments to modules such as `F` and `Right` are valid numbers. It is useful for finding division-by-zero errors in the model. |
| `-d` | Start *lpfg* in debug mode. Information regarding the execution of the program is sent to the standard output. This mode is intended for *lpfg* developers only. |

| Option | Description |
|---|---|
| `-dll` *filename*`.so`<br>`-o` *filename*`.so` | Use the precompiled library file *filename*`.so` (usually `lsys.so` unless explicitly renamed) instead of *lsystem*`.l`. See the `-c` option. This is useful when a simulation will be run many times but the L-system does not change (although other input files may). These two options are synonymous. |
| `-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`. |
| `-out` *filename* | In batch mode, use *filename* for the output string. This will be a text file.<br>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:<br>`expl` = explicit<br>`cont` = continuous<br>`trig` = triggered<br>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. 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 L+C modeling language in *lsystem*`.l` is described in this manual, as are the *lpfg*-specific input files, *animation*`.a` and *view*`.v` (Sections 8.2 and 8.1 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.

| File | Description |
|---|---|
| *lsystem*.`l` | Defines the L+C model. |
| *view*.`v` | Defines the drawing and viewing parameters, including setting the view, rendering, surfaces, etc. See Section 8.1. |
| *animation*.`a` | Defines the parameters for controlling animation of the model. See Section 8.2. |
| *colormap*.`map`<br>*material*.`mat` | Specifies 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 for information on how to use the colors within *lpfg*, and the *palette* and *medit* tools in the ***vlab* Tools** manual. |
| *contour*.`con`<br>*contourset*.`cset` | Specifies contours defined as B-spline curves. 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 6.2.3 for information on how to access the contours within *lpfg*, and the *cuspy* and *gallery* tools in the *Vlab Tools* manual. |
| *function*.`func`<br>*functionset*.`fset`<br>*timeline*.`tset` | Specifies 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 6.2.2 for information on how to access the functions within *lpfg*, and the *funcedit, gallery*, and *timeline* tools in the ***vlab* Tools** manual. |
| *parameter*.`vset` | Defines parameters that can be read from the L-system without recompiling. See Section 6.2.1 for the `val` function used to access the parameters within *lpfg*, as well as the file format. |
| *environment*.`e` | Specifies the environmental program and its parameters. See the ***vlab* Environmental programs** manual for more information. |

## 1.2 User Interface

When *lpfg* is opened, it typically runs the L-system and graphically interprets the final 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 | Rotate around the Y axis by moving the mouse horizontally, and around the X axis by moving the mouse vertically. |
| Roll | SHIFT + middle mouse | Roll clockwise around the Z axis by moving the mouse to the right, and roll counter-clockwise by moving the mouse to the left. |
| Zoom | COMMAND + left or middle mouse | Zoom in by moving the mouse up, and zoom out by moving down. |

| Action | Key & Mouse | Description |
|---|---|---|
| Pan | SHIFT + left mouse | Move model in all directions using the mouse. |
| Change frustrum angle | COMMAND + middle mouse | 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 to the default file name and type. The initial file name is the same as the original L-system file, with a PNG extension. Use Save as to change the name and/or file type. The file used in the Save as window will then become the default file for this Save command. |
| Save as ... | Open a dialog window to save the current view with a different name and/or in a different format. (See below for a description of the dialog window). |
| 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*. |

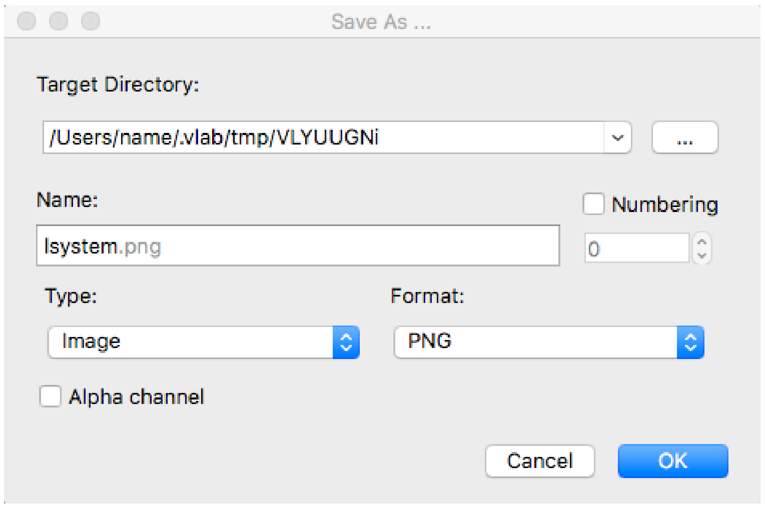


Figure 1: An example of the Save as... dialog window.

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 | |

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

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

### 1.2.3 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 8.2), 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 | Display consecutive animation frames after each `step` derivation steps until `last frame` is reached or passed. | 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*. | |

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

## 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 the animation file only. |
| Colormap<br>Material | New rendering |
| Surface<br>Texture | New rendering |
| Function<br>Contour<br>Timeline<br>Parameters | New run |

# 2 The L-system file

L-system files use the L+C modeling language [4; 5; 6]. 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).

Statements may appear in any order with the following restrictions:

- The `#include` statement should be the first line in the file.[1] It contains embedded header files with declarations and definitions used by *lpfg* and the L2C translator, including predefined types.[2]
- All elements referred to in a statement must be declared beforehand:
  - Types used as parameters of a module must be declared before the module is declared.
  - Modules must be declared before they appear in any statements.
- Productions are matched in the order in which they are listed in the L-system file. (The set of potentially applicable productions may change, however, from one derivation step to the next. See Section 4.4.)

## 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*. Non-integer values are turncated to an integer. However, some care should be taken to ensure that *expression* is constant, as the behaviour of *lpfg* is undefined if the value changes during the simulation.

## 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]Optional non-*lpfg*-specific header files should be included before `lpfgall.h`, since this file signals the start of the translation from L+C to C++.

[2]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);
```

## 2.3 Module and type declarations

### 2.3.1 Modules

L+C requires that all modules be declared before they appear in any other statements. No module can be declared twice, even with a different number of parameters. Many standard modules are predefined (see Section 5) and, therefore, must not be redeclared. The syntax for declaring a new module is:

`module` *name*( *ptypes* );

where *name* is the module name, and *ptypes* is a list of the parameter types. If a module has no parameters, the parentheses can be omitted. For example:

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

Note that, unlike function arguments, module parameters have no names (they are identified by position). 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*/);
```

### 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(s) to be changed, as well as the context they must be found in for the production to apply; and the *production body* defines how the predecessor will be changed in the next derivation step if the production is applied. 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(): { ... }
```

All module parameters in the predecessor must be listed and given unique names, even if they are not used in the production body. In addition, 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*:

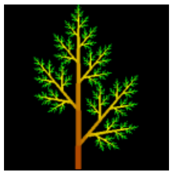
For example, the production

```
F(x) > G(y): { F(x+1) }
```

will replace `F(x)` with `F(x+1)` in the next derivation step only if `G(y)` is to the right of `F(x)` in the string. `G(y)` is not replaced by this production: it remains in the string unless it is replaced by another production in which it is the strict predecessor.

### 3.1.3 Left and right new context

In each derivation step, the right and left context constructs above are matched to modules in the current (or "old") string, in order to produce the next (or "new") string. 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 new 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 (Section 6.1.2), usually called within a control statement (see Section 4.1). When the string is matched from right to left ("backward"), the right new context can be used 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, a production with the predecessor:

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

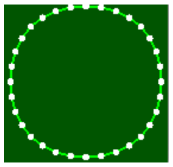
will match the module `B()` in the current string if the derivation is proceeding in the forward direction, the last module in the new string is `Age(age,length)`, and the current string has another `B()` to the right of the strict predecessor.

### 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 no 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 instance, 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 ending with a `return` statement (or reaching the end of the production body without applying a `produce` statement), in which case *lpfg* 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. 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)
```

If the production body ends without a `produce` statement, the production is not applied, and the partial successor is ignored.

