HAL
open science

PlantGL : a Python-based geometric library for 3D plant modelling at different scales

Christophe Pradal, Frédéric Boudon, Christophe Nouguier, Jérôme Chopard, Christophe Godin

To cite this version:

Christophe Pradal, Frédéric Boudon, Christophe Nouguier, Jérôme Chopard, Christophe Godin. PlantGL : a Python-based geometric library for 3D plant modelling at different scales. [Research Report] 2007. inria-00191126v1

HAL Id: inria-00191126

https://inria.hal.science/inria-00191126v1

Submitted on 23 Nov 2007 (v1), last revised 27 Nov 2007 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

PlantGL: a Python-based geometric library for 3D plant modelling at different scales

Christophe Pradal — Frederic Boudon — Christophe Nouguier — Jérôme Chopard — Christophe Godin

N° ????

Novembre 2007

Thème BIO

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--????--FR+ENG

PlantGL: a Python-based geometric library for 3D plant modelling at different scales

Christophe Pradal * , Frederic Boudon * , Christophe Nouguier , Jérôme Chopard , Christophe Godin

Thème BIO — Systèmes biologiques
Équipes-Projets VirtualPlants

Rapport de recherche n° ???? — Novembre 2007 — 34 pages

Abstract: In this paper, we present `PlantGL`, an open-source graphic toolkit for the creation, simulation and analysis of 3D virtual plants. This `C++` geometric library is embedded in the `Python` language which makes it a powerful user-interactive platform for plant modelling in various biological application domains.

`PlantGL` makes it possible to build and manipulate geometric models of plants or plant parts, ranging from tissues and organs to plant populations. Based on a scene graph augmented with primitives dedicated to plant representation, several methods are provided to create plant architectures from either field measurements or procedural algorithms. Because they reveal particularly useful in plant design and analysis, special attention has been paid to the definition and use of branching system envelopes. Several examples from different modelling applications illustrate how `PlantGL` can be used to construct, analyse or manipulate geometric models at different scales.

Key-words: software architecture, surfacic geometry, virtual plants, plant architecture, crown envelops, canopy reconstruction, scene graph

* These two authors contributed equally to the paper

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

PlantGL : une bibliothèque géométrique en Python pour la modélisation 3D des plantes à différentes échelles

Résumé : Dans cet article, nous présentons `PlantGL`, une bibliothèque graphique libre pour la création, la simulation et l'analyse de plantes virtuelles 3D. Cette bibliothèque géométrique écrite en `C++` est accessible depuis le language Python. Elle constitue la base d'une plateforme interactive pour la modélisation des plantes dans plusieurs domaines applicatifs de la biologie.

`PlantGL` permet de construire et de manipuler des modèles géométriques de plantes à differentes échelles, depuis les tissus cellulaires et les organes jusqu'aux populations de plantes. Plusieurs méthodes sont proposées pour génerer des architectures de plantes à partir de données mesurées sur le terrain ou de méthodes procédurales. Ces méthodes s'appuient sur une structure de graphe de scène augmentée de primitives géométriques adaptées à la representation de plantes. Du fait de leur importance pour le design et l'analyse de plante, une attention particulière a été apportée à la définition et à l'utilisation d'enveloppes pour représenter des systèmes ramifiés . Plusieurs exemples applicatifs illustrent comment `PlantGL` peut être utilisée pour construire, analyser et manipuler des modèles géométrique de plantes à differentes échelles.

Mots-clés : architecture logicielle, géométrie surfacique, plantes virtuelles, architecture des plantes, enveloppes de houppiers, reconstruction de canopées, graphe de scène

1 Introduction

The representation of plant forms in computer scenes has for long been recognized as a difficult problem in computer graphics applications. In the last two decades, several algorithms and software platforms have been proposed to solve this problem with a continuously improving level of efficiency, e.g. [1, 2, 3, 4, 5, 6, 7, 8]. Due to the increasing use of computer models in biological research, the design of 3D geometric models of plants has also become an important aspect of various biological applications in plant science, e.g. [9, 10, 11, 12, 13, 14, 15]. These applications raise specific problems that derive from the need to represent plants with a botanical or geometric accuracy at different scales, from tissues to plant communities. However, in comparison with computer graphics applications, less effort has been devoted to the development of geometric modelling systems adapted to the requirements of biological applications.

