vlab Environmental Programs
Reference Manual

Last updated: November 30, 2021

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

Contents

1 Introduction

For interaction between plants and their environment, both the *cpfg* and *lpfg* modeling programs include predefined *communication modules* that send and receive information to/from an external process simulating environmental factors. When in *environmental mode*, the modeling program:

- Spawns an external environmental program, and sets up the communication link between the two processes.
- Performs an environmental step after each derivation step, that determines the state of the turtle associated with each communication module in the string.
- Sends information to the environmental program as each communication module is encountered:
  - The address (position in the string) of the module.
  - The parameter values associated with the module.
  - The state of the turtle associated with the module (optional).
  - The type and parameters of the module following the communications module (optional).
- Receives results from the environmental process for each communication module. This includes:
  - The address of the communication module that should receive the results.
  - The parameters to be returned.

1.1 Components of the system

The functionality within *cpfg* and *lpfg* is complemented by the following external components:

- Purpose-built environmental programs, with an optional file of parameters. (See Section 3 for the environmental programs included in *vlab*, and Section 4 for creating new environmental programs).
- A communication specification file, *filename*`.e`, that defines how to run the environmental program, what information should be sent, and how (Section 2).
- A set of "to" and "from" files, if the communication is through files (Section 2).

1.2 Usage in *cpfg*

Environmental mode is set in *cpfg* using the `-e` command line option, which also defines the communication file, *filename*`.e`. For example:

```
cpfg -m plant.map -e enviro.e plant.l plant.v plant.a
```

See object:
CPFG-Sierpinski

The communication module, `?E`, is used within *cpfg*. It can have as many parameters as required, where each parameter is a floating point number. For example, to send a value to the environmental program, and take a different action depending on whether it returns 0 or 1:

```
axiom: ?E(VAL)

?E(c) : c==0 --> ...
?E(c) : c==1 --> ...
```

The parameters are used to both send and receive information, and do not necessarily have the same meaning in both directions. The number of parameters should equal the maximum needed in either direction. For example, to send a value to the environment and produce results depending on two returned parameters:

```
axiom: ?E(VAL,0)

?E(a,b) : a < 10 --> X(a)
?E(a,b) : a >= 10 --> X(a-b)
```

1.3 Usage in *lpfg*

Environmental mode is set in *lpfg* by the presence of an environmental file (with a `.e` extension) on the command line. For example:

```
lpfg plant.map enviro.e plant.l plant.v plant.a
```

Note that, unlike *cpfg*, no command line option is required: the file is recognized by its extension.

See object: LPFG-Sierpinski

Parameters are used to both send and receive information. However, modules in *lpfg* must have a fixed number of parameters. Therefore, there are two predefined communication modules, `E1` and `E2`, for sending/receiving one and two floating point numbers respectively. For example, to send a single value and receive a 0 or 1 in return:

```
axiom: E1(VAL)

E1(c):
{
     if (c==0) produce ...;
     if (c==1) produce ...;
}
```

In addition, module `EA20` has a single parameter that is an array of floating point numbers. The array can be defined using the `EA20Array` type. For example, to send a value to the environment and produce a module with different values depending on the first four parameters returned:

```
EA20Array ea = {VAL,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
module A(float,float);

axiom: EA20(ea);

EA20(x):
{
     if (x[0] < 10)
          produce A(x[1],x[3]) ;
     else
          produce A(x[2],x[3]) ;
}
```

See object: LPFG-Sierpinski -EA20

There is also a command line option associated with this array, `-cleanEA20`, that zeroes the array before the next iteration. This is useful if the environmental program returns an arbitrary number of values.

1.4 Flow of information

An environmental program has one of two possible modes of operation in each step:

- Immediate answer - Results depend on local properties of the environment, not on information related to other communication modules within the string. A response can be sent to the modeling program immediately. This mode is suitable for simulations of static environments.
- Delayed answer - Results depend on information related to other communication modules within the string, due to the propagation of information through the environment. The information from all communication modules in the string is first sent to the environmental program (and stored in internal data structures). Computations are then performed based on all the information, and results are returned to the modeling program for each module. This tends to be the most common mode: it is used for all the *vlab* environmental programs.

The communication follows a standard set of steps:

1. The modeling program reads the communication specification file, establishes the data structures necessary for communication, starts the environmental process, and waits for confirmation from it.
2. The environmental program reads the communication specification file, connects itself to the data streams, sends a confirmation, and waits for the first transmission.
3. The modeling program receives the confirmation and starts the simulation.
4. After each derivation step, the modeling program performs an environmental step to process the communication modules. The information from each communication module is sent to the environmental program, with a final reserved end-of-transmission message after the last module.
5. The environmental program receives the information, and either sends an immediate answer for each communication module, or waits until it receives the end-of-transmission message and then send responses for all the modules. It terminates the transmission with a similar end-of-transmission message, and waits for the next transmission.
6. The modeling program receives the data coming from the environment and sets the parameters of the specified communication module accordingly. When the end-of-transmission message is encountered, the environmental step is complete and the program continues, returning to Step 4.

2 The communication specification file

The communication specification file, used by both *cpfg* and *lpfg* to set up the environmental program, is specified on the command line (Sections 1.2 and 1.3 respectively). It contains the following commands:

Command	Description
`executable:` *commandline*	The complete command line required to spawn the environmental program. This command is mandatory.
`communication type:` *type*	The data transfer mechanism between the modeling program and the environmental program, where *type* can be one of `files` (see below), `memory`, or `pipes`. This command is mandatory.
`turtle position:` *format* `turtle heading:` *format* `turtle left:` *format* `turtle up:` *format* `turtle line width:` *format* `turtle scale factor:` *format*	Optional turtle parameters to be sent to the environment program, and the associated C-like format string for each. See examples below.
`following module:` *flag*	If *flag*=`yes`, the module following the communication module is sent. The default is `no`.
`interpreted modules:` *list*	The list of modules that can follow a communication module. See the *QuasiMC* environmental program (Section 3.2.4). This feature works in *cpfg* only.
`verbose:` *flag*	Verbose mode is switched `on` or `off`. The default is `off`.

There are always two data streams: one for sending information to the environmental program, and another for receiving information from it. With the command `communication type: files`, the modeling program creates two files:

`.to_field`*nnnn*`.0`
`.from_field`*nnnn*`.0`

where *nnnn* is a unique number (the modeling program's process ID). Both are text files located on the lab table that can be accessed for debugging.

To distinguish the values representing the turtle state in the above files, the *format* can include a name, and can also limit the values to a specified number of decimal places. For example:

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

will send the x and y coordinates of the turtle position and heading vectors, preceded by `P:` and `H:` respectively. The letters are not mandatory.

3 Environmental programs in *vlab*

The following environmental programs are included in the *vlab* distribution. To create additional environmental programs see Section 4.

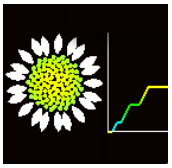
3.1 Collisions

3.1.1 Vigor - 2D (*honda81*)

This program tests for collisions between disks of a fixed radius. If two disks collide, the one with the lower *vigor* value is set to 0. This has been used to test for collisions between leaf clusters, for example. It can also be used to test for point collisions, by assuming a small radius (e.g. 0.01).