## 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:

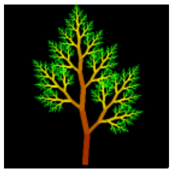
```
G(length):
{
   float x,y;
   if (InLeftContext(F(x)) && InRightContext(H(y))
        { ... };
}
```

This applies to InNewContext expressions as well where

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

is equivalent to:

```
G(length):
{
     float x;
     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.

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):  {
     float x;
     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):
{
   float fl, fr, a, b;
   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):
{
   float f1, fr, a, b;
   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.

For details regarding context matching in branching structures see Section 9.

# 4 Control statements

## 4.1 Start and End statements

There are four statements used to specify C++ statements to be executed outside of productions at specific points during the L-system execution process:

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

Each of these statements 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 that the `End` statement is called after the final derivation step. Therefore, in Animate mode, if the animation is stopped or rewound before it reaches the final derivation step, the `End` statement is not 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 for exceptions that may occur when modeling branching structures). It is often convenient, however, to consider only certain module types. There are two statements that can be used for this purpose:

`ignore:` *module list*;

or

`consider:` *module list*;

where *module list* is a sequence of module names separated by spaces. 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 5.4) are **always** considered. Listing them in an `ignore` or `consider` statement has not effect.

## 4.3 Decomposition and Interpretation Rules

While productions are rules which define how the model advances over time, decomposition rules divide modules into simpler components, and interpretation rules specify how modules should be displayed.

### 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. 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 format is:

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

where each rule (predecessor/successor) has the same syntax 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 in 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 $n$ is (truncated to) an integer. 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 graphical 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 creating the visualization. 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 5.12).

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 as a sphere of radius `age`.

### 4.3.3 Rule blocks

Generally, an L-system is written as an axiom followed by a block of productions, then decomposition rules, and finally interpretation rules:

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

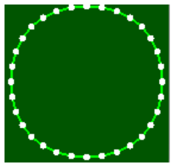
However, in some cases it may be convenient to change this order, which will require a `production` statement to return to regular productions after rules of a different type have been listed.

For example, in the following code the L-system is organized into a block of rules pertinent to module `A()`, and then another block of rules pertinent to module `X()`, using a `production` statement to end the first set of interpretation rules.

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

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

## 4.4 Production groups

In some models it is convenient to organize rule blocks into groups, such that only one specific group is considered in a given derivation step. 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.


See object: B-spline

To specify an additional group, use the `group` statement:

> `group` *id*:

where the group identifier, *id*, is an integer constant (not an expression or enumerated value) with a value greater than zero. A group ends with another `group` statement, or with an `endgroup` statement.

When *lpfg* is started, the default (group 0) rules are applied. To select another group for the subsequent derivation step, use the function:

> `UseGroup(`*id*`);`

where *id* evaluates to an integer. It can be called at any time, but only takes effect at the beginning of the next derivation step. Consequently, it is often called in a `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. Note that the `interpretation` block returns to `group 0`; therefore, the productions in this block will be used in each 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. 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.). See Section 6.3.1 for a description of the predefined vector data types, `V2f, V2d, V3f,` and `V3d`.

### 5.1 Position and drawing

| **Module** | **Description** | **Equiv. in *cpfg*** |
|---|---|---|
| `F(float d)`<br>`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 5.6), the final position is recorded as a vertex of the current polygon. | `F(d)`<br>`G(d)` |
| `f(float d)`<br>`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 5.6), the final position is recorded as a vertex of the current polygon. | `f(d)`<br>`g(d)` |
| `MoveTo(float x, float y, float z)` | Move the turtle to point `(x,y,z)` | `@M(x,y,z)` |
| `MoveTo3f(V3f p)`<br>`MoveTo3d(V3d p)`<br>`MoveTo2f(V2f p)`<br>`MoveTo2d(V2d p)` | Move the turtle to point `p`. | |
| `MoveRel3f(V3f p)`<br>`MoveRel3d(V3d p)`<br>`MoveRel2f(V2f p)`<br>`MoveRel2d(V2d p)` | Move the turtle to the turtle's current position + `p`. The heading, left and up vectors are not changed. | |
| `LineTo(float x, float y, float z)` | Draw a line from the turtle's current position to point `(x,y,z)`. | |
| `LineTo3f(V3f p)`<br>`LineTo3d(V3d p)`<br>`LineTo2f(V2f p)`<br>`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)`<br>`LineRel3d(V3d p)`<br>`LineRel2f(V2f p)`<br>`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)`<br>`Line3d(V3d p1, V3d p2)`<br>`Line2f(V2f p1, V2f p2)`<br>`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. | `@D(s)` |

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

## 5.2 Turtle rotations

In the following modules, parameter `a` is an angle expressed in degrees.

| 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 $\epsilon$, 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 $\epsilon$, no rotation will occur. | |
| `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 $\epsilon$. | `@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 in some cases. This has been fixed; however, in order to keep compatibility with existing models, the view file command `corrected rotation` can be used to switch between the original and corrected behaviour (see Section 8.1.3).

## 5.3 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`. In pixel mode, the result is undefined if `v` $< 0$, and in other modes if `v` $\leq 0$. | `#(n)` `!(n)` |

## 5.4 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`. | `%` |

## 5.5 Circles and spheres

| Module | Description | Equiv. in *cpfg* |
|---|---|---|
| `Circle0()` | 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 front plane, with inner radius = `r - width/2` and outer radius = `r + width/2`, where `width` is the current line width. Note that it is always drawn in the front plane even when the object is rotated. This is different from *cpfg* where it would keep its orientation in relationship to other elements in the scene. | `@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 5.8), or the `contour sides` command in the view file (Section 8.1.3). For spheres, there will be `contour sides` longitudinal sections and `(contour sides+1)/2` transversal sections.

## 5.6 Polygons, rhombi, and isosceles triangles

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

## 5.7 Surfaces and Meshes

Predefined surfaces and meshes are specified in the view file (Section 8.1.5), 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. | |

## 5.8 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). Contours are referenced sequentially by an `id` in the order in which they are listed, starting with 1.

| 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`). | |

## 5.9 Textures

Cylinders and surfaces can be texture mapped using an image file specified in the view file (Section 8.1). Textures are referenced sequentially by an `id` in the order in which they are listed in the view file, starting with 0.

| Module | Description |
|---|---|
| `CurrentTexture (int texid)` | Use texture `txtid` to texture map a surface or cylinder. If `txtid` = -1, texture mapping is turned off. |
| `TextureVCoeff (float vscale)` | Scale the texture mapped on cylinders in the v direction (along the cylinder axis). `vscale` is the portion of the 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 `vscale` to 0.1. If the texture v coordinate is greater than 1, the texture wraps. This module only affects the texturing of cylinders; surfaces are not affected. |

## 5.10 Labels

| Module | Description | Equiv. in *cpfg* |
|---|---|---|
| `Label(char* str)` | Print the string, `str`, at the current turtle position. | `@L(str)` |

Note that if the `Label` module is drawn before other objects when `render mode` is set to `shadows`, the shadows will not be rendered. For example:

```
    Label("a") F(1) F(1) F(1)...
```

will break shadow mapping, but the following will not:

```
F(1) F(1) F(1)...  Label("a")
```

## 5.11 Tropism Control

Tropisms are defined in the view file (Section 8.1.4). They are assigned consecutive integer *id* numbers, starting with 1, in the order in which they appear in the file.

| **Module** | **Description** | **Equiv. in *cpfg*** |
|---|---|---|
| `SetElasticity (int id, float v)` | Set the elasticity parameter of tropism or torque `id` to `v`. This is equivalent to the `E:` parameter of the `tropism` and `torque` commands in the view file. | `@Ts` |
| `IncElasticity (int id)` | Increment the elasticity parameter of tropism or torque `id` by the value defined by the `S:` parameter of the `tropism` and `torque` commands in the view file. | `@Ti` |
| `DecElasticity (int id)` | Decrement the elasticity parameter of tropism or torque `id` by the value defined by `S:` parameter of the `tropism` and `torque` commands in the view file. | `@Td` |
| `Elasticity (float v)` | Set the elasticity or a “simple” tropism (command `stropism` in the view file) to `v`. | _ (under-score) |

## 5.12 Query modules

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.

If there are multiple views (Section 7.3), the interpretation rules in `vgroup 0` will be used.[3]

| **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)` |

[3]`Vgroup 0` is a special group, distinct from the vgroups defined using the `window` command in the view file (Section 8.1.1).

## 5.13 Environment communication

| Module | Description | Equiv. in *cpfg* |
|---|---|---|
| `E1(float v)`<br><br>`E2 (float v1, float v2)`<br><br>`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 ***vlab*** **Environmental Programs** manual for more details, including the definition of the `EA20Array` data type.

## 5.14 Dynamic view control

| Module | Description |
|---|---|
| `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 8.2). |

## 5.15 Mouse interaction

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 | Equiv. in *cpfg* |
|---|---|---|
| `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. | `x` |
| `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.

# 6 Predefined functions

## 6.1 Controlling the simulation

### 6.1.1 Simulation progress

| Function | Description |
|---|---|
| `StepNo()` | Return the current derivation step number. |
| `void RunSimulation()` | Run the simulation. Including this function in the `Start` block (Section 4.1) will start the animation as soon as *lpfg* is called, without selecting the `Run` menu item (Section 1.2.3) or using the keyboard shortcut to start. |
| `void PauseSimulation()` | Pause the simulation after completing the current derivation step. The simulation can be resumed using the `Step`, `Run` or `Forever` menu options (Section 1.2.3). The `End` statement block (Section 4.1) is not executed. |
| `void Stop()` | Stop the simulation. The `End` statement block (Section 4.1) is executed after the current derivation step. The animation can be rewound, but cannot be advanced further using the Animate menu items (Section 1.2.3) or the keyboard shortcuts. |
| `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 8.2). If it is `off`, this function has no effect. |
| `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 `display on request` = `on` in the animation file (Section 8.2), this call must be preceded by a `DisplayFrame()` function so that the frame buffer is updated. If this function is called outside of Animate mode, it is ignored. The actual saving is done at the end of the derivation step: if `Stop()` is called in the same step, the frame will not be output. |

### 6.1.2 L-system derivation

L-system derivation can be affected by groups (Section 4.4) and Gillespie groups (Section 7.2) using the following 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. |
| `void SeedGillespie(long seed)` | Seed the pseudo-random number generator used by Gillespie groups (see Section 7.2). |

Derivation direction, used with new context functionality (Section 3.1.3), is determined by the following 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()` | Return 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. |

### 6.1.3 Communication with the environment

| Function | Description |
|---|---|
| `void Environment()`<br>`void NoEnvironment()` | Specify whether the "interpretation for environment" step should be performed after the current derivation step. This affects whether environmental information will be available in the next derivation step. Interpretation for environment is performed after the `EndEach` block, so these functions can be used in the `StartEach` or `EndEach` blocks with the same effect. |

See the ***vlab*** **Environmental Programs** manual for more information.

## 6.2 Accessing externally defined entities

Parameters, functions, contours and surfaces accessed using functions listed in this section can be manipulated without modifying, re-reading and recompiling the L-system file. This facilitates and accelerates exploration of the models, using the tools in the ***vlab*** **Tools** manual. To see the results of changes, rerun the L-system using the `New run` menu item (Section 1.2.2). To see the results immediately, set *lpfg* and the tool to continuous refresh mode (see Section 1.3).

### 6.2.1 Parameters

Parameters may be defined in a `.vset` file (Section 1.1.2) which can be modified using the `panel` tool. Each parameter is identified by a name (`pname`) which *lpfg* maps to an integer number (using a `#define` statement invisibile to the user).

The following function is used to retrieve the value of a parameter from the file:

| Function | Description |
|---|---|
| `float val(int 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 n*

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

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

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

the L-system can retrieve the values of these parameters with the statements:

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

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

### 6.2.2 Graphically defined functions

Graphically defined functions can be input into *lpfg* using three types of input files: *function*`.func`, *functionset*`.fset`, and *timeline*`.tset` (see Section 1.1.2). Each function is identified by a name, `fname`, which is mapped to a sequential integer number (using a `#define` statement invisible to the user).

The functions can be accessed within the L-system using the following calls:

| Function | Description |
|---|---|
| `float func(int fname, float x)` | Return the value of function `fname`, specified in a `.func` or `.fset` file. The parameter `x` is clamped to the interval [0,1]. |
| `float pfunc(int fname,`<br>`float x, float min, float max)` | Return the value of function `fname`, specified in a `.func` or `.fset` file. The function is evaluated over the interval [`min,max`]. The parameter `x` is clamped to that interval. |
| `float tfunc(int id, float x)` | Return the value of function `fname`, specified in a `.tset` file. The function is evaluated over the interval specified in the file. The parameter `x` is clamped to that interval. |
| `V2f funcTangent (int id, float x)` | Return the tangent to function `fname`, specified in a `.func` or `.fset` file. The parameter `x` is clamped to the interval [0,1]. |

### 6.2.3 Graphically defined curves

Similar to *cpfg*, *lpfg* supports B-spline curves, defined via their control points. Planar curves can be defined using the *cuspy* tool. See the ***vlab* Tools** manual for the file format, which can also be used to define 3-dimensional contours. Contour files are listed on the command line (Section 1.1.2). Each contour is assigned an `id` in the order in which it was listed, starting at 1.

See Section 5.8 for the use of contours to create generalized cylinders.

| Function | Description |
|---|---|
| `float curveX(int id, float s)`<br>`float curveY(int id, float s)`<br>`float curveZ(int id, float s)`<br><br>`--- curveXY(int id, float s)`<br>`V2f curveXYf(int id, float s)`<br>`V2d curveXYd(int id, float s)`<br><br>`--- curveXYZ(int id, float s)`<br>`V3f curveXYZf(int id, float s)`<br>`V3d curveXYZd(int id, float s)` | Return the coordinates of curve `id` defined in the contour file. Parameter `s` is the normalized arc-length positin clamped to the interval [0,1]. The `curveXY` and `curveXYZ` functions return a structure type internal to *lpfg* and may require explicit casting to `V3f` or `V3d` within the L-system code. |
| `V3f curveNormal(int id, float s)` | Return the normal vector to the curve `id` at position `t`. |
| `void curveScale (int id,`<br>`float x, float y, float z)` | Scales curve `id` by the factors `x, y,` and `z`. |

| Function | Description |
|---|---|
| `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. |
| `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 file. The file is not re-read. |

### 6.2.4 Graphically defined surfaces

Predefined surfaces can be created using either the *bezieredit* or *stedit* tool and are listed in the view file (Section 8.1.5). Surfaces are assigned an `id` in the order in which they were listed, starting at 1.

| Function | Description |
|---|---|
| `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). |

## 6.3 Auxiliary math functions

### 6.3.1 Vector algebra

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; };
```