In this context, the most successful and widespread plant modelling system has been developed by P. Prusinkiewicz and his team since the late 80's at the interface between biology and computer graphics. They designed a computer platform, known as `L-Studio/VLab`, for the simulation of plant growth based on L-systems [16, 1]. This system makes it possible to model the development of plants with efficiency and flexibility as a process of bracketed-string rewriting. In a recent version of `LStudio/VLab`, Karwowski and Prusinkiewicz changed the original `cpfg` language for a compiled language, L+C, built on the top of the C++ programming language. The resulting gain of expressiveness facilitates the specification of complex plant models in L+C [18]. An alternative implementation of a L-system-based software for plant modeling was designed by W. Kurth [19] in the context of forestry applications. This simulation system, called `GroGra`, was also recently re-engineered in order to model the development of objects more complex than bracketed strings. The resulting simulation system, `GroIMP`, is an open-source software that extends the chain rewriting principle of L-Systems to general graph rewriting with relational graph growth (RGG), [20, 21]. Similarly to L+C, this system has been defined on the top of a widely used programming language (here Java). Non-language oriented platforms were also developed. One of the first ones was designed by the AMAP group. The `AMAP` software [2, 22] makes it possible to build plants by tuning the parameters of a predefined, hard-coded, model. Geometric symbols for flowers, leaves, fruits, *etc.*, are defined in a symbol library and can be modified or created with specific editors developed by AMAP. In this framework, a wide set of parameter-files has been designed corresponding to various plant species. In the context of applications more oriented toward computer graphics, the `XFrog` software [4, 23] is a popular example of a plant simulation system dedicated to the intuitive design of plant geometric models. In `XFrog`, components representing basic plant organs like leaves, spines, flowerlets or branches can be multiplied in space using high-level multiplier components. Plants are thus defined as graphs representing series of multiplication operations. The `XFrog` system provides an easy to use, intuitive system to design plant models, with little biological expertise needed.

Therefore, if accuracy, conciseness and transparency of the modeling process is required, object-oriented, rule-based platforms, such as `L-studio/VLab` or `GroIMP`, are good candidates for modelers. If interactive and intuitive model design is required, with little biological expertise, then component-based or

sketch-based systems like XFrog are the best candidates. However, if easiness to explore and mathematically analyse plant scenes is required, none of the above approaches is completely satisfactory. Such an operation requires high-level user interaction with plant scenes and dedicated high-level mathematical primitives. In this aim, our team developed several years ago the AMAPmod software [24], and its most recent version, VPlants, which enables modelers to create, explore and analyse plant architecture databases using a powerful script language. In a way complementary to L-Studio/VLab, VPlants allows the user to efficiently analyse plant architecture databases and scenes from many exploratory perspectives in a language-based, interactive, manner [25, 26, 27, 28]. The PlantGL library was developed to support geometric processing on plant scenes in VPlants, for applications ranging from computer graphics [29, 30] to different areas of biological modeling [31, 32, 33, 14, 34, 35]. A number of high-level requirements were imposed by this context. Similarly to AMAPmod/VPlants, the library should be open-source, it should be fully compatible with the data structure used in AMAPmod/VPlants to represent plants, *i.e.* multi-scale tree graphs (MTGs), it should be accessible through an efficient script language to favor interactive exploration of plant databases, it should be easy to use for biologists or modellers and should not impose a particular modelling approach, it should be easily extended by users to progressively adapt to the great variety of plant modelling applications, and finally, it should be interoperable with other main plant modelling platforms.

These main requirements lead us to integrate a number of new and original features in PlantGL that makes it particularly adapted to plant modelling. It is based on the script language Python, which enables the user to manipulate geometric models interactively and incrementally, without compiling the scene code or recomputing the entire scene after each scene modification. The embedding in Python is critical for a number of additional reasons: i) the modeller has access to a powerful object-oriented language for the design of geometric scenes, ii) the language is independent of any underlying modelling paradigm and allows the definition of new procedural models, iii) high-level manipulations of plant scenes provide users the ease to concentrate on application issues rather than on technical details, iv) the large set of available Python scientific packages can be freely and easily accessed by modelers in their applications. From its contents perspective, PlantGL provides a set of geometric primitives for plant modelling that can be combined in scene-graphs dedicated to multiscale plant representation. New primitives were developed to address biological questions at either macroscopic or microscopic scales. At plant scale, envelope-based primitives have been designed to model plant crowns as volumetric objects. At cell scale, tissue objects representing arrangements of plant cells enable users to model the development of plant tissues such as meristems. Particular attention has been paid to the design of the library to achieve a high-level of reuse and extensibility (e.g. data structures, algorithms and GUIs are clearly separated). To favor the exchange of models and databases between users, PlantGL can communicate with other modelling platforms such as LStudio/VLab and is available under an open-source license.