The command line for the program is:

`honda81 [-e` *commfile*`.e]` *eparamsfile*



See object: LPFG-Fibonacci Florets

The communication specification file, *commfile*`.e`, used by the modeling program should include the following command:

```
turtle position:  %g %g %g
```

The *honda81* parameter file, *eparamsfile*, contains the following:

Parameter	Description
`radius:` *r*	The radius of each disk. All disks have the same radius; they differ only in vigor.
`3d case:` *flag*	If `on`, each object is represented as a sphere and collision tests are performed in three dimensions. The default is `off`.
`verbose:` *flag*	Switch verbose mode `on`. The default is `off`.

The communications module can have one or two parameters, p_1 and p_2. On input the parameters are:

p_1 = vigor of the object, represented by a number between 0 and 1.
p_2 = the index of a group. Collisions will be tested only between objects in the same group. The default is 0.

On output the parameters are:

p_1 = 0 if the object collides with another object with more vigor, and 1 otherwise.
p_2 unchanged.

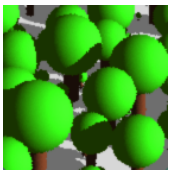
The environment stores all queries corresponding to communication modules in a linked list. After all queries are inserted, the program computes the distance from a given object to other objects having a higher or equal vigor. The object is tested only with objects in the same group. The response is 0 (lower vigor) or 1 (equal or higher vigor). If the vigor of a object is 1, it will always stop the growth of other objects.

3.1.2 Vigor - 2D or 3D (*ecosystem*)

This program tests for collisions between a set of objects. Internally, the objects are represented as disks in 2D (or spheres in 3D) with a given radius. If two disks partially overlap, the one with the lower radius (or the lower vigor, if using) is reported as colliding. This program was originally used to simulate plants growing in a field.

The command line for the program is:

`ecosystem [-e` *commfile*`.e]` *eparamsfile*



See object: CPFG-ecosystem

The communication specification file, *commfile*`.e`, used by the modeling program should include the following command:

```
turtle position:  %g %g %g
```

The *ecosystem* parameter file, *eparamsfile*, contains the following:

Parameter	Description
`grid size:` *x y z*	The size of a regular grid used for reducing the time necessary to determine the colliding disks. The *z* value is optional; it is for use only with `3D case: on`.
`vigor:` *flag*	If `on`, an additional parameter is sent with each communication module, specifying the vigor of the plant. If two disks/spheres collide, the one with the lower vigor is reported as colliding, even if its radius is larger. The default is `off`.
`3d case:` *flag*	If `on`, each object is represented as a sphere and collision tests are performed in three dimensions. The default is `off`.
`verbose:` *flag*	Switch verbose mode `on`. The default is `off`.

The communications module can have one or two parameters, p_1 and p_2. On input the parameters are:

p_1 = radius of the disk/sphere representing the object.
p_2 = vigor of the plant. Used only when `vigor:` is set to `on`.

On output the parameters are:

$p_1 = 0$ if the object collides with another object of bigger radius or vigor, and 1 otherwise.
p_2 unchanged.

Each object is represented by a communication module with one parameter if the vigor is not used, or two parameters if the vigor is used. All modules in a given simulation step are stored in a linked list. After all modules are processed, a regular grid of `grid size` is created so that it tightly encompasses all disks (or spheres) representing the objects. The grid is used to speed up the collision tests and is rebuilt after each step. Each voxel of the grid contains a linked list of the disks (spheres) occupying a portion of the voxel. For each disk, a check is done to see whether it intersects with any other disk stored in the same voxels.

If a collision is found and `vigor` is `off`, the program returns 0 if the radius of the given disk is less than or equal to the radius of the colliding disk. Otherwise, it returns 1.

If a collision is found and `vigor` is `on`, the program returns 0 if the vigor of the given disk is less than the vigor of the colliding disk. If both vigors are equal, but the radius of the given disk is less than or equal to the radius of the colliding disk, the program also returns 0. Otherwise, it returns 1.

3.1.3 Climbing (*arvo*)

This program is used to simulate climbing around surfaces. For a given segment, it determines whether the segment collides with a surface and, if it does, the program computes a new orientation for the segment such that the collision is avoided and the segment's tip keeps a given distance from the surface. The algorithm is based on a paper by Arvo and Kirk [1].

The command line for the program is:

`arvo [-e` *commfile*`.e]` *eparamsfile*

The communication specification file, *commfile*`.e`, used by the modeling program should include the following commands:

```
turtle position:  %g %g %g
turtle heading:   %g %g %g
turtle up:        %g %g %g
following module: yes
```

The *arvo* parameter file, *eparamsfile*, contains the following:

Parameter	Description
`domain size:` *x y z*	The range (in world coordinates) of a regular grid used to store objects for the intersection test.
`position:` *x y z*	The position of the lower front left corner of the grid.
`grid size:` *x y z*	The size of the grid in voxels.
`surface distance:` *d*	The distance (a real value) from the surface that all segment endpoints will try to maintain.
`max surface distance:` it d	The maximum distance between the surface and a segment endpoint. This is usually about 2 × `surface distance`.
`tries for Q:` *n*	The number of randomly generated candidates for selecting a new position `Q` for the segment's tip.
`tries for surface:` *n*	The number of tries (rays) traced in order to find the closest surface.
`obstacles:` *filename*	A file containing obstacles. See the background file description in the *cpfg* manual for the format of this file.
`add objects:` *flag*	If *flag* = `on`, objects specified by the symbol following the communication module `?E` are added to the grid (and removed at the end of each simulation step). The default is `off`. See below for the recognized modules.
`remove objects:` *flag*	If *flag* = `on`, objects are removed from the grid at the beginning of each simulation step. The default is `on`.
`seed:` *n*	The seed (an integer value) for the random number generator.
`verbose:` *flag*	Switches verbose mode `on` or `off`. The default is `off`.

The *x y z* parameters can also be delimited by commas (,) or 'x'.

The `add objects` parameter recognizes the following module types (located directly after the communications module) when set `on`:

Module	Description
`S(`*r*`)`	A sphere with radius *r*.
`C(`*r*`,`*h*`)`	A cylinder with radius *r*, and height *h*.
`C(`*r1*`,`*r2*`,`*h*`)`	A cone with base radius *r1*, top radius *r2*, and height *h*.

The communication module will have a differing numbers of parameters (p_i) depending on the action to be taken:

# of pmtrs	Description	Input parameters	Output parameters
0	Add the module following the communication module to the grid.		
1 or 4	Check for intersection of the segment with an object. The communication module MUST be followed by a segment module: `F`, `f`, `G`, or `g`.	p_1-p_4 ignored	$p_1 = 1$ if intersection, 0 otherwise p_2-p_4 = surface normal at intersection point (optional).
7	Get a new segment length, and new heading and up vectors.	p_1 = desired segment length p_2-p_7 ignored	p_1-p_3 = new heading vector (of unit length). p_3 = length of segment. 0 if segment not found. p_5-p_7 = new up vector (the normal of the closest point on the surface).

The program stores all incoming queries in a dynamically allocated array. Queries are answered after the string is processed. For each point P a new point Q with the desired length (parameter p_1 in the 7 parameter case) is calculated, beginning on a line perpendicular to the up vector, and sweeping from the heading direction +/- 180°.