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. The argument vector is not affected. |
| `Set(x,y)`<br>`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);
```

However, there are also `Normalize` functions:

| Function | Description |
|---|---|
| `V2d Normalize(V2d v)`<br>`V2f Normalize(V2f v)`<br>`V3d Normalize(V3d v)`<br>`V3f Normalize(V3f v)` | Normalize vector `v`, and return a copy of this vector. |

And, unless the preprocessor symbol `NOAUTOOVERLOAD` is specified before `#include lpfgall.h`, more operators are defined on vectors, including the addition and subtraction of two vectors of the same type, unary change of sign (direction), multiplication and division of a vector by a scalar, dot product, and the assignment operators `+=`, `-=`, `*=`, and `/=`. Moreover, the cross product is defined on `V3f` and `V3d` with operator `%`. Some usage 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;
```

### 6.3.2 Random number generation

The random number functions, `ran` and `sran`, provide convenient access to simple random number generation, based on Unix `drand48()`.

| Function | Description |
|---|---|
| `float ran(float range)` | Generate a peseudo-random number uniformly distributed over [0, `range`]. |
| `void sran(long seed)` | Seed the pseudo-random number generator used by `ran`. Use this function in the `Start` block to ensure every run is identical, even after rewinding. |

## 6.4 View-related functions

Views are identified by name in the view file (Section 8.1.1), which *lpfg* maps to sequential integer numbers internally (using `#define` statements invisible to the user). Therefore, the following functions have an integer parameter which can be specified using the view name. See Section 7.3 for more information on multi-view mode, and the use of `vname`.

| Function | Description |
|---|---|
| `void UseView(int vname)` | Activate the view, `vname`, specified in a `window` command in the view file. |
| `float vvXmin(int vname)`<br>`float vvYmin(int vname)`<br>`float vvZmin(int vname)`<br>`float vvXmax(int vname)`<br>`float vvYmax(int vname)`<br>`float vvZmax(int vname)` | Return the coordinate of the bounding box of view `vname`. |
| `float vvScale(int vname)` | Return the current projection scaling factor of view `vname`. |
| `CameraPosition GetCameraPosition(0)` | Get the current position of the camera. |

`CameraPosition` is a predefined data type:

```
struct CameraPosition {
  V3f position, lookat;
  V3f head, left, up;
  float scale;
};
```

## 6.5 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 id)` | Add a menu item, `label`, to a user-defined menu, with a numerical `id` to be returned when the item is selected. The menu is accessed by holding down the '4' key and clicking the left mouse button over the *lpfg* window. |
| `int UserMenuChoice()` | Return the `id` defined by `UserMenuItem()` associated with the last selection made from the user-defined menu since the previous call to this function. |
| `void UserMenuClear()` | Clear the user-defined menu. |

See Sections 7.4 and 7.5 for examples of mouse interactions and user-defined menus, respectively..

## 6.6 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* when *lpfg* is not called from and 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. This function should be called in a control block, not within a production. |
| `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. |

# 7 Advanced topics

## 7.1 Dynamic surfaces

Single-patch Bézier surfaces can be dynamically created and/or manipulated from within the L-system. This is useful, for example, when creating an animation with "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, and
- manipulation of individual control points,

### 7.1.1 Creating dynamic surfaces

A dynamic surface can be initialized for further manipulation by:

- Using the `GetSurface` function (Section 6.2.3) to get the control point coordinates of a predefined surface specified in the view file (Section 8.1)
- 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 6.3.1 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)
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(const V3f& scale)
```

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

```
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. Then use the predefined `DSurface` module (Section 5.7) to draw the resulting surface. For example:

```
interpretation:
Leaf(age):
{
     SurfaceObj young = GetSurface(L_YOUNG);
     SurfaceObj mature = GetSurface(L_MATURE);
     SurfaceObj leaf_surface = young*(1-age) + mature*age;
     produce DSurface(leaf_surface);
}
```

## 7.2 Gillespie groups

Gillespie groups are a special type 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* [7; 8]. 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 propensity (likelihood) of each reaction. For example, if the `Cell` module implements reactions:

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

then the production for the `Cell` module in the Gillespie group will 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 $\tau$ to the next reaction as $\tau = ln(1 - \chi)/p$, where $\chi$ is a uniform random number in [0,1] and $p$ is the sum of the propensities of all modules. To access $\tau$, call the function:

`float GillespieTime();`

There are two restrictions when using Gillespie groups:

- `ring L-system` statements are ignored, and
- new context is not supported.

## 7.3 Multi-view mode

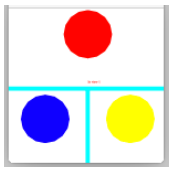
*lpfg* allows multiple views to be displayed simultaneously. The location of each view within the main window is defined in the view file (Section 8.1.1) using the `window` command. For example, to create three views, one at the top and two beneath it, the commands would be:

```
window: View1 0.0 0.0 1.0 0.5
window: View2 0.0 0.5 0.5 0.5
window: View3 0.5 0.5 0.5 0.5
```

where `View1, View2,` and `View3` are mapped to integer numbers and used in `UseView` and `vgroup` statements, and the four numbers are:

- the $x$ coordinate of the top left corner,
- the $y$ coordinate of the top left corner,
- the width, relative to the main window, and
- the height, relative to the main window.

The default border between the views is a black line, one pixel wide. This can be altered with the `window border` command (Section 8.1.1). Note that the view area is reduced by the size of the border. This is especially noticeable if a wide border is used.