In this paper, we present the PlantGL geometric library and its application to plant modelling. Section 2 describes the design principles and rationales that underly the library architecture. It also briefly introduces the main scene graph structure and the different library objects: geometric models, transformations,

algorithms and visualization components. Then, a detailed description of the geometric models and methods dedicated to the construction of plant scenes is provided in section 3. This includes the modeling of organs, crowns, foliage, branching systems and plant tissues. A final section illustrates how `PlantGL` components can be used and assembled to answer particular questions from computer graphics or biological applications at different levels of a modelling approach: creating, analysing, simulating and assessing plant models.

2 `PlantGL` design and implementation

A number of high-level goals have guided the design and development of `PlantGL` to optimize its reusability and diffusion:

- *Usefulness* : `PlantGL` is primarily dedicated to researchers in the plant modelling community who do not necessarily have any *a priori* knowledge in computer graphics. Its interface with modellers and end-users should be intuitive with a short learning curve.
- *Genericity* : `PlantGL` should not impose a particular modelling paradigm on users. Rather, it should allow them to design their own approach in a powerful way.
- *Quality* : Quality is a major aspect of software diffusion and reusability. `PlantGL` should therefore be developed with software quality guarantees.
- *Interoperability* : `PlantGL` should also be interoperable with various plant modelling systems (e.g. `L-studio/VLab`, `AMAP`, *etc.*) and graphic toolkits (e.g. `Pov-Ray`, `Vrml`, `Blender`, *etc.*).
- *Portability* : `PlantGL` should be available on major operating systems (e.g. `Linux`, `Microsoft Windows`, `MacOSX`).

In this section we detail how these requirements have been translated into choices at different levels of the system design.

2.1 Software architecture

The system architecture derives from a set of key design choices:

- *Open-source* : `PlantGL` is an open-source software that may be freely used and extended.
- *Script-language based system* : `PlantGL` is built on the top of the powerful script language, Python. The use of a script language allows users to have a high level of computational interaction with their models.
- *Software engineering* : Object-oriented design is useful to organize large computational projects and enhance code reuse. However, designing reusable and flexible software remains a difficult task. We addressed this problem by using advanced software engineering tools such as *design patterns* [36].

- *Modularity* : `PlantGL` is composed of several independent modules like a geometric library, GUI components and `Python` wrappers. They can be used alone or combined within a specific application.
- *Hybrid System* : Core computational components of `PlantGL` are implemented in the `C++` compiled language for performance. However, for flexibility of use, these components are also exported in the `Python` language.

Among all the script available languages, Python was found to present a unique compromise between the following features:

- Python is an open-source software;
- it is available on the main operating systems;
- it is a powerful object-oriented language;
- it is simple to use and its syntax is sufficiently intuitive for non-computer scientists (e.g. for biologists);
- it is fully compatible with C++;
- the Python community is large and very active;
- a large number of other scientific libraries are available and can be imported in the language at any time of a model development.

The overall architecture layout of `PlantGL` is shown in Figure 1, including several layers of encapsulation and abstraction. The geometric library and the GUI lie in the core of `PlantGL`. The geometric library encapsulates a *scene-graph* data structure, a taxonomy of *geometric objects* and a set of *algorithms* (for instance for `OpenGL` rendering). The GUI is developed as a set of `Qt` widgets and can be combined with previous components to provides visualization facilities. On top of this first layer, a set of wrappers create interfaces of the `C++` classes into the `Python` language. All these interfaces are gathered into a `Python` module named `PlantGL`. This module is integrated seamlessly with standard `Python` and thus enables further abstraction layers, written in `Python`. Extension of the library can be done using either `C++` or `Python`.

2.2 Basic components

The basic components of `PlantGL` are *scene-graphs* that contain scene objects (geometry, appearance, transformation, *etc.*), *actions* that define algorithms that can be applied to scene-graphs, and *the visualization tools*.