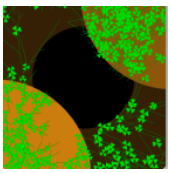
For each trial point Q, the closest surface is found (a number of rays are shot seeking the closest intersection). If no intersection is found, a new point Q is generated. Otherwise, the new endpoint P is taken, specified as the intersection of the trial ray with the closest surface plus the normal vector of the intersection times the desired minimum distance from the surface. The program then returns the new heading vector (P minus the `turtle position`, T), its length $|P - T|$, and the up vector (the surface normal).

NOTE: In the case of 1 or 4 parameters, when only segment intersection is tested, the `surface distance` parameter in *eparamsfile* influences the returned intersection. This can be used to keep the object a small distance away from the surface to account for its width. The size of a voxel must be bigger than this `surface distance` parameter.

3.2 Light

3.2.1 Direct light – 2D (*clover*)

This program determines the amount of direct light coming from the top and reaching the apices of plants (represented as points). The light can be obstructed by leaves, stored as disks in a high-resolution grid.



See object: CPFG-Clover

The command line for the program is:

`clover [-e` *commfile*`.e]` *eparamsfile*

The communication specification file, *commfile*`.e`, used by the modeling program should include the following command:

```
turtle position:  %g %g %g
```

The *clover* parameter file, *eparamsfile*, contains the following:

Parameter	Description
`grid range:` x y	The range (in world coordinates) of a regular 2-dimensional grid used to store the leaves.
`grid position:` x y	The position of the lower front left corner of the grid.
`grid size:` x y	The size of the grid in voxels.
`transmittance:` t	Transmittance of leaves. A value in the range [0,1].
`input image:` *imagefile*	The name of an image file used to specify the light intensities coming from the top. Only the green channel of the image is used.
`remove old leaves:` *flag*	If set to `yes`, all leaves are removed from the grid after each simulation step. The default is `no`.
`z is up:` *flag*	If set to `no`, the y coordinate is considered as up, and the program uses the x and z coordinates of each point. The default is `yes`, with the x and y coordinates used to specify the location of leaves and queries.
`verbose:` *flag*	Switch verbose mode `on`. The default is `off`.

The communication module can have either one or two parameters, depending on the action required:

# of pmtrs	Description	Input parameters	Output parameters
1	Calculate the light intensity reaching the point.	p_1 ignored	p_1 = light intensity
2	Perform one of the following actions: 0 - query the light reaching the point 1 - add a leaf 2 - remove a leaf	p_1 = type of action (0, 1, or 2) p_2 = leaf area for actions 1 and 2; ignored for action 0	p_1 = light intensity when action = 0; otherwise not changed p_2 not changed

Note that the communication module with one parameter is the same as the module with two parameters where the input values are $p_1 = p_2 = 0$.

The program stores communication modules querying the light in a linked list. A high-resolution grid (usually 2000×2000) is used to store leaf information if the module action is to add or remove a leaf. Each voxel of the grid contains information about the number of leaves obstructing it, where each leaf is represented as a disk with the specified area. Thus for a given leaf, the values in all voxels which are covered by the disk representing it are incremented by one. Similarly, if a leaf is removed

from the grid, the values in the corresponding voxels are decreased by one.

In addition to the number of leaves obstructing the voxel, the voxel also contains the initial intensity of light. This intensity defaults to 1, but can also be specified with an image file (using `input image` in the parameter file). In that case, the green channel of the image specifies the intensity at each voxel. Note that the resolution of the grid need not match the resolution of the input image.

After the grid is updated, all queries stored in the linked list, are processed. For each query, a corresponding voxel is determined. The initial intensity associated with the voxel is multiplied by a factor t^n, where t is the `transmittance` value and n is the number of leaves obstructing the voxel. The resulting intensity is returned in the communication module.

3.2.2 Direct light – 3D (*chiba*)

This program determines the amount of direct light reaching spheres. It is based on a paper by Chiba *et. al.* [2]. The spheres can be seen as an approximation of objects such as leaf clusters.

The command line for the program is:

`chiba [-e` *commfile*`.e]` *eparamsfile*

The communication specification file, *commfile*`.e`, used by the modeling program should include the following command:

```
turtle position:  %g %g %g
```

The *chiba* parameter file, *eparamsfile*, contains the following:

Parameter	Description
`grid size:` *x y z*	The size of the grid in voxels. (The grid range in world coordinates is determined according to the location of the spheres so that the grid tightly encloses them.)
`number of samples:` *s*	The number of samples for the light sphere. The closest number higher than $8 \times k^4$ is used, where k is an integer. Generally 128 works well.
`lower to upper ratio:` *u*	The ratio of the lower and upper hemisphere intensities. The number should be in the range 0.0 to 1.0. The default is 0.7.
`use CIE formula:` *flag*	If `no`, all light sources in the same hemisphere have the same intensity. If `yes`, the light intensity of sources in the upper hemisphere is determined by the standard CIE formula for an overcast sky [3] (based on the direction towards the light source). The default is `no`.
`source direction:` *x y z intensity*	A light source with the given *intensity*. This command takes precedence over the previous three commands. After all sources are defined, their intensities are normalized such that their sum is 1. The direction is also normalized.
`transmittance:` *t*	The transmittance of the spheres. This is a floating point number between 0 and 1. The default is 0.6.
`radius:` *r*	The default radius for spheres whose radius is not explicitly given. The default is 25.
`beam radius:` *r*	The radius of the beam of rays from each sphere. The radius is expressed as a fraction of the current sphere's radius (i.e. r is a number between 0 and 1). The default is 0.
`estimate intersection area:` *flag*	If `on`, the intersection function projects the sphere onto a plane perpendicular to the ray. See the explanation below. The default is `off`.
`verbose:` *flag*	Switch verbose mode `on`. The default is `off`.

The communication module can have one or four parameters. On input, only the first parameter is used: p_1 = the radius of the sphere. If $p_1 = 0$, the value of `radius` from the parameter file is used.

On output, the parameters are:

p_1 = percentage of light perceived by the centre of the sphere (a number between 0 and 1)
p_2 - p_4 = the brightest direction (of unit length)

The program stores the incoming communication modules in a dynamically allocated array. It then build a regular grid (generally $64 \times 64 \times 64$) to speed up ray casting.

The amount of incoming light is computed for each module by shooting rays from the center of each sphere. In the case of a sky hemisphere, the number of rays is set to the `number of samples` parameter. When a fixed number of light sources is specified, the number of rays is set to this number. Each intersected sphere reduces the perceived light intensity.

If the communication module has 4 parameters, the brightest direction is calculated as the sum of all sample rays multiplied by their intensities.

When `estimate intersection area` is set to `off`, the ray is tested against a sphere with radius equal to the intersected sphere radius, r, plus the beam radius to obtain the length, l, of the line segment inside the sphere. The intensity associated with the ray is multiplied (reduced) by $pow(t, l/2 \times r)$, where t is the `transmittance` value. Unfortunately, this formula is not ideal since it does not account for the beam radius which, in itself, is a fixed value that may not work well with variable-sized spheres.

A better approach, when the spheres vary widely in size, is to set `estimate intersection area` to `on`. The function will then project the spheres onto a plane perpendicular to the ray, and calculate the intersection area of disks on the plane. It will return a ratio of this area to the area of the sphere from whose center the ray is traced.

3.2.3 Direct light – 3D (*takenaka*)

This program determines the amount of direct light reaching spheres, reduced by a *maintenance cost*. It is based on a paper by Takenaka [4]. Like *chiba*, the spheres can be seen an approximation of objects such as leaf clusters.

The command line for the program is:

`takenaka [-e` *commfile*`.e]` *eparamsfile*

The communication specification file, *commfile*`.e`, used by the modeling program should include the following command:

```
turtle position:  %g %g %g
```

The *takenaka* parameter file, *eparamsfile*, contains the following:

Parameter	Description
`grid size:` *x y z*	The size of the grid in voxels. (The grid range in world coordinates is determined by the location of the spheres so that the grid tightly encloses them.)
`parameter:` *s*	The sparsity of matter (e.g. leaf distribution) on the sphere. If the radius of the sphere without any matter is r, then the radius of the entire sphere is $r \cdot s$. The default is 1.5.
`transmittance:` *t*	The transmittance of the spheres. This is a floating point number between 0 and 1. The default is 0.1.
`efficiency:` *e*	A multiplicative parameter influencing the resulting light product. The default is 0.015.
`source:` *x y z intensity*	A light source with the specified *intensity*. After all sources are defined, their intensities are normalized such that their sum is 1.
`beam radius:` *r*	The radius of the beam of rays from each sphere. The radius is expressed as a fraction of the current sphere's radius (i.e. r is a number between 0 and 1). The default is 0.
`verbose:` *flag*	Switch verbose mode `on`. The default is `off`.

The communication module has four parameters. On input, the parameters are:

p_1 = the sphere area
p_2 = the maintenance cost

On output, the parameters are:

p_1 = the sphere area (unchanged)
p_2 = the product of photosynthesis, after the maintenance cost is subtracted

The program stores all incoming communication modules in a dynamically allocated array. It then builds a grid to speed up ray casting.

The amount of incoming light for each sphere is computed by shooting a beam of rays from the sphere centre towards each light source. Each intersected sphere reduces the perceived light intensity. The weight of the final product is computed according to the values in the appendix to the paper.

3.2.4 Objects in a scene (*QuasiMC*)

The program computes light distribution within a scene, such as a plant canopy. It is based on the the *MonteCarlo* program (Section 5.3) but uses a randomized quasi-Monte Carlo (RQMC) method rather than a Monte Carlo method. RQMC estimation has been shown to reduce the error and computational effort of Monte Carlo estimation for plant-light simulations [5]. See the QuasiMC User's Manual for more detailed information on this program.

The command line for the program is:

`QuasiMC [-e` *commfile*`.e]` *eparamsfile*

The communication specification file, *commfile*`.e`, used by the modeling program should include the following command:

```
turtle position:  %g %g %g
turtle heading:  %g %g %g
turtle up:  %g %g %g
turtle head:  %g %g %g
following module:  yes
interpreted modules: M1 M2 ...
```

where M_1, M_2 ... are user-defined modules that may follow the communications module.

The *QuasiMC* parameter file, *eparamsfile*, can be divided into several sections:

Pre-tracing commands

Parameter	Description
`return type:` $k_1, k_2, ..., k_s$	The return type for each of s wavelengths (see `spectrum samples`): `F` = absorbed flux (default) `D` = absorbed flux density `U` = incident irradiance of upper surface `L` = incident irradiance of lower surface `H` = number of intersections per ray.
`grid size:` x y z	The size of the grid in voxels. Used to speed up intersection testing. The default is 1×1.
`remove objects:` *flag*	If `yes`, all objects are removed after each simulation step. This is the default. If set to `no`, the modeling program should not send modules that have already been sent.
`number of runs:` n	The number of randomizations to perform on the QMC point set. This value should be ≥ 5 for a good estimate of the variance between objects. The default is 1.
`sampling method:` *name*	The name of the RQCM sampling method to be used: `monte carlo` - fast, high variance, number of rays not restricted `sobol` - slow, low variance, number of rays not restricted `korobov` - fast, low variance, number of rays restricted `poly-korobov` - fast, low variance, number of rays very restricted
`number of rays:` n	The number of rays to trace through the scene, where n is an integer greater than 1. For `korobov` and `poly-korobov` methods, the value n must be a power of 2 such that $128 \leq n \leq 1048576$ or $1024 \leq n \leq 65536$, respectively.
`surface:` *filename sdiv*	The name of a surface file, and the number of subdivisions, *sdiv*, to perform on the Bézier patches that describe the surface. There may be several `surface` commands that will be assigned an *id* sequentially starting with 0.

Parameter	Description
`verbose:` *level*	The level of output, used for debugging: `0` = no output `1` = output some scene information `2` = level `1`, plus the debug window, with objects shaded according to `return type` (the default) `3` = level `2`, plus the path each ray takes.
`visualization:` *lflag bflag wflag* r_b g_b b_b r_s g_s b_s	The content of the debug window when `verbose` is set to `2` or `3`. *lflag* = show the light source(s) *bflag* = show the bounding box *wflag* = show in wireframe mode r_b g_b b_b = RGB values of the background color r_s g_s b_s = RGB values of a shaded object The default is `yes no no 0 0 0 1 1 1`
`return variance:` *flag*	If `yes`, the individual object variance is written to a file. See the file naming convention below. The default is `no`.
`cylinder sides:` n	The level of detail used to generate a triangulated cylinder (representing an internode). The value must be in the range $3 \leq n \leq 128$. For smooth connections between cylinders, use a power of 2. The default is 4.

When `return variance` is set to `yes`, a file is created using the following naming convention:

<`sampling method`><*simulation step*> . <`number of runs`>.<`0` or `1`>

The last extension is `1` if `one ray per spectrum` is set to `yes` (see below). Otherwise, it is `0`.

Generating rays

Parameter	Description
`light source:` *x y z weight*	A directional light source approximating a very distant light such as the sun. Several light sources can be specified, by including several of these commands. Their weights should sum to 1. The default light source is (0.0, -1.0, 0.0) with weight = 1.0.
`spectrum samples:` s	The number of wavelengths for which light distribution should be computed. This command must precede the `source spectrum` command, and all commands defining materials. The default is 1, and the maximum is 20.
`source spectrum:` λ_1 w_1 λ_2 w_2 ... λ_s w_s	The size of the wavelength (λ) and weight (w) for each of the s wavelengths. The current implementation does not use the λ values, but they must be specified. The weight can be thought of as the initial energy of the ray.
`one ray per spectrum:` *flag*	If `yes` (the default), only one ray is used to carry information for all wavelengths. The main advantage of this is speed and variance reduction. Otherwise, if `no`, a separate ray is used for each wavelength.
`rays from objects:` *flag*	If `yes` (the default), rays will be generated from virtual sensors that are represented by a rhombus (the `P(`a, b`)` module). For any other module, the absorbed irradiance will not be returned. The default is `no`.

Tracing rays

Parameter	Description
`maximum depth:` n	The limit on the number of hits a ray can make before it is terminated. If the value is set to -1, there is no limit on the depth and the `Russian roulette` command must be specified. The default is 5.
`Russian roulette:` t p	The parameters of the Russian roulette method for terminating rays. If a ray's radiant energy drops below the threshold t, it is terminated with probability p. Otherwise, the energy of the ray is increased to make up for rays that have been terminated. The values must satisfy $0 \leq t$ and $p < 1$. The defaults are $t = p = 0$.
`ignore direct light:` *flag*	If `yes`, light hitting objects directly from a light source is ignored. This is useful for observing the effect of indirect light only. The default is `no`.

Materials

Parameter	Description
`local light model:` *type*	The type of BRDF/BTDF used in the light simulation can be either `modified Phong` or `Lambertian`.
`leaf material (top)` : r_1 n_{r_1} t_1 n_{t_1} n_1 : r_2 n_{r_1} t_2 n_{t_1} n_2 ⋮ : r_s n_{r_s} t_s n_{t_s} n_s	The properties of the default material. Five values are specified for each of s wavelengths (`spectrum samples`). The properties are: r: reflectance, in the range [0,1] n_r: reflectance scattering exponent, in the range [0,∞] t: transmittance, in the range [0,1] n_t: transmittance scattering exponent, in the range [0,∞] n: refractive index (not used) where $0 \leq r + t \leq 1$. This command has not default value. It must be specified.
`leaf material (bottom)` : r_1 n_{r_1} t_1 n_{t_1} n_1 : r_2 n_{r_1} t_2 n_{t_1} n_2 ⋮ : r_s n_{r_s} t_s n_{t_s} n_s	The default material associated with the bottom side of an object. This command should be specified even if the values are the same as `leaf material (top)`.
`material` : r_1 n_{r_1} t_1 n_{t_1} n_1 : r_2 n_{r_1} t_2 n_{t_1} n_2 ⋮ : r_s n_{r_s} t_s n_{t_s} n_s	Additional materials can be specified using this command. The materials are identified by their order, starting with 2 (`leaf material (top)` is index 0 and `leaf material (bottom)` is index 1).
`material parameter:` p	The parameter in the module that is the material index. This command only applies to modules other than `T` or `P`.

Sky model

Parameter	Description
`location:` x y	The geographic location to use for the simulation, where x is the latitude, and y is the longitude. The range of values for each parameter is: -90 ≤ x ≤ 90, where values below zero are in the southern hemisphere, and -180 ≤ y ≤ 180, where values below zero are in the west.

Parameter	Description
`weather:` c t	A simple description of the weather over the growth period, where c represents clouds, and t is the turbidity in the atmosphere. The values are: $c = 0$: an overcast day $c = 1$: a clear day $t = 1$: pure air $t = 4$: clear $t = 64$: fog The defaults are $c = 1$ and $t = 2.45$.
`growth period:` h_s h_e	The time period over a day for which sampling occurs, where h_s is the starting hour and h_e is the ending hour. Both values must be between 0 and 23.
`julian day:` d	The day of the year that the sky is sampled. The value must be between 1 and 365.

On input The program always receives the communication module and the following module. The communication module has s parameters, one for each wavelength, where s is defined by `spectrum samples`.

The following module can be one of:

Module	Description
`T(`a, b, mt, mb`)`	An isosceles triangle, where b is the height in the direction of the turtle heading vector, and the length of the edge perpendicular to the heading vector is $2a$. The parameters mt and mb are the index to the material for the top and bottom of the object. They are optional: the default materials, `leaf material (top)` and `leaf material (bottom)` will be used if they are not included.
`U(`l_1, l_2, h, β`)`	A triangle, where h is the height in the direction of the turtle heading vector, β is the angle between the heading vector and the base of the triangle, and l_1 and l_2 are the lengths of the left and right sides respectively, of the base intersected by the heading vector.
`P(`a, b, mt, mb`)`	A rhombus with a diagonal in the direction of the heading vector of length b, and a perpendicular diagonal of length $2a$. The parameters mt and mb are the index to the material for the top and bottom of the object. They are optional: the default materials, `leaf material (top)` and `leaf material (bottom)` will be used if they are not included.
`S(`$id, scale$`)`	A predefined Bézier surface identified by id, that is uniformly scaled by the factor *scale*. The id is associated with the sequential list of files specified by the `surface` command in *eparamfile*. Note this id may not be the same as the surface id specified in the modeling program's view file.
`I(`r, l`)`	An internode, where r is its radius and l is its length.
`Q(`n, x, y, z`)`	Override the `number of rays` (n) and `grid size` (x, y, z) specified in *eparamfile* in the next light simulation step. Useful for changing the accuracy of path tracing as a simulation proceeds and the number of objects increases or decreases.
`J(`d, c, t`)`	Override the `julian day` and `weather` specified in *eparamfile* in the next light simulation step, where d is the day of the year, and c and t are defined as in the `weather` command.

Module	Description
<user-defined>	A module with a corresponding homomorphism in *cpfg* that gives the vertices of a polygon representing an object. The interpretation symbol must be specified using the `interpreted modules` command in *commfile*`.e`. The parameter specifying its material properties is identified using the `material parameter` command.

On output The s parameters of the communication module are set to the absorbed/incident radiant energy of the object for each wavelength. The values are normalized such that the flux density of a flat surface perpendicular to the direction of the incoming light is $1W \cdot m^{-2}$.

4 Creating new environmental programs

New environmental programs can be created using the communication library, *comm*. It is included in both the modeling program and the environmental process to facilitate the exchange of information between the model and the environment.

4.1 Data structures

There are two data structures used in the communication library: one to hold the turtle state information, and a second to hold the parameters of the communication module and/or the module immediately following it.

The turtle state information is received in the structure `CTURTLE`. It contains each turtle parameter together with the number of actual values received. Thus if a particular turtle parameter is not listed in the communication specification file (Section 2), its corresponding count (*name*`C`), is set to 0.

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

The module parameters are stored in a `module_type` structure:

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

Both structures are defined in the library header file, `comm_lib.h`.

4.2 Library functions

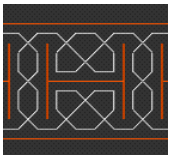
Both modes The following functions are used in both Immediate and Delayed mode (Section 1.4).

void **CSInitialize***(int *argc, char ***argv)*
Initialize the communication and parse the options. This should be the first call in the function *main*(). The parameters are the same as function *main*(), specifying the number of command line options *(argc)* and an array for storing these options *(argv)*. Since the communication library may add more options to the command line, the function may update the values of *argc* and *argv*.

void **CTerminate***(void)*
End the communication. This should be the last call in the function *main*().

Immediate Answer mode If the parameters of a communication module can be modified immediately, the following function is used:

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


See object: CPFG-Celtic Knot

When **CSMainLoop()** is used, the communication is fully controlled by the modeling program. This function only returns when it receives a message to terminate the program. At that point, local data structures can be cleared, and the **CTerminate()** function called. See the example in Section 4.3.1.

Delayed Answer mode When the incoming communication module cannot be answered immediately, the following functions must be called in this specific order:

int **CSBeginTransmission***(void)*
Start transmission (of all communication modules in the string). The process waits for the modeling program to perform a simulation step and to send the first communication module. The function always returns a value of 1.

int **CSGetData***(int *master, unsigned long *module_id, Cmodule_type *two_modules, CTURTLE *turtle)*
Obtain a communication module and possibly the following module (if the second module is not present, its name is an empty string, i.e. *two_modules*[1].*symbol*[0] is equal to 0). The parameter *module_id* is a unique identification number for the communication module, the pointer *two_modules* points to a two-dimensional array containing the communication module and the following module, and the pointer *turtle* points to the turtle structure (note that only some of the turtle parameters may be sent, depending on the specification file). The parameter *master* can be ignored.
The function returns 0 when there is no other module (at the end of the environmental pass). In this case, *module_id* is set to the number of the current simulation step.

int **CSGetString***(int *master, char *str, int length)*
Read a string *str*, with maximum length *length*. The *master* parameter can be ignored. If defined in the communication specification file, selected modules can be interpreted during an environmental step and the polygons representing the modules (or their homomorphic image) sent as a set of strings following the communication module. This function is used in a loop after each call to **CSGetData**() to retrieve any strings that may have been sent. It is recommended to always include a loop of

calls to this function (see examples below) since any strings not read will interrupt the communication.

The function returns 0 when there are no more strings.

void **CSSendData***(int master, unsigned long module_id,Cmodule_type *comm_module)*
Return the modified communication module to the modeling program. The original *module_id* must be specified. The parameter *master* should always be 0.

int **CSEndTransmission***(void)*
End transmission (after all modified communication modules have been returned). The function returns 1 when the process is requested to terminate. In this case, the communication loop should be exited, the process should free its data structures, and call **CTerminate**() above.

The Delayed Answer functions should be used in a `MainLoop()` function, with the following general form.

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

     for (;;) {
          CSBeginTransmission();
          while (CSGetData(&master, &module_id, two_modules, &turtle)) {
               StoreQuery(module_id, two_modules, &turtle)
                   /* store queries, with their module_id*/
              while (CSGetString(&master, str, sizeof(str))) {
                   ProcessGraphics(str);
                        /* process the graphical representation of the
                           module following the communication module */
               }
          }
          DetermineResponse(); /* determine the answers*/
          SendBackResponse();
                /* return modified communication modules using
                   CSSendData(master, module_id, &two_modules[0]); */

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

where `StoreQuery()`, `ProcessGraphics()`, `DetermineResponse()`, and `SendBackResponse()` are defined based on the data structures chosen for storing and processing the incoming communication modules.

4.3 Example environmental programs

4.3.1 Immediate answer environmental program

This example illustrates Immediate Answer mode, where the response to each communication module can be returned immediately. Thus the program uses the `CSMainLoop()` function.

In this example, the `Answer()` function determines the distance from the current turtle position to the point (0,0,0). Therefore, the communication specification file, *commfile*`.c`, should include the command:

```
turtle position: %g %g %g
```

If the turtle position is greater than the first parameter of the communication module, the parameter is set to 1. Otherwise it is set to 0.

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

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

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

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

4.3.2 Delayed answer environmental program

This example illustrates Delayed Answer mode, when incoming communication modules must be stored before determining the return parameters for all modules. Each incoming module is stored in a one-dimensional array of fixed size. This program detects collisions between modules, based on the turtle position of each. Therefore, the communication specification file, *commfile*`.c`, should include the command:

```
turtle position: %g %g %g
```

The coordinates of the each point are compared with the coordinates of all other points. If there is another point with the same coordinates, the program returns the value 0. Otherwise, there is no reply.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "comm_lib.h"
#define EPSILON 0.001 /* precision of comparisons */
#define MAXQUERIES 1000 /* maximum number of queries */

struct item_type {
     float position[2];
     float query;
     unsigned long id;
     int master;
} queries[MAXQUERIES]; /* queries */
int num_queries; /* actual number of stored queries */

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

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

     comm_symbol.num_params = 1;
     comm_symbol.params[0].set = 1;
     comm_symbol.params[0].value = 0; /* report only collisions */
     for (i=0; i< num_queries; i++) /* for all queries */
          if(queries[i].query) { /* don?t answer if no parameter */
               for (j=0; j<num_queries; j++)
                   if (i!=j)
```

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

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

     /* infinite loop - until message ?exit? comes */
     for(;;) {
          CSBeginTransmission();
          num_queries = 0;
          while (CSGetData(&master,&module_id,two_modules,&turtle))
               StoreQuery(master, module_id, two_modules, &turtle);
          DetermineResponse();

          /* EndTransmission returns 1 when the process is
                  requested to exit */
          if (CSEndTransmission())
               break;
     }
}

/*************************************************************/
int main(int argc, char **argv)
{
     /* initialize the communication as the very first thing */
     CSInitialize(&argc, &argv);
     MainLoop();
     CTerminate(); /* should be the last function called */
     return 1;
}
```

This environmental program will create a Sierpinski triangle (Figure 1) with a *cpfg* model that contains the following axiom and production:

```
axiom: ?E(0)
?E(c) : c==0 --> [F?E(0)]-(60)/(180)[F?E(0)]
```

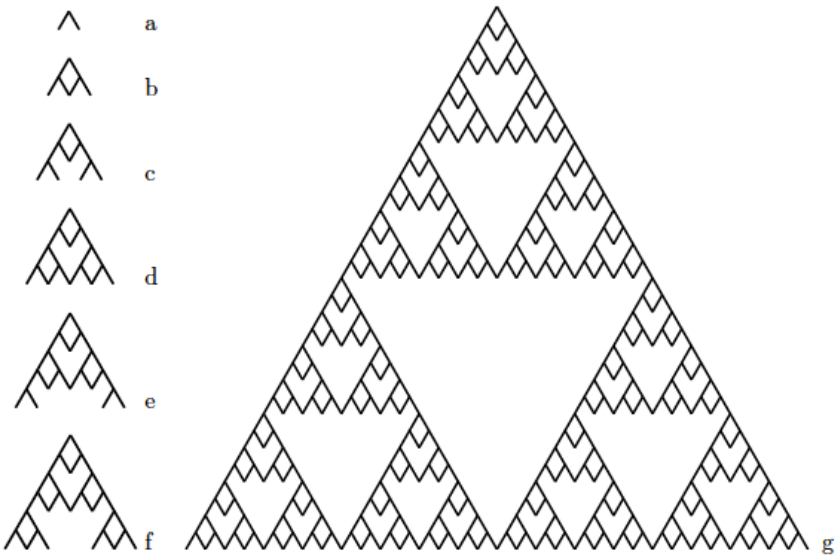


Figure 1: Sierpinski triangle generated after 1, 2, 3, 4, 5, 6, and 32 steps.

The end point of a segment is represented by module `?E` with one parameter. This parameter is initialized to 0 and, if the point collides with another point, the environment sets it to 1. If the point does not collide, the parameter remains 0, and the production creates two new branch segments.

5 Deprecated features

5.1 Point collisions (*ulam*)

This program has been replaced by *honda81* (Section 3.1.1). It determined whether a point in a given set of two-dimensional points occupies the same place as other points – i.e. the program determines collisions in a discrete grid of points.

The command line for the program is:

`ulam [-e` *commfile*`.e]`

The communication specification file, *commfile*`.e`, used by the modeling program should include the following command:

```
turtle position:  %g %g %g
```

Note that this program does not have a parameter file.

The communications module has a single parameter that is ignored on input. On output the parameter has the value 0, if there is a collision, and 1 if not.

The program stores all queries, representing two dimensional points in a linked list. After all points are inserted, it computes the distance from a given point to each other point. The response is 0 if there is a collision, and 1 if not.

5.2 Forces (*collisions*)

This program has not been ported to the Mac iOS. It tested for collisions between balls of a given radius. If two balls collide, the program returned the force the colliding ball initiates. The program was used in simulations of dynamic systems such as a cluster of cherries.

The command line for the program is:

`collisions [-e` *commfile*`.e]` *eparamsfile*

The communication specification file, *commfile*`.e`, used by the modeling program should include the following commands:

```
turtle position:  %g %g %g
```

The *collisions* parameter file, *eparamsfile*, contains only two parameters:

Parameter	Description
`radius:` r	The default radius of a ball.
`verbose:` *flag*	Switch verbose mode `on`. The default is `off`.

The communications module can have two or three parameters, p_1, p_2, and p_3. On input the parameters are:

p_1 = the radius of the ball. If 0, the value of `radius` is used.
p_2 - p_3 are ignored.

On output the parameters are the force acting on the colliding ball.

The program stores all balls corresponding to communication modules in a linked list. After all queries are inserted, the program computes the distance from a given ball to all other balls. If there is a colliding ball, the force acting on the tested ball t is computed as:

$$\vec{F}_t = \sum_{i \in B} 2 \cdot (c_t - c_i) \left(\frac{|c_t - c_i|}{r_t + r_i} - 1 \right)$$

where B is the set of balls colliding with ball t, and c and r denotes the center and radius of a ball, respectively.

5.3 *MonteCarlo* ENVIRONMENT PROGRAM

This program was developed to determine the amount of light reaching objects in a scene, using a path tracing algorithm based on Monte Carlo techniques. It has been superceded by *QuasiMC* (Section 3.2.4). See [6] for more details on this program.

As light reaches an object (in the form of rays), the program determines whether the light is absorbed, reflected, or transmitted through the object, based on the surface parameters associated with the object.

The command line for the program is:

`MonteCarlo [-e` *commfile*`.e]` *eparamsfile*

The communication specification file, *commfile*`.e`, should include the following command:

```
turtle position:  %g %g %g
turtle heading:   %g %g %g
turtle up:        %g %g %g
turtle head:      %g %g %g
following module: yes
```

The *MonteCarlo* parameter file, *eparamsfile*, can be divided into several sections:

Specifying the grid

Parameter	Description
`domain size:` *x y z*	The range (in world coordinates) of a regular grid used to store objects to speed up the intersection test.
`position:` *x y z*	The position of the lower front left corner of the grid.
`grid size:` *x y z*	The size of the grid in voxels.
`remove objects:` *flag*	If `off` or `no`, the objects are not removed from the grid after each simulation step. The default is `on` or `yes`.
`obstacles:` *filename*	A file containing additional objects to be added to the grid (see the background file description in the *cpfg* manual). There can be only one `obstacles` command.

Controlling generation of initial rays

Parameter	Description
`light source:` *x y z weight*	A light source. Several light sources can be specified by including several of these commands. The initial rays are generated based on the *weight* associated with each source.
`sky file:` *filename*	The file defining the intensities coming from all directions of a sky hemisphere. The file contains $n \times m$ numbers defining the intensity of the sky from different directions specified by two angles, θ and ψ, where $\theta \in [0, \pi/2]$ and $\psi \in [-\pi, \pi]$. This command takes precedence over the `light source` command. If neither command is specified, all initial rays come from the top.
`ray density:` *n*	The number of initial rays generated per unit area. The value of n should be changed if the scene is scaled up or down.

Parameter	Description
`stratified sampling:` *flag*	If `on` or `yes`, and neither `light source` or `sky file` are specified, the rays coming from the top will be generated using stratified sampling, which provides a better distribution of rays.
`spectrum samples:` s	The number of wavelengths for which light distribution should be computed. This command must precede the `source spectrum` command, and all commands defining materials.
`source spectrum:` λ_1 w_1 λ_2 w_2 ... λ_s w_s	The wavelengths (currently not used) and weight for each of the s spectrum samples. The weight distribution is the same for all light sources.
`one ray per spectrum:` *flag*	If `no` (the default), light intensities reaching the objects for different wavelengths are computed separately, one set of initial rays for each wavelength. If `yes`, each ray carries the information about all wavelengths and the light intensities are determined after one set of initial rays is traced. See below for more details.
`rays from objects:` *flag*	If `no` (the default), rays are initiated from the light sources, or from the sky hemisphere. If `yes`, rays are traced backwards, from the objects towards the light source. This can be more efficient for a small number of objects. See below for more details.

Tracing a ray

Parameter	Description
`periodic canopy:` *flag*	If `on` or `yes`, the rays which leave the scene are transformed to the opposite side of the scene and tracing continues. This simulates an infinite canopy with periodically repeating sets of objects. The default is `off`/`no`.
`maximum depth:` d	The number of hits (intersection of a ray with an object) for each ray. If set to -1, there is no limit. The default is 3. Note that it is better to set this limit high (e.g. 10) and use the `Russian roulette` command.
`Russian roulette:` t p	An optional method for determining the termination of rays. After each hit, the ray's intensity is compared with the threshold intensity t. If it is below t, the ray is terminated with probability p. The intensity of rays which are not terminated is increased by $1/(1-p)$. Both t and p are numbers between 0 and 1.
`reflectance model:` *type*	There are three ways to generate reflected rays: `blinn`, `phong`, or `parcinopy`. The default is `parcinopy`.
`no direct light:` *flag*	If `yes` or `on`, direct light will not be included in the amount of light reaching an object. This is useful for comparing the effect of reflected and transmitted light only. The default is `no`/`off`.

Materials

Parameter	Description
`leaf material (top):` $r\ n_r\ t\ n_t\ n$ ...	The parameters of the default material. A set of 5 parameters must be included for each of s wavelengths: r = reflectance (0-1) n_r = reflectance scattering exponent (0-∞) t = transmittance (0-1) n_t = transmitance scattering exponent (0-∞) n = refractive index (ignored, but must be present)
`leaf material (bottom):` $r\ n_r\ t\ n_t\ n$ ...	If included, the material parameters for the bottom side of each surface. Otherwise, the `top` is used for both sides. This is referenced as material index 2.
`material:` $r\ n_r\ t\ n_t\ n$	Additional materials. There may be multiple `material` commands. These materials are referenced by their position in the file, beginning with index 3. (Index 1 is `leaf material (top)`, and index 2 is `leaf material (bottom)`.)
`material parameter:` n	In the case where polygons representing the module following the communication module are sent from the modeling program, n indicates which parameter of the module specifies the material index of the object. If n=0 (the default), the `leaf material (top)` values are used.

Miscellaneous commands

Parameter	Description
`seed:` i	The seed for the random number generator, where i is an integer.
`number of runs:` n	The number of times the computation of light reaching objects is performed. Used to determine the precision of the algorithm for a given ray density. The default is 1. See the output parameters for details.
`version:` v	If $v > 1$, the first parameter of the communication module specifies the number of rays shot from each object when `rays from objects` is set to `yes`. The default is 1.
`verbose:` *flag*	Switch verbose mode `on`. The default is `off`.

On input This program always receives the communication module and the following module. All parameters of the communication module are ignored, unless the `rays from object` command is `yes`, and the `version` command is set to a number greater than 1, in which case the first parameter specify the number of rays per unit area that should be shot from the object for each wavelength.

The following module can be one of:

Module	Description
`P`(a, b, mf, mb)	A parallelogram with one diagonal in the direction of the turtle heading vector and length $2a$, and the second diagonal in the direction of the left vector with length b. The turtle position defines one vertex of the polygon. The remaining parameters are optional but, if present, specify the index of the material for the front (mf) and back (mb) of the object.

Module	Description
`T`(a, b, mf, mb)	A triangle with one edge in the direction of the turtle left vector and length a. The midpoint of this edge corresponds to the turtle position. The third vertex of the triangle is on an axis corresponding to the turtle heading vector at distance b from the turtle position. The remaining parameters are optional but, if present, specify the index of the material for the front (mf) and back (mb) of the object.
`L`$(I_1, I_2, ...I_s)$	The current intensities for different wavelengths emitted from all light sources. The parameters specify the intensities in the order given in the `source spectrum` command.
$<$ Otherwise $>$	A set of polygons representing the module. To define a complex object, homomorphism productions can specify the geometry of the module and all the polygons representing the module are transferred, and are considered a single object. The material indices can be specified by the parameters of the module following the communication module (see the `material parameter` command). This option takes precedence over modules `P` and `T`. If polygons are received for module `P` or `T`, they are used to define the object.

On output The parameters of the communication module are set depending on the `number of runs:` n:

- $n = 1$: The first s return parameters are set to the amount of light reaching the object for each wavelength (defined by `spectrum samples:` s).
- $n > 1$: The first 2 return parameters are set to the mean and standard deviation of the ratio of the amount of light reaching the object for the first two wavelengths, computed after n runs. The next $2s$ parameters are the mean and standard deviation of the amount of light reaching the object for each of wavelength (defined by `spectrum samples:` s).

The program receives information for all communication modules and the geometry of the modules that follow. These modules can be seen as leaves, stems, the ground, or objects around a plant, and are interpreted as a set of polygons. After all scene polygons are input, they are stored in a regular grid to speed up the algorithm.

The computation of light distribution starts by generating initial rays representing light of a specified intensity emitted from the light sources. Each initial ray is then traced in the grid. If the ray intersects an object, the light carried by the ray is either absorbed by the object, reflected by the surface, or transmitted through it. The fate of the ray depends on the material parameters specified in *eparamfile*.

Each material is defined by four parameters, controlling the probability of tracing a reflected ray, its scattering coefficient (affecting the direction of the reflected ray), the probability of tracing a transmitted ray, and its scattering coefficient.

Reflected or transmitted rays are traced further until the ray depth (equal to the number of hits on the ray's path) reaches the `maximum depth`, or its intensity is below the threshold set by the `Russian roulette` command. After all initial rays have been traced, parameters storing the light flux absorbed by an object are returned.

5.4 Distributed systems

The communications library includes a number of functions that were designed to allow communication between multiple plant models and environmental programs, using a single drawing window. This functionality was originally designed for *cpfg*, but has not been tested on newer operating systems. It required the *cpfg* command line argument `-C`, and `communication type: sockets`, which have both been deprecated as well.

The `-C` command line argument in *cpfg* was followed by a single string defining master and slave processes, and the sockets used:

Option	Description
`-c,`*socket,machine*`;`	The socket number and machine name to which confirmation of successful execution should be sent. The program monitors the socket for a possible request to terminate.
`-m:`*commfile*`.e,`*socket*`;`	A master process, sending data in the `.e` files to the specified socket, and expecting the reply on socket number *socket* + 1. There may be multiple `-m` options.
`-s:`*commfile*`.e,`*socket*	A slave process, expecting the data defined in *commfile*`.e,` from the specified socket. It processes the incoming data and responds back through socket number *socket* + 1. If there are multiple slave connections, the data is processed in the order given.

Note that the `executable` and `communication type` commands in the communications file, *commfile*`.e`, are ignored.

The following functions still exist in the communications library, in addition to the functions defined for creating a new environmental program (Section 4). Note that the *master* parameter in some of the functions in Section 4 would not be ignored when developing a distributed system of programs.

void **CInitialize***(char *program, char *commandstr)*
: Initialize communication, where *program* is the name of the process and is used to distinguish messages from different processes, and *commandstr* is the string following the `-C` command line option.

int **CShouldTerminate***(void)*
: Return 1 when an end-of-transmission character is sent from a control process.

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

int **CSSendString***(int master, char *item)*
: Send a string to the specified master.

int **CSSendBinaryData***(int master, char *item, int size, int n)*
: Send binary data to the specified master. The function returns 0 if the data is not sent.

int **CMBeginTransmission***(void)*
: Initialize connections to all slave processes for a single data exchange. The function always returns 1.

int **CMEndTransmission***(int step)*
: Terminate the data sent by the master in a single data exchange. The *step* parameter is the current simulation step, which is returned as the *module_id* parameter from the **CSGetData** function (Section 4.2). This function always returns 1.

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

int **CMGetNumberOfSlaves***(void)*
Return the number of slave process communicating with the master.

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

int **CMGetString***(int slave, char *str, int length)*
Receive a string from the specified slave process. Returns 0 if no string is received.

int **CMSendBinaryData***(int slave, char *item, into size, int n)*
Send binary data to the specified slave process, where n is the number of items.

int **CMGetBinaryData***(int slave, char *item, into size, int n)*
Receive a string from the specified slave process, where n is the number of items. Returns 0 if no data is received.

int **CMSendCommSymbol***(int slave, unsigned long module_id, Cmodule_type *two_modules, CTURTLE *turtle)*
Send the communications module and the following module to the specified slave process. The function returns 1 if the second module should be graphically interpreted and the resulting set of triangles transferred to the slave. This function is used within *cpfg*.

int **CMGetCommunicationModule***(int slave, unsigned long module_id, Cmodule_type *comm_module)*
Receive a communication module from the specified slave process. The function returns 0 if there are no more modules from the slave.

6 Credits

Environmental programs were designed for Open L-systems, the thesis work of Radomír Měch [6], and described in [7]. The *QuasiMC* program was developed by Mikolaj Cieslak and described in [5].

7 Document revision history

Date	Description	By
1997	Open L-systems incorporated into the *cpfg* manual	Radomír Měch
1998	User Manual for Environmental programs	Radomír Měch
2008	*QuasiMC* User Manual	Mikolaj Cieslak
2021	Consolidation of the concepts and programs in the above documentation into a single manual that applies to both *cpfg* and *lpfg*.	Lynn Mercer Mikolaj Cieslak

References

[1] J. Arvo and D. Kirk. Modeling plants with environment-sensitive automata. In *Proceedings of Ausgraph '88*, pages 27–33, 1988.

[2] N. Chiba, S. Ohkawa, K. Muraoka, and M. Miura. Visual simulation of bothanical trees based on virtual heliotropism and dormancy break. *The Journal of Visualization and Comupter Animation*, 5:3–15, 1994.

[3] CIE Technical Committee 4.2. Standardization of luminance distribution on clear skies. Technical report, Commission International de l'Eclairaze, Paris, 1973.

[4] A. Takenaka. A simulation model of tree architecture development based on growth response to local light environment. *Journal of Plant Research*, 107:321–330, 1994.

[5] Mikolaj Cieslak, Christiane Lemieux, Jim Hanan, and Przemyslaw Prusinkiewicz. Quasi-monte carlo simulation of the light environment of plants. *Functional Plant Biology*, 35(10):837–849, 2008.

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

[7] Radomír Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *Computer Graphics*, pages 397–410. Proceedings of SIGGRAPH '96, 1996.