See object: Multiview

To display these views, they must be activated with the `UseView` function (Section 6.4). The function is normally called within the `Start` statement (Section 4.1). For example:

```
Start: {
     UseView(View1);
     UseView(View2);
     UseView(View3);
}
```

If a `UseView` function is added or changed within the L-system file, *lpfg* must be restarted or New model selected from the pop-up menu. (Selecting New L-system does not suffice.)

The actual content of each view is defined in the `interpretation` section of the L-system using the `vgroup` command. For example, `View1` above would be defined as follows:

```
interpretation:
vgroup View1:
     ...
     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 the cursor ray with
                         // front plane, back plane, and halfway between them.
                         // Not affected by any keys.

    bool lbDown;         // Left button currently down

    bool lbPushed;       // Left button pressed
                         // since last call to GetMouseStatus

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

The cursor ray is the line passing through the cursor location, perpendicular to the screen, mapped into world space.

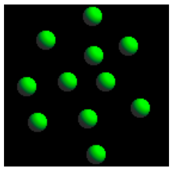
The left button values, `lbDown` and `lbPushed`, are only set when the left button is pushed with a combination of keys. These key combinations are also used to determine which mouse module is inserted when the left mouse button identifies a component of the model:

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

See object: SimpleDraw

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

```
derivation length: 1;

module Cursor();
MouseStatus ms; // mouse status

StartEach:   { ms = GetMouseStatus(); }

Axiom: SetColor(2) SetWidth(0.3) Cursor();

production:

Cursor() :
{
     if(ms.lbPushed)      // start a line
          produce MoveTo3d(ms.atMiddle) Cursor();
     if(ms.lbDown)        // continue drawing
          produce LineTo3d(ms.atMiddle) Cursor();
}
```


See object: MoveBalls

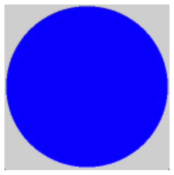
The following code draws a sphere when the left mouse button is pushed along with one of the key combinations for `MouseIns()`. Spheres can then be selected and moved.

```
derivation length: 1;

module AddBall();      // generates new balls
module Ball(V3d, int); // position, currencly selected?
MouseStatus ms;
int moving;            // Set to 1 after selecting a module
                       // and set to 0 after releasing the mouse button
Axiom: AddBall;

Start: { moving = 0; }

StartEach: { ms = GetMouseStatus(); }

/* Add sphere if mouse clicked outside any existing sphere. */

AddBall():
{
     if (!moving && ms.lbPushed) {
          moving = 1;
          produce Ball(ms.atMiddle, 1) AddBall();
     }
}

/* An existing ball has been selected. Adjust its position to
   that of the mouse. Set flag "moving" to 1 to prevent the
   addition of another ball at the same location. */

MouseIns() Ball(pos, selected) :
{
     moving = 1;
     produce Ball(ms.atMiddle, 1) ;
}

/* Move the selected ball as long as the mouse button
   is not released. */

Ball(pos, selected):
{
     if (selected) {
          moving = !ms.lbReleased;
          produce Ball(ms.atMiddle, moving);
     }
}

interpretation:

Ball(pos, selected) :
{
     produce MoveTo3d(pos) SetColor(1+selected) Sphere(1);
}
```

## 7.5 User-Defined Menus

In addition to the standard *lpfg* pop-up menu, a user-defined pop-up menu can be created and accessed by holding down the '4' key and using the left mouse button. The content of the menu is defined with multiple calls to `UserMenuItem()` functions, and each menu selection is identified using the `UserMenuChoice()` function.

Typically, the menu content is defined in the `Start` statement, and the menu item selected in the `StartEach` statement:


See object: UserMenu

```
int choice;

Start:
{
     UserMenuItem("First menu item",1);
     UserMenuItem("Second menu item",2);
     ...
}

StartEach:
{
     choice = UserMenuChoice();
}
```

# 8 *Lpfg*-specific input files

## 8.1 View file

Viewing and drawing parameters are stored in the view file, identified by its extension (*filename*.`v`)[4].

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.

### 8.1.1 Setting the view

| Command | Description | Default |
|---|---|---|
| `projection:` *pvalue* | Set the projection to `parallel` or `perspective`. | `parallel` |
| `scale:` *s*<br>`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` |
| `generate on view change:` *vchange* | Rerun the simulation. *Lpfg* rewinds to the axiom and performs the derivation again depending the value of *vchange*:<br>`on`: rerun every time the view changes through rotation, zoom, or pan.<br>`triggered`: rerun after the user releases the mouse button.<br>`off`: never rerun. | `off` |
| `view:` *id*<br>`dir:` *dx dy dz*<br>`up:` *ux uy uz*<br>`pan:` *px py pz*<br>`fov:` *val*<br>`shift:` *val*<br>`scale:` *val* | Define the view transformations to be used for view window *id*. In single view mode, set *id* to 0. All the transformation commands are optional.<br>`dir` and `up`: the view direction and up direction.<br>`pan`: the point that is the center of the view, relative to the center of the bounding box.<br>`fov`: the angle of the field of view in the *y* direction.<br>`shift`: the distance between the camera and the point being looked at.<br>`scale`: the scale of objects.<br>See Section 8.1.8 for a deprecated form of this command. | `dir: 0 0 -1`<br>`up: 0 1 0`<br>`pan: 0 0 0`<br>`fov: 45`<br>`shift: 1`<br>`scale: 0.9` |
| `box:` *id xmin xmax ymin ymax zmin zmax* | Define the default bounding box for view window *id*. In single view mode, set *id* to 0. | |

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

| Command | Description | Default |
|---|---|---|
| `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 mode. | |
| `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` |
| `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. | See below |

Default clipping planes are calculated on the basis of minimum and maximum distances to the bounding box in the view direction.

### 8.1.2 Rendering commands

| Command | Description | Default |
|---|---|---|
| `z buffer: on\|off` | Turn Z buffer `on` and `off`. | `off` |
| `render mode:` *rvalue* | Set the rendering mode to `filled, wireframe, shaded,` or `shadows`. Note that when the render mode is `shadows`, the Z buffer is set to `on` regardless of the value of the `z buffer` command above. | `filled` |
| `shadow map:` `size:` *n* `color:` *r g b* `offset:` *factor units* | Define parameters for shadow mapping when the `render mode` command is set to `shadows`.[5] 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. Shadows may not be displayed if values are too small ($n < 100$) or too large (dependent on graphics card). `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` |

[5]For details see https://registry.khronos.org/OpenGL-Refpages/gl4/html/glPolygonOffset.xhtml.

| Command | Description | Default |
|---|---|---|
| `light:`<br>`V:` *x y z*<br>`O:` *x y z*<br>`P:` *x y z e c*<br>`A:` *r g b*<br>`D:` *r g b*<br>`S:` *r g b*<br>`T:` *c l q* | Define a light source.[6] Include either:<br>`V`: vector of directional light source, or<br>`O`: origin of point light source, or<br>both `O` and `P` where `P`: spotlight with direction *(x,y,z)*, exponent *e*, and cutoff angle *c*.<br>In addition, the following options may be specified:<br>`A`: ambient color of light source.<br>`D`: diffuse color of light source.<br>`S`: specular color of light source.<br>`T`: constant *c*, linear *l*, and quadratic *q* attenuation factors (affecting point light sources and spotlights).<br>It is possible to define up to 8 light sources, one per line. | `V: 0 0 1`<br>(single light source in the default view direction)<br><br>`A: 1 1 1`<br>`D: 1 1 1`<br>`S: 1 1 1`<br>`T: 1 0 0` |
| `stationary lights: on\|off` | Enable stationary light sources when `on`, keeping the position of all light sources fixed. | `on` |
| `concave polygons: on\|off` | Enable the OpenGL tesselator when `on`, which divides polygons into triangles. This allows rendering of concave polygons, but *lpfg* may run slower. | `off` |
| `auto normals: on\|off` | Calculate normals to user-defined polygons generated with the `SP/EP` modules. When `on`, the normals are calculated as the cross-product of the first and second polygon edge. When `off`, the up vector of the turtle is used as the normal to the polygon. | `off` |
| `antialiasing:` *n* | Set the preferred number of samples per pixel in the anti-aliasing method. The parameter *n* is in the range 0 to 10. The final effect may depend on the hardware used. | `0` (no multi-sampling) |

### 8.1.3 Geometry presentation

| Command | Description | Default |
|---|---|---|
| `line style:` *lstyle* | Set the line style to `pixel, polygon` or `cylinder`. | `pixel` |
| `wireframe line width:` *width* | Set the line width when the `render mode` command is set to `wireframe`. The value of *width* must be greater than zero. | |
| `contour sides:` *n* | Set the number of sides *n* of polygonal approximations of circles, cylinder cross-sections, and sphere sections rendered in the *lpfg* window. This command can be overridden by either the `ContourSides` module (Section 5.8), or the `samples` parameter in the file created by the contour editor (see the *cuspy* tool in the ***vlab* Tools** manual). Values of $n < 3$ are clamped at 3. | 16 |

[6]For details see https://www.glprogramming.com/red/chapter05.html.

| Command | Description | Default |
|---|---|---|
| `capped cylinders: on\|off` | Cap generalized cylinders defined by a closed contour whenon, and generate a mesh consisting of one closed surface (a watertight mesh). If a concave contour is used to define the shape of the generalized cylinder, the command `concave polygons: on` should also be present. If an open contour is specified, this command should be set to `off`, otherwise an incorrect triangulation will be generated. | `no` |
| `backface culling: on\|off` | Specify that backward-facing polygons should not be drawn when `on`. This may speed up rendering or improve the rendering of transparent objects. | `off` |
| `corrected rotation: on\|off` | Use the corrected rotations when `on`. 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, set `corrected rotation` to `off`. | `on` |

### 8.1.4 Tropism commands

| Command | Description | Default |
|---|---|---|
| `tropism:`<br>`T:` *x y z*<br>`A:` *a*<br>`I:` *x*<br>`E:` *e*<br>`S:` *de* | Set tropism parameters. The tropism vector (`T`) is required. The remaining parameters are optional:<br>- `A:` angle (in degrees) that segments are trying to reach, with respect to the tropism vector<br>- `I:` (global) intensity of the tropism<br>- `E:` initial elasticity<br>- `S:` elasticity step | `A: 0`<br>`I: 1`<br>`E: 0`<br>`S: 0` |
| `torque:`<br>`T:` *x y z*<br>`I:` *x*<br>`E:` *e*<br>`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`<br>`E: 0`<br>`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 5.11.

### 8.1.5 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 the actual 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* `C:`*x y z* | Declare a predefined mesh in *filename* with optional scaling (`S:`), and contact point coordinates (`C:`). The mesh file must be in OBJ 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. RGB and PNG files are supported. | |

See Section 5.7 for surface and mesh modules.

### 8.1.6 Fonts

These commands define the font type to be used with the `Label` module (Section 5.10).

| **Command** | **Description** | **Default** |
|---|---|---|
| `font:` *Xfont* | Define the font type using the Xfont specification. | `-*-courier-bold-r-*-*-12-*-*-*-*-*-*-*` |
| `winfont:` *font size* `bi` | Define the font in Windows format: - *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` |

### 8.1.7 Postscript output control

*Lpfg* provides limited support for outputting models in postscript format. This feature is particularly useful when outputting 2D patterns. A key limitation in 3D is that hidden line/surface elimination is not implemented; graphical elements are output and painted (and overpainted) in the order in which they were generated and output to the postscript file.

| **Command** | **Description** | **Default** |
|---|---|---|
| `PS linecap:` *type* | Define whether line caps should be applied when exporting to postscript.<br>*type*=`0`: butt caps<br>*type*=`1`: round caps<br>*type*=`2`: projecting caps | `0` |
| `gradient:` *direction magnitude* | Define whether gradient shading should be applied to Beziér surfaces when exporting to postscript.<br>*direction*=`0`: gradient off<br>*direction*=`1`: gradient in the vertical ($y$) direction<br>*direction*=`2`: gradient in the horizontal ($x$) direction<br>*magnitude*: percentage change from near to far edge (where 1.0 represents 100%). May be positive or negative. | |

### 8.1.8 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*:<br>- *px py pz*: The center of the view (pan) relative to the center of the bounding box.<br>- *scale*: The scale of objects.<br>- *ux uy uz*: The up direction. | `view:` (Section 8.1.1) |

## 8.2 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*. In principle the units are one-hundredths of a second, but the actual interval may be affected by other factors (dependent on the operating system and the graphics card), especially at small *t* values. | `0` |
| `double buffer: on\|off` | Set double buffering `on` or `off`. | `on` |
| `clear between frames: on\|off` | Clear between frames when `on`. | `on` |
| `hcenter between frames: on\|off` | Horizontally center the model between frames when `on`. | `off` |
| `scale between frames: on\|off` | Scale the model to fit the view window between frames when `on`. | `off` |
| `new view between frames: on\|off` | Reset the view between frames when `on`. This command is most useful when using the `Camera()` module (Section 5.14) to dynamically position the camera. | `off` |
| `display on request: on\|off` | Display frames only on request. When `on`, only the first and last frame are displayed automatically. The `DisplayFrame()` function (Section 6.1.1) must be called to display intermediate frames. This makes it possible, for example, to skip frames that do not advance time but perform other calculations. When `off`, frames are displayed according to the `step` parameter. | `off` |

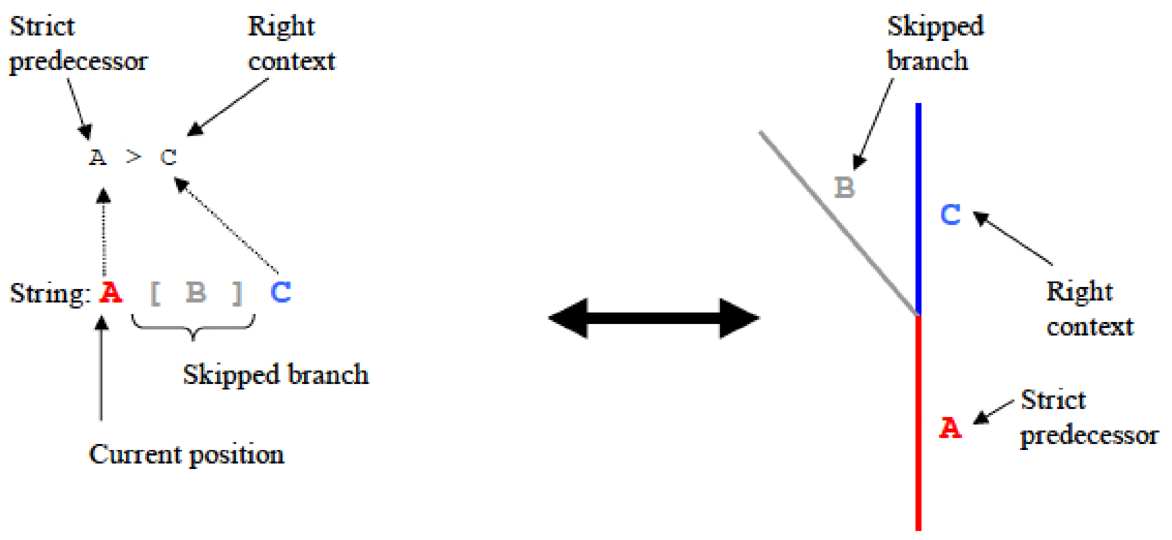


Figure 2: Matching right context. Lateral branches are implicitly ignored.

## 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 entire branch (up to and including the corresponding ] symbol) 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 (e.g. branch `B` in Figure 2).
- 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 3, module `C` is closer to `A` than `D`.
- If multiple lateral branches start at a given branching point, then the predecessor in Figure 3 will only check the first branch. To skip a branch it is necessary to specify explicitly which branch at the branching point should be tested (see Figure 5). This notation is a simple consequence of the rule presented in Figure 3. In the current L-system notation there is no shortcut to specify

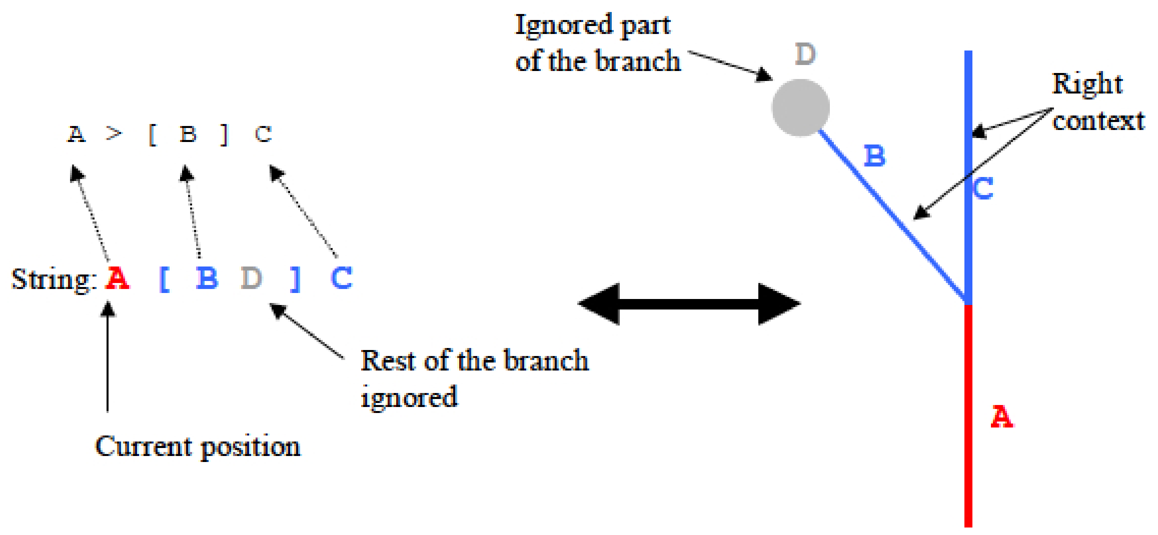


Figure 3: Matching right context. Remainder of lateral branch is implicitly ignored.

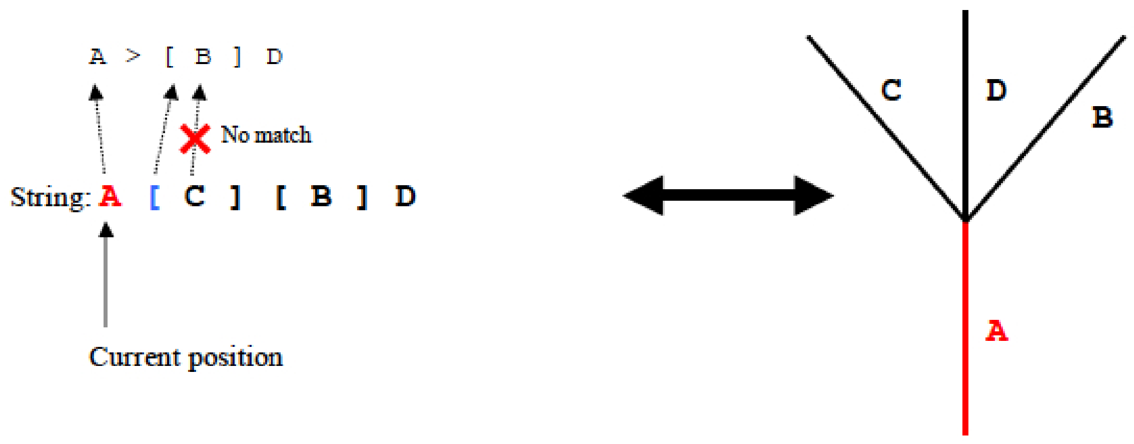


Figure 4: Problem with multiple lateral branches when matching the right context.

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 6).
- If the module in the string indicates the end of a branch (i.e. it is a ] module) then the entire branch (up to and including the corresponding [ symbol) is skipped (Figure 7).

The rule illustrated in Figure 6 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`.

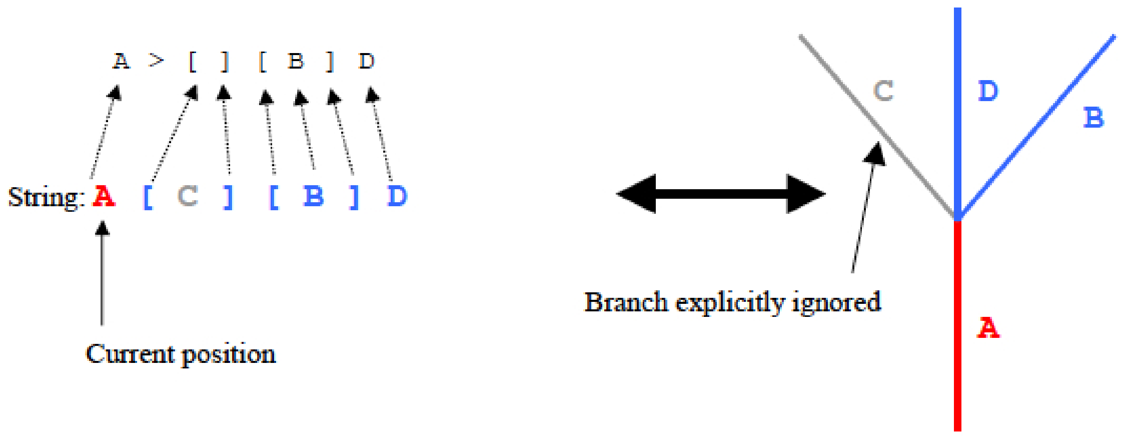


Figure 5: Explicit enumeration of lateral branches in the right context.

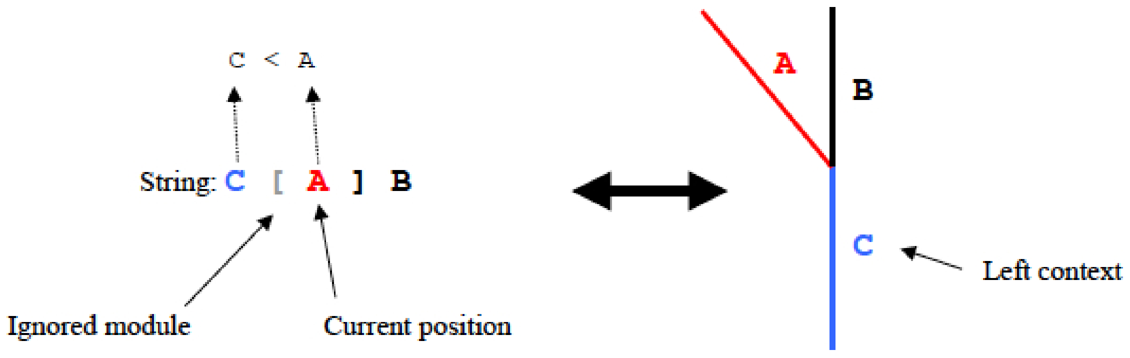


Figure 6: Matching left context. The beginning of the branch is implicitly ignored..

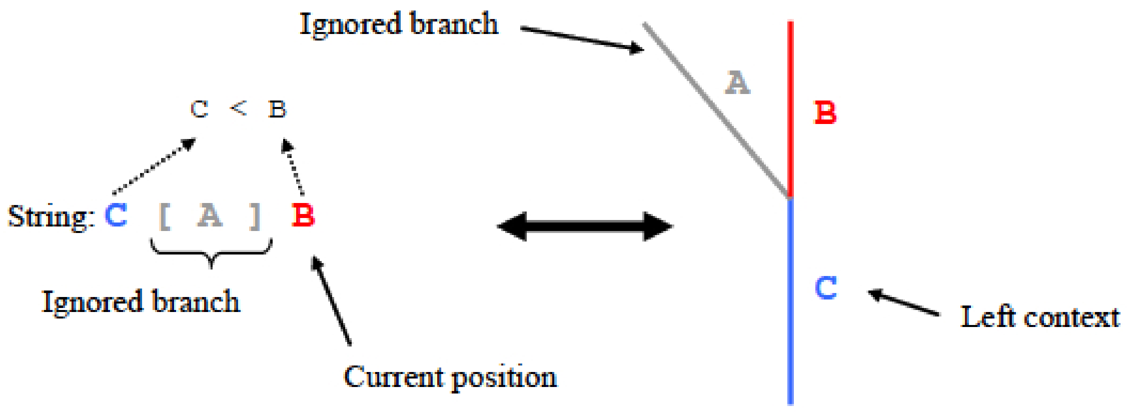


Figure 7: Matching left context. The lateral branches are implicitly ignored..

## 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 are not presently 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*, in the expectation that an editor for them will be available at some point.

#### 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 8.1.5). 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 6.2.3) and module (Section 5.7):

| Module | Description |
|---|---|
| `BsurfaceObjS GetSurface(int id)`<br>`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)`<br>`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 macOS or the newest tablet drivers. It consists of:

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

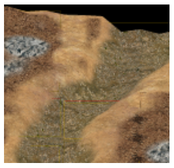
When `-tablet` is specified on the command line, tablet input is consider separately from mouse input, resulting in two separate input devices. With this option the tablet pointer is input using the `GetTabletStatus()` function, not the `GetMouseStatus()` function. However, the tablet cannot be used 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 them. `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

An earlier version of *lpfg* supported a multi-resolution terrain object, with the terrain defined using a specialized editor. Unfortunately, detailed documentation of the terrain feature (including the `.patch` file format) is not available, and the editor has not been maintained. Nevertheless, the *lpfg* functionality related to terrain has been kept to provide backward compatibility with existing models, and as a stepping stone for reviving and supporting this functionality in the future.


See object: TerrainDemo

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 6.3.1 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 the `GetCameraPosition()` function.

## 10.4 String verification

A mechanism was developed for automatic testing the *lpfg* derivation process. It is activated by including the following statement in the L-system file (at the same level as the `derivation length` or `consider` statements, for example):

`VerifyString:` *module list*;

This statement has effect only in the batch mode (see command line option `-b` in Section 1.1.1). If the statement is present in the L-system after deriving the string, *lpfg* will compare the contents of the derived string with the strings listed in *module list* (it compares only the module names, not parameter values). If the derived string matches, *lpfg* will print the message: `Verify: Success`. If the strings do not match, *lpfg* will print the message: `Verify: Fail`. It will also create two files containing the textual representation of the strings (module names only, no parameters) named:

```
Verify_ lsystemfile_expected.txt
Verify_ lsystemfile_actual.txt
```

where *lsystemfile* is the name of the L-system file specified in the *lpfg* command line.

## 11 Credits

The original specification of the L+C language (then named L) was developed jointly by Przemyslaw Prusinkiewicz, Radoslaw Karwowski, Jari Pettunen, and Risto Sievanen. On this basis, *lpfg* was designed and implemented by Radoslaw Karwowski, employing his idea of translating L+C to C++ and using an existing C++ compiler, rather than developing an L+C compiler from scratch [4; 5]. Further extensions have been introduced by Brendan Lane [6], Adam Kromm, Thomas Burt, Jason Kraft, Steve Longay, Adam Runions, Cordell Bloor, Pascal Ferraro, and Mikolaj Cieslak.

## 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<br>Brendan Lane |
| 2022 | Updated and re-formatted in LaTex | Lynn Mercer<br>Przemyslaw Prusinkiewicz<br>Pascal Ferraro<br>Mikolaj Cieslak |

## References

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

[2] Przemyslaw Prusinkiewicz. Art and science for life: Designing and growing virtual plants with L-systems. *Acta Horticulturae*, 630:15–28, 2004.

[3] Przemyslaw Prusinkiewicz, Mikolaj Cieslak, Pascal Ferraro, and Jim Hanan. Modeling plant development with L-systems. In RJ Morris, editor, *Mathematical Modelling in Plant Biology*, pages 139–169. Springer, 2018.

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

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

[6] 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.

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

[8] Mikolaj Cieslak and Przemyslaw Prusinkiewicz. Gillespie-Lindenmayer systems for stochastic simulation of morphogenesis. *in silico Plants*, 1(1):diz009, 2019.