2.2.1 Scene-graphs

Scene-graphs are a common data structure used in computer graphics to model scenes [37]. They provide both a high-level view of the application's data as well as a mechanism to perform the low-level computations necessary to drive a rendering pipeline such as OpenGL. Basically, a *scene-graph* is a directed, acyclic graph (DAG), whose nodes hold the information about the elements that compose the scene. In a DAG, nodes may have more than one parent, which

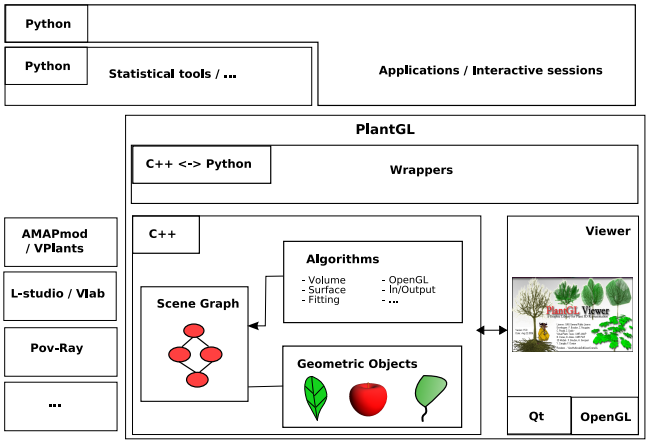


Figure 1: Layout of the `PlantGL` architecture. It contains three main components: a geometric library, a GUI and Wrappers. The two first components are written in C++ for efficiency. `PlantGL` primitives can be used by modellers to develop scripts, procedures and applications in the embedding language `Python`. Data structures can be imported from and exported to other plant modelling softwares

enables parent nodes to share child nodes within the graph via the *instantiation* mechanism.

In `PlantGL`, an object-oriented version of scene-graphs was implemented to facilitate the customization of scene-graph node processing. Scene-graphs are made of nodes of different types. Types may be one of *geometry*, *appearance*, *shape*, *transformation*, and *group*. Scene-graph nodes are organized as a DAG hierarchy. Terminal nodes contain *shapes* pointing to both a geometry node (which contains a geometric model, see section 2.2.2) and an appearance node (e.g. which defines a color, a material or a texture). Non terminal nodes correspond to group or transformation nodes (see section 2.2.3). They are used to build complex objects from simple geometric models.

2.2.2 Geometric models

Two families of geometric models are available in the library: *Explicit* and *Parametric* models. Explicit models are defined as sets of discrete primitives like points, lines or polygons that can be directly drawn by graphics cards. They enable to express geometric models in a generic but low level way and are thus complex to manipulate. On the opposite, parametric models are defined using equations that involve one or several parameters. Parametric models offer a higher level of abstraction and are thus simpler to manipulate. However, to be displayed, parametric models have to be transformed into an explicit representation. This is done by discretizing algorithms defined for each parametric model. The discretization process is controlled by parameters indicating how many primitives have to be created.

`PlantGL` contains a number of geometric models to represent 3D points, curves, surfaces and volumes. Curves include *Polyline*, *Bézier* and *NURBS*. They are used for instance to generate surfaces or to represent the medial axis of

an object. Surfaces and volumes may be represented by *PolygonMesh*, *surfaces of revolution* (SOR), *Patch*, *Extrusion* and *Hull*. SOR include simple shapes (sphere, cylinder, frustum, paraboloid) and more generic ones like *Revolution* (revolution of a curve along an axis). Patch surfaces include *Bézier*, *NURBS*, and *ElevationGrid* surfaces used for terrain modelling. *Extrusion* (or generalized cylinder) is a sweep surface useful for the modelling of branches. Hulls are 3D envelopes that may be used to model tree crowns.

2.2.3 Transformations and compositions

Transformed nodes allow the user to define the position, orientation and size of a geometric object. They contain classical translation, rotation and scaling nodes. Rotations may be specified in various forms (via matrices, Euler angles,...). A particular transformation named *Taper* proposes a gradual scaling of subsequent perpendicular planes to a given direction. This non-affine transformation enables the modeller to transform a cylinder into a truncated cone and is generally used in the representation of branch segments. *Transformed* nodes embed a shape on which the transformation is applied.

A *Group* node, or a composite node, defines a union of scene objects which embeds a list of geometry nodes. Groups are mainly used to apply a given transformation to a set of objects as a whole. A *Group* node may point to other Group nodes and define a subgraph of nodes.

2.2.4 Algorithms

In `PlantGL`, algorithms are separated from data structures for flexibility and maintenance purpose. To be applied on a scene, some algorithms need to traverse the scene-graph structure. Scene-graph traversals are different from standard graph traversal since nodes are heterogeneous and need to be treated according to their type. To achieve this efficiently, traversal processes are based on double dispatch technique via the visitor design pattern [36]. This design pattern makes it possible to keep functions outside node objects by delegating the mapping from node to function to a separate visitor object called an *action*. The *action* holds all the algorithm functions for every types of nodes in the system. It is thus possible to add new algorithms by implementing new actions without modifying the node classes (see section 2.4). By splitting specification and implementation, `PlantGL` scene-graph preserves flexibility without loosing performances.

In the actual implementation, `PlantGL` supports approximatively 40 *actions*. The main ones can be classified into the following categories :

- *Converter* : Algorithms that convert geometries into explicit representations (e.g. *Discretizer* or *Tesselator* actions), or into Bounding Boxes or Wire.
- *Renderer* : `OpenGL` rendering for various modes: volumetric, skeleton, bounding box, *etc.*
- *Import / Export* : parser and exporter of various classical formats namely `PovRay`, `Vrml`, *etc.* or to plant dedicated systems like `AMAPmod/VPlants`, `LStudio/VLab`, `AMAPsim` and `VegeStar`.

- *Fitting* : Fitting algorithms have been integrated into the library. Facilities to compute convex hull [38], bounding volumes (sphere [39], box) can be used to compute a global representation from a set of detailed shapes.
- *Projection* : Algorithm based on `OpenGL` to project 3D scenes onto 2D planes.
- *Grids* : Partitioning of the 3D space into multiscale grids such as Octree. Detection of intersection between voxels and triangles of the scene is carried out using the algorithm described in [40].
- *Ray casting* : Two versions of the algorithms are implemented. The first one is CPU-based, and is implemented as `C++` routines with grids used to determine which shape to test for ray intersection. The second one is GPU-based, and uses `OpenGL` picking feature to compute the different shape interceptions of a small orthographic viewport, featuring a beam.
- *Analysis* : Algorithms that compute particular properties of the scene, like the total number of polygons, the surface or volume, the center or the inertia axes of the scene.

The use of these algorithms is illustrated in section 4 in the context of different biological applications.

2.2.5 Visualization tools

The `PlantGL` *Viewer* (Figure 2) provides facilities to visualize scene-graphs. Different types of rendering modes are available including volumetric, wire, skeleton, and bounding box. Camera and lights positions and properties can be interactively modified. Scene-graphs organization and node properties can be explored with adapted widgets allowing to access to various pieces of information about the scene, like the number, volume, surface or bounding box of the scene elements. Simple edition features (such as a shape material editor) are available. Screen-shots can be easily made and exported in documents or assembled into movies. Most of the viewer features can be set using buttons or context menus and are also available with procedure call.

The viewer and the scene access are multi-threaded so that a user can still interact with the scene from a `Python` shell during visualization.

Several types of dialog boxes can also be interactively created in the Viewer from the `Python` shell. Results of the dialog is returned to the `Python` process using multi-threaded communications (see section 2.4). This enables graphical control of a `Python` modelling process through the Viewer.

2.3 Creating scene-graphs

There are different ways for application developers to create and process a scene-graph depending both on performance constraints and their familiarity with computer graphics concepts.

- *Declarative language.* `PlantGL` provides a declarative language, `GML`, similar to `VRML` [41] with extensions for plant geometric objects. `GML` adds

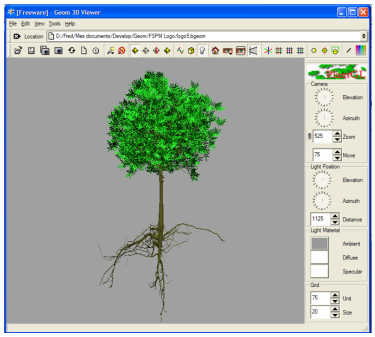


Figure 2: Visualization of the detailed representation of a 15 years walnut tree [10] with the PlantGL viewer.

persistence capabilities to `PlantGL` in both ascii and binary versions. External applications can use this file format to describe a static scene and call `PlantGL` as an external module to render it. The GML language is described in [42].

- *Scene-graph programming API.* For `C++` applications, it is possible to link directly with the `PlantGL` library. In this case, `PlantGL` features can be extended by adding new objects and new actions.

- *Interpreted language.* The full `PlantGL API` is accessible from the `Python` language. Combination of high level tools and language such as `PlantGL` and `Python` allows rapid prototyping of new models and applications [43]. Moreover, numerous scientific modules for `Python` exist and can be reused [44].

`PlantGL` provides different ways for users and applications to create scenes. A scene can first be described in GML. The GML file can then be loaded and the scene rendered. In such a case, the scene is static and the communication is one way. For performance and fast communication, a second solution consists of creating the scene in C++ and link the application with the `C++ PlantGL` library. Finally, for rapid developments, prototyping, and tests, a scene can be created procedurally in `Python`.

2.4 Implementation issues

Different issues have to be addressed in order to implement an efficient scene-graph that preserves performances and flexibility. Literature on scene-graphs [45] offers various interesting solutions that were formalized as design patterns and which inspired our implementation.

Memory management - Scene-graphs have to deal with large number of geometric elements. This is particularly true for natural scenes. Memory consumption is thus an issue. Repetitive structures such as trees enable massive use of *instantiation*. This technique, however, implies to reference several times the same object. Allocation and deallocation of this object in memory can thus be problematic. To address this issue, we use *Reference counting pointers* [46]

which manages pointers to dynamically allocated objects. They are responsible for automatic deletion of the objects when no longer needed.

Double dispatch - As stated in section 2.2.4, scene-graph traversal implements a double-dispatch technique via the visitor design pattern. With such an approach, algorithms can be added without any modification of the hierarchy of objects. However, at runtime, matching a particular data structure to its appropriate algorithms is not a trivial task. Each node of the scene-graph has an *apply* method that takes an action object as an argument. The node makes a call to the action, passing itself as an argument, and the action executes its algorithm depending on the actual node type. Thus, the visitor design pattern simulates a double-dispatch in `C++`, which is a single-dispatch object oriented language.

Wrappers - The seamless transition between `C++` and `Python` is ensured by the `Boost.Python` library [47]. In contrast with concurrent tools, the interaction between `C++` and `Python` is encoded explicitly in `C++`. This wrapping code is then compiled into a dynamic library, usable as a `Python` module. On one hand, `Boost.Python` maps `C++` classes and their interfaces to corresponding `Python` classes. On the other hand, it transparently converts `Python` objects back to `C++` pointers or references, thus providing dynamic, run-time dependent interaction between `C++` based objects. Since `Boost.Python` is based on advanced meta-programming techniques, the code wrapping mainly consists of simple declaration of entry points in the library and is automatically translated into conversion functions.

Graphic performances -Plant geometry generally rely on important vertex set. In order to minimize time cost for sending all the vertex to the GPU, the vertex are packed into arrays that can be sent in one command to the card. Moreover, `OpenGL` command for drawing shapes instantiated multiple times are packed into display lists that can be reused efficiently when needed.

Multi-threading - Viewer and shell processing are done in separated threads. For this, `Qt` threads implementation were used. Inter-threads communication is made using `Qt` event dispatch mechanism which is extended with synchronized dispatch using mutex.

3 Construction of Plant Models

3.1 Plant organs

As stated in section 2.2.2, a taxonomy of geometric models is implemented in `PlantGL`. During implementation, we focused on models useful for natural scene modelling and rendering. For instance, `PlantGL` contains thus cylinders and frustums to represent inter-nodes, Bézier and NURBS patches to model complex organs such as leaves or flowers and generalized cylinders for branches and stems.

During the creation of a scene-graph, complex objects can be modeled using composition and instantiation. Simple procedures can be written in `Python` that position predefined symbols, for instance petals or leaves, using Transformation and Group objects, to create more complex objects by composition (Figure 3). `Python`, as a modelling language, allows the modeller to express a full range of modelling techniques. For instance, the pine cone (Figure 3.c) is built by placing

its scale using the golden angle. The following code sketches the placement of each scales. Size of successive scales in the spiral follows a logarithmic laws in this case.

```
from PlantGL import *
sc = Scene()

deltaAngle = pi /(1+sqrt(5)) # the golden angle
def distToTrunk(u): # with u in [0,2], u < 1 for the bottom part
  if i < 1: return MaxDist*u
  else: return MaxDist*((2-u)**2)
def sizeScale(u):
  if i < 1: return MaxSize*log(1+u,2)
  else: return MaxDist*log(3-u,2)
for i in range(nbScale):
  s += AxisRotated((0,0,1),i*deltaAngle,
    Translated((distToTrunk(2*i/nbScale),0,i),
      Scaled(sizeScale(2*i/nbScale),smb))
```

(a) (b) (c)

Figure 3: (a) a tulip, (b) a cactus and (c) a pine cone. Models were created with simple `Python` procedure that create and position organs, like petal and leaves.

3.2 Crown models

3.2.1 Envelopes

Tree crown envelopes are used in different contexts like studying plant interaction with its environment (i.e. radiative transfers in canopies [9] or competition for space [48]), 3D tree model reconstruction from photographs [6], and interactive rendering [49]. They are one original key issue of `PlantGL`. In this section, we describe in detail three specific envelope models that were specifically designed to represent plant volumes: asymmetric, extruded and skinned hulls.

Asymmetric Hull - This envelope model, originally proposed by Horn [50] and Koop [51], then extended by Cescatti [9], makes it possible to define asymmetric crown shapes based on ellipsoidal shapes.

The envelope is defined by six control points and two shape factors C_T and C_B (see Figure 4). The two first control point, P_T and P_B, respectively represent top and base points of the crown. The four other points P_1 to P_4 represent the different radius of the crown in four orthogonal directions. P_1 and P_3 are constrained to lie in the xz plane and P_2 and P_4 in yz plane. These points define

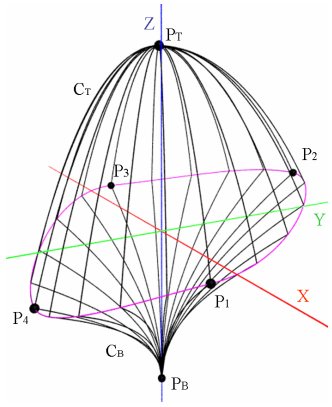


Figure 4: Asymmetric hull parameters.

a peripheral line L at biggest width of the crown when projecting it into the horizontal plane. The projection of L in the horizontal plane is composed of four ellipse quarters centered on $(0, 0)$. The height of the points of L is defined as an interpolation of the heights of the control points. To be continuous at the control points, we used factors $\cos^2\theta$ and $\sin^2\theta$. Thus, a point $P_{\theta,i,j}$ of a quarter of L between control points P_i and P_j with $i, j \in [(1, 2), (2, 3), (3, 4), (4, 1)]$, located at an angle $\theta \in \left[0, \frac{\pi}{2}\right)$, is defined as

$$P_{\theta,i,j} = \left[r_{P_i}\cos\theta, r_{P_j} sin\theta, z_{P_i}\cos^2\theta + z_{P_j}\sin^2\theta\right]. \tag{1}$$

The two shape factors C_T and C_B describe the curvature of the crown above and below the peripheral line. Points of L are connected to the top and base points with quarters of super-ellipse respectively of degrees C_T and C_B. Let $P_l \in L$ and P_T be the top point of the envelope, the above super-ellipse quarter connecting P_l and P_T is defined as

$$\left\{P = (\theta, r, z) | \frac{(r - r_{p_T})^{C_T}}{(r_{P_l} - r_{p_T})^{C_T}} + \frac{(z - z_{p_T})^{C_T}}{(z_{P_l} - z_{p_T})^{C_T}} = 1\right\}. \tag{2}$$

Equivalent equation is obtained for below super-ellipse quarter using P_B and C_B instead of P_T and C_T.

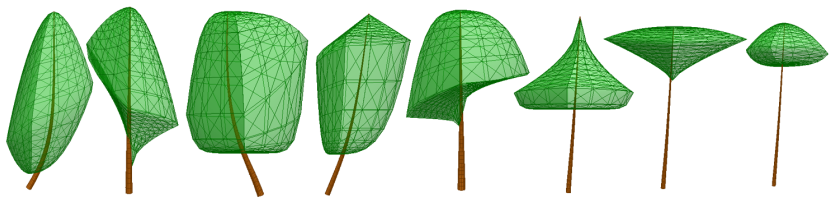


Figure 5: Example of asymmetric hulls showing plasticity of this model to represent tree crowns (inspired from Cescatti [9]).

Different shape factor values generate conical ($C_i = 1$), ellipsoidal ($C_i = 2$), concave ($C_i \in]0, 1[$), or convex ($C_i \in]1, \infty[$)shapes. A great variety of shapes can be achieved by modifying shape factor values (see Figure 5). Cescatti proposes

a simple methodology to measure the six control points and estimate the shape factors directly in the field. In `PlantGL`, a graphic editor has been implemented which makes it possible to build envelopes from photographs or drawings. For this, reference images can be displayed in the background of the different edition views. An illustration of this shape usage can be found for the interactive design of bonsai tree [29].

Extruded Hull - This model of extruded envelope was originally proposed by Birnbaum [52]. Such an envelope is built from a vertical and an horizontal profile, respectively in the xz plane and in the xy plane. To build the envelope, the horizontal profile is swept inside the vertical profile.

For this, the vertical profile V is virtually split into slices by planes passing through equidistant points from the top (or by horizontal planes). These slices are used to map H inside V. To simplify this procedure, two fixed points, B and T, respectively at the top and bottom of the vertical profile, are defined and V is split into two open profile curves V_l and V_r, respectively the left and the right part of V, with their first and last points equals to B and T. A slice is thus defined using two mapping points $V_l(u)$ and $V_r(u)$, with $u \in [u_{min}, u_{max}]$.

On the horizontal profile H, two anchor points, H_l and H_r, are defined. We consider, without any loss of generality, that $\overrightarrow{H_l H_r}$ is collinear to the $\overrightarrow{x}$ axis and H_l and H_r corresponds respectively to the min and max x values of H. A reference frame R_0 associated with H can thus be defined using $\overrightarrow{H_l H_r}$ and the $\overrightarrow{y}$ axis.

For each slice, a second reference frame R can be defined with $\overrightarrow{V_l(u)V_r(r)}$ and $\overrightarrow{y}$. An horizontal section of the extruded hull is computed at R so that the resulting curve fits inside V (see Figure 6.b). The transformation thus consists of mapping R into R_0. It is made of a translation of H from H_l to $V_l(u)$, a rotation around the $\overrightarrow{y}$ axis of an angle $\alpha(u)$ equal to the angle between the $\overrightarrow{x}$ axis and $\overrightarrow{V_l(u)V_r(u)}$, and finally, a scaling factor $\left\|\overrightarrow{V_l(u)V_r(u)}\right\| / \left\|\overrightarrow{H_l H_r}\right\|$.

The equation of the Extruded Hull surface is thus:

$$S(u,v) = V_l(u) + \frac{\|\overrightarrow{V_l(u)V_r(u)}\|}{\|\overrightarrow{H_l H_r}\|} * R_y(\alpha(u))(H(v) - H_l) \tag{3}$$

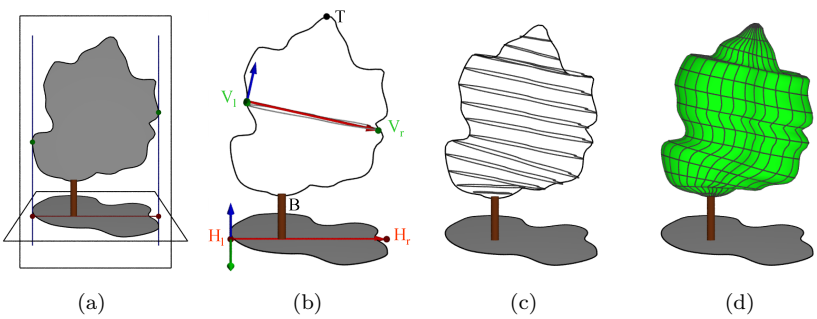


Figure 6: Extruded Hull construction. (a) Acquisition of a vertical and an horizontal profile. (b) Transformation of the horizontal profile into a section of the hull. (c) Computation of all sections (d) Mesh reconstruction.

Skinned Hull - Surface skinning, also known as lofting [53, 54, 55], is a process of passing a surface through a set of profile curves.

A *skinned hull* is built from a set of planar profiles, $\{P_k(u), k = 0, \ldots, K\}$ and a set of associated angles $\{\alpha_k, k = 0, \ldots, K\}$. The envelope of a skinned hull is a closed skinned surface which interpolate all the profiles. Profiles are thus iso-parametric curves of the surface (see Figure 7.a). The plane of each profile results from a rotation of the vertical xz plane of an angle α_k around the z axis.

The skinned hull is a generalization of a surface of revolution with a variational section being an interpolation of the various profiles. For the particular case of a unique profile, the surface is a surface of revolution. Similarly to the vertical profile of the extruded hull, two fixed points B and T are defined and all profiles are split into two open profile curves. In this case, $\overrightarrow{BT}$ defines the rotation axis of the shape.

We assume that all the profiles $P_k(u)$ are non-rational B-spline curves. First, profiles are homogenized to have the same parametrization and the same number of control points. A first homogenization algorithