L-System Explorer Quick Guide

Table of Contents

1. Intro

If you just can't remember what the module for yaw is, go to the Turtle Orientation section.

If you need a bit more info on L-System syntax, skip to that section, L-System Syntax

If you'd like to see some examples of L-Systems, select the **Examples** menu item on the main page and wander through the samples there, many of which have long comments.

2. Modules

A module is either a single character, or some system-defined sequences of characters. The behavior/meaning of the system-defined modules is described below. Note, that I've only tested US ASCII characters, others may or may not work. An L-system consists of the sequence of modules and the production rules that modify them: see L-System Syntax, below.

Why are they called modules and not commands or something else? Because module is an old term used by botanists for a unit of growth ("article" in French from the Latin for branch and translated to module in English.)

```
Modules marked u are unimplemented
Modules marked p are partially implemented
Modules marked c are changed from TABOP/cpfg
Modules marked n are new for this implementation
```

2.1. Turtle Motion

Without a parameter move 'step' amount, default 1 unit **Motion Modules**

F(d)	move d units w/pen down, records track point, creates geometry
f(d)	move d units w/pen up, records track point
G(d)	== F(d), but does not record track points, creates geometry
g(d)	== f(d), w/o recording track points
@[Mm](x,y,z)	goto position, 'M' draw, creates geometry, or 'm' don't draw

2.2. Turtle Orientation

The turtle maintains three unit vectors, heading (**H**), left (**L**), and up (**U**).
The turtle home orientation is **H** = **+x**, **L** = **+z**, and **U** = **+y**,
but L-system Explorer pitches up 90 degrees, initially, so **H** points **+y** and **L** points **-x**.

* positive yaw (left) rotates **H** towards **L** around **U**. (**x** -> **z** around **y**)
* positive pitch (down) rotates **U** towards **H** around **L**. (**y** -> **x** around **z**)
* positive roll rotates **L** towards **U** around **H**. (**z** -> **y** around **x**)

Without a parameter, a turn is 'delta' degrees, default is 90. Change delta
in an L-system with the line:

```
delta = DD /* where DD is your angle in degrees */
```

Note We make the turtle behave as if it were in a right-handed system, but all the underlying Babylonjs is left-handed (as is much of computer graphics). This has some odd consequences when comparing models with TABOP, and with, say, aeronautical usage.
If I'd known I could just tell Babylonjs to use a RHS, things would have been simpler.

Orientation modules

+(a)	yaw left 'a' degrees ('a' optional)
-(a)	yaw right 'a' degrees ('a' optional)
&(a)	pitch down 'a' degrees ('a' optional)
^(a)	pitch up 'a' degrees ('a' optional)
\(a)	roll left 'a' degrees ('a' optional)
/(a)	roll right 'a' degrees ('a' optional)
|	yaw 180 degrees

@v	roll so U is parallel to up [y]
@R(hx,hy,hz[,ux,uy,uz])	set heading to [hx,hy,hz] set U parallel to [ux,uy,z], if supplied
@H(h) **n**	set turtle to home position and orientation. If optional h is 0, then set H to +y, otherwise, H is +x. Does not draw or record point

2.3. Turtle Settings

Defaults without a parameter to one

;(i)	increase or set color/material index
,(i)	decrease or set color/material index
@;(n) **u**	increase or set back face color/material index
@,(n) **u**	decrease or set back face color/material index
#(n)	increase or set line width
!(n)	decrease or set line width
@Tx(n) **u**	set texture index
@D(s) **u**	set scale factor of subsequent geometry
@Di(f) **u**	multiply scale factor by f

2.4. Turtle Geometry

In addition to orientation and color/material, the turtle also maintains a *track shape*, which is the shape that it will extrude in moving when the pen is down. The default track shape is a circle with an initial size, as in diameter, of one.

@o(d)	circle of diameter d, centered at turtle position
@C(d) **u**	circle of diameter d in HL plane
@O(d)	sphere of diameter d
~(S,s) **c**	insert predefined surface S, with optional scale, s
@Cs(n[,t]) **n**	start contour with n final pts. n==0 => use just the control pts t == 0 => open contour (default), t == 1 => closed
@Ce(id) **n**	end contour and save it as id (number or string)
@Ca([t[,a]]) **n**	if t=0 (default), create arc between previous 3 pts if t=1, create arc from previous two points, w/ctr at p0, start at p1, with angle a

@Cc([n[,t]]) **n**	create Catmull-Rom spline from previous n points n == 0 (default) will use **all** prior generated points. t==0 => open (default) , else closed curve. This will add 1 + m * (n - (t==0) ? 1 : 0) points to the contour, where m = contour points per segment (default = 16)
@Ct(m1,m2) **n**	create Hermite spline from previous two points, with optional length multipliers m1 and m2
@Cb **n**	Create a cubic Bezier spline segment from prior four points
@Cm(m) **n**	create contour of multiplicity m, default 1, only if closed type
@Cn(n) **n**	Set the number of points per segment (default = 16). This is for inserted arcs or splines; control pts inserted with '.' or f, or g, are used as is. Total points is overridden by initial @Cs(n), if n != 0
@#(id)	set contour(id) as current track shape. The special id, 'default', are used as is. Total points is overridden by initial @Cs(n), if n != 0
{	start polygon
{(0) **c**	start path with stepwise control pts, not polygon
{(1)	start open path with Hermite spline control pts
{(2) **u**	start closed path with Hermite spline control pts
{(3) **u**	start open path with B-spline control pts
{(4) **u**	start closed path with B-spline control pts
}[(id)] **p**	end current polygon/path of any type and extrude contour, With parameter, id, save path as mesh
@Gs	start Hermite spline path, same as '{(1).', i.e. saves point
@Ge(n,id) **c**	close path started with @Gs. if n, set # intermediate points. If 'id', save extrusion as mesh
. ←-- that's a period	add current position to path, polygon, or contour
@Gt(m1,m2)	Set tangent length multipliers - default is 1.2
@Gr(a1,l1,a2,l2)	Set slope and length of tangent vectors for radius curve
@Gc(n) **p**	Store control point. Optionally set number of interpolated strips

2.5. L-system Geometry and Control

[	start branch, pushing state onto stack

]	end branch, popping branch state stack
% **p**	cut modules to end of branch
$(id,scale) **c**	push current Lsystem and use sub-Lsystem id
$	end current sub-Lsystem, return to previous

3. L-System Syntax

3.1. Rule-based systems

An L-System is a rule-based object defined on a set of symbols; formal grammars are available in the literature, e.g. Hanan thesis. Here, we'll be far less strict, and attempt to describe how to create one.

In outline, you begin with an axiom, which is a sequence of symbols called 'modules', and provide a set of rules, called, productions, which modify the axiom based which production matches the module currently encountered. Some strict L-systems can be evaluated in parallel, but this implementation cannot do that. We evaluate strictly from left to right, with some caveats. Note that context dependent rules do not strictly rule out parallel evaluation, but variables, both global and local, as well as some semantics such as % (= cut), make it difficult and, in some cases, impossible to avoid nonsensical results.

Evaluation proceeds in steps: the entire input string/axiom is completely rewritten before the next step. When all modules have been evaluated and substitutions made, the result string becomes the input for the next evaluation step.

3.1.1. Modules

A module is one of

- a single, case-sensitive character,
- a single, case-sensitive character with a parenthesised argument list, i.e. it looks like a function call,
- a system-defined sequence starting with @, which may contain more than one character.

Modules with arguments are called parametric or parameterized modules. In the case of context matching (see below), a parameterized module with the same character as a non-parameterized module will not match. The arguments are typically numeric, but in some system-provided modules they may be strings, or, potentially, any valid JavaScript object. Examples:

- aAa(1)A(1) is an L-System string with four different modules: a, A, a(1), A(1)
- b('leaf') is a user module with a string parameter
- C(1, '2', {x: 2}) is a user module with three parameters, a number, a string and a JavaScript object (although, there is no way to use an object, currently)

The system-defined modules implement branching and the turtle interpretation of the L-system. Their behavior can't be overridden, but the matching and production rules apply equally to them. Many system-defined modules are multi- character sequences starting with @, such as @#(id). Perhaps, I'll add a user-defined multi-character module, say, @Uxyz(…).

The arguments of parametric modules in the axiom are constant values. In the predecessor they are dummy variables, and in the successor, they may be constants, variables, or expressions which are evaluated before being substituted into the result string as constant values. The actual value of the parameter is substituted into the dummy variable at match time.

3.1.2. Axiom

This is the starting sequence of modules, and can be any valid sequence. The L-system is evaluated in discreet steps, where every module is evaluated in each step. So, you can think of the axiom as the starting point of each iteration, but we will refer to it as the result 'string'. If you define a derivation length, we will do that many iterations, otherwise the result is just the axiom. Declare the axiom like this:

```
axiom: <modules>
```

e.g. an axiom with three modules:

```
axiom: CA(10)B
```

3.1.3. Productions

A production has two main parts, the predecessor and the successor, which most simply is:

```
P --> S
```

where P is a single module and S is a sequence of one or more modules which will replace P in the axiom.

Productions can have an optional tag at the beginning with the format, ‘pd: ‘, where ‘d’ is a number:

```
p1: P → S
```

The default rule, if no production matches, is to move the module to the result string, i.e.:

```
P --> P
```

As a special case, if S is "*", then, P will be removed from the result string.

The predecessor can be more complicated though, and more formally is:

```
[left-context < ] strict-predecessor [ > right-context] [ : condition ]
```

Things in brackets (which are not part of the syntax!) may be omitted. In addition, the condition may be more complex, see below.

Finally, note the header line after parsing, which tells you exactly what the parser saw:

```
rules:= {pre, strict, post} {cond} {succ} {scope}
```

The {scope} term just tells you what variables, if any, are in scope for each production, and is only there for information.

3.1.4. Context-sensitive Matching

Context-sensitive matching is optional, and is based on what precedes or follows a module, or both, reading the string from left to right. The left context is separated from the strict predecessor by '<', and the right context by the '>' characters. Example:

```
abc < d > efg    which matches d in the string:  ...abcdef...
```

Branches introduce an apparent non-locality in matching; consider the following predecessor and strings:

```
predecessor:   a < b > c

string 1:     aaa[bc]ddd    *matches*
string 2:     aaa[b]cddd    *does NOT match*
string 3:     [aaa]bcddd    *does NOT match*
```

The way to think about this is that the right bracket,], denotes the end of a branch, which, topologically, is not next to whatever follows it in the string representation of the tree. The left bracket, [, starts a branch, so the immediate right neighbors of the last 'a' are 'b' and 'd' in string 1. For string 1, the predecessor, **a > d**, would also match.

It is possible to control which modules should be used in matching with the directives, **include:** and **exclude:**. For example,

```
exclude: FG
```

Tells the matcher to ignore and skip, the modules F or G if they are encountered when looking for a match, e.g.:

```
a < b , given the string aFGFGb, would produce a match.
```

Conversely, it is possible to consider only certain modules by using the include directive:

```
include: ab
```

then, only a and b are considered when matching, so, "a < b" matches any string where "a" precedes "b" with any intervening modules, except “b” and respecting the branching rules, above.

3.1.5. Matching Order

Rules are matched in top-down order, with the caveat that the most-specific match which occurs first will be used. This means the rule which has the longest context. Example, given the following rules:

```
p1: a --> aa
p2: b(i) < a --> c
p3: bb(i) < a : i < 10 --> d
p4: bb(i) < a : i < 5 --> c

and the string, bbb(4)a, the result will be:

bbb(4)d
```

because p3 is more specific, i.e. it has a longer context, than p1 or p2, and has the same specificity as p4, but it comes first.

3.1.6. Conditional Tests

The simplest condition is a test, which is an expression that returns true or false and follows mathjs syntax and expression rules. For example:

```
A(i) : i<10 --> F(i)A(i+1)
```

If the string were BCA(1), it would be expanded to BCF(1)A(2).

However, since a condition is

```
[pre-condition] test [post-condition]
```

where both pre- and post-condition have the format {expression} or are empty. Again, an expression is any valid mathjs expression that can use dummy variables in the current production, or local or global variables. The pre-condition is evaluated before the test, so you can compute values needed for the test. The post-condition is evaluated after both the pre-condition and the test and can be used to compute substitutions in the successor after a successful test. It's possible to use the post-condition without either pre-condition or test by inserting a '*' or 1 for the test:

```
v=0
axiom: bbb(4)a
bb(i) < a : * {v=i} -> d(v)
```

In the UI, the parse of this system is shown as:

```
axiom = bbb(4)a
rules:= {pre, strict, post} {cond} {succ} {scope}
{b,b(i),a,}{,true,v=i}{d(v)}{has scope}
```

Here, the condition is shown as {,true,v=i}, where the pre-condition is empty, test is true, and the post-condition is v=i.

Evaluating the L-system results in:

```
bbb(4)d(4)
```

3.2. Decomposition and Homomorphism

These are supported and documented here. Examples coming soon.

3.3. Pre-parsing

Before parsing, the L-system text is run through a JavaScript cpp-like parser which handles comments and macro defines. After that, all empty lines are removed and the system is parsed. See cpp.js README for details on how this differs from standard CPP. It works for basic pre-processing, but don't try to get fancy with macros with parameters.

3.3.1. Comments

C-style comments, '/* ...stuff..., including newlines... */', are supported.

Single-line '//' style comments are not supported because they conflict with L-system module syntax.

3.3.2. Defines

Lines that start with '#define macro value', 'macro' being some word, create standard C-style macros where 'macro' is replaced with 'value' in the L-system before parsing starts. They are not as powerful as cpp.

3.4. Settings

Before the axiom is specified, settings variables can be set for the L-system. The following settings can be used to control default/initial L-system values, using standard syntax, i.e. var = value. Multiple settings on the same line must be separated by a semi-colon. **Note** that the view setting is a JS object with the properties 'auto', 'position' and 'target'. The latter two have values that are JS arrays of X,Y,Z coordinates.

stemsize	width of extrusion. default: 0.1
delta	angle in degrees of yaw, pitch, and roll. default: 90
step	distance traveled by F,f,G,g modules. default: 1
view	**position:** 3D position of viewer/camera. **target:** 3D position of view/camera target. e.g. view = {position: [20,20,5], target: [0,8,0]} — or --- **auto:** direction where direction is one of 'X', '-X', 'Y', '-Y', 'Z', '-Z', or a direction array defining where the camera position should be. The target is always the center of the bounding sphere of the drawn geometry and the distance from the target is about twice the radius of the bounding sphere. e.g. view = {auto: 'y'} will look down on the XY plane. Note, case-insensitivity and quotes. — or --- view = {auto: [1,1,1]} will place the camera on a line through the target center parallel to the vector 1,1,1, looking at the target.

The default view is {auto: 'X'}

3.5. Variables

Variables used in the L-system can be set and used in the rules. If you define and set a variable before the lsystem: keyword (which is not required), the variable will be global across the main L-system and any sub-L-systems. If the variable is set after the lsystem: keyword, it is local to that L-system. Module parameter variables are local to the rule they are used in. However, since global and L-system local variables can be used in rule expressions along with module parameters, it's best not to have name conflicts. For example if you have a parameterized module like, A(t), then defining a global or L-system scope variable, t, may cause you grief. LS Explorer uses a dynamic scoping mechanism where it looks for variables first in rule scope, then L-system scope, and, finally, global scope.

3.6. Expressions

Expressions occur in tests, parameters, and pre/post test; they follow mathjs syntax: Expression syntax for mathjs.

Of particular note, are the logical operators which must be written explicitly as

and	instead of **'&&'**
or	instead of **'||'**
not	instead of **'!'**

for example:

```
(t > 0) and not u, instead of  (t > 0) && !u
```

3.7. Syntax overview

Generally, you should try for something close to this, in order from top down:

1. A section of #defines at the top, which will be expanded before parsing.
2. Global variable statements, including view, stemsize, delta, ….
3. The lsystem: tag statement, not strictly required unless you use sub-L-systems.
4. The derivation length: steps, or n = steps, or dlength = steps statement.

   Then any L-system local variable definitions.
5. axiom: <your axiom here>
6. A set of productions, optionally with
7. Optionally, the keyword, decomposition, followed by decomposition productions. Optionally, with the maximum depth: n statement.
8. Optionally, the keyword, homomorphism, followed by homomorphism productions. Optionally, with the maximum depth: n statement.
9. The keyword, endlsystem, not required unless sub-L-systems follow.
10. Optionally, the keyword lsystem: x, followed by the previous followers of lsystem. Note that the derivation length for sub-L-systems is ignored.

4. UI and controls

- subject to constant change -

4.1. L-system controls

Here, you can load an L-system file from disk or enter one manually in the text area. Below the choose file entry are controls to build, step, parse, rewrite, and draw the L-system.

- **Choose File** allows you to load an lsystem locally. It is a text file, typically with a .ls extension
- **Save LS** allows you to save the L-system text file.
- **Build** will clear geometry, reset the turtle, re-parse, and rewrite the L-system before drawing.
- **Step** will do one iteration step of the L-system, i.e. one rewrite step and one draw, no matter how many iterations are specified in the L-system spec. As a special case, if the **Parse** button is used to reparse and recreate the L-system, **Step** will initially write and interpret just the axiom; subsequent steps will clear the previous geometry, rewrite one step of the L-system, and redraw it.
- **Parse** this button will parse whatever text is in the L-system source area, re-create the current L-system and show the result in the L-system Expansion text box. If you make changes to the L-system, choose this first.
- **Rewrite** will rewrite the parsed L-system, and, again, place the result in the L-system Expansion box.
- **Draw** will interpret the expanded system and draw the geometry on the canvas. Note, that **Draw** neither clears, nor resets any previously drawn geometry.
- **Gen Code** causes the interpretation to generate the turtle code that it uses to draw the geometry. This is simpler in single turtle mode. This should run standalone (with the Turtle3d class) to generate the geometry
- **Save LS file** allows you to save the text of the L-system description as a .ls file.
- **Save Model** allows you to save drawn geometry to a .babylon or GLTF/GLB file.
- **MultiTurtle?** is a checkbox to turn this mode on/off. When on, the interpretation/drawing creates a new turtle for each branch and then gives each turtle one step on its branch in a round-robin draw mode. When a turtle reaches the end of the branch it is destroyed. This mode typically appears more natural, however there are several operations, including TABOP nested polygon productions, that do not work in this mode, so it's best to turn it off if things look wonky.

Below the L-system source box are status and more controls:

- **L-system status:** |X|Y|Z| , where X is the number of iterations/expansions of the axiom; Y is the number of modules in the L-system expansion, and Z is the number of modules that have been interpreted/drawn. This last box will turn green when drawing is complete.
- **Draw Speed** is an input to control the drawing speed in modules/frame. It defaults to 200 and runs from 1 to 500. Higher rates tend to bog down the browser.
- **Save Code** allows you to save that generated code as a GLB or BABYLON file.

4.2. Turtle controls

Below the L-system controls are the turtle controls. To the right of the label, "Turtle Controls" is a widget that will expand and collapse the controls. The first expansion is a buttonbar that controls the default turtle. If you open the JavaScript console, you can directly enter Turtle3d commands to get an idea for how it works. You can also turn on the Gen Code mode to get examples of how the turtle is used.

- **Hide/Show** will hide or show the turtle shape, which is a mini axis of the HLU system of the turtle. Note that the size of the turtle axes is 0.5 units.
- **Home** moves the turtle back to 0,0,0 and orients it along the axes.
- **Reset** is the same as **Clear** and **Home**
- **Clear** will clear all the geometry generated by the turtle(s).
- **Look at Turtle** orients the camera so the turtle is in the center of view, or you can pick the origin,

or the center of the drawn geometry bounding sphere.

- **Show/Hide Color Table** displays the current color/material table

4.3. Scene Controls

Here you can toggle visibility of the Ground, Sky, coordinate axes, and gridded planes on the primary axes. The latter can help to debug size issues. You can also 'look at' the turtle, the current origin for the turtle, or the center of the mesh bounding sphere: helpful if you've lost track of where you are when navigating around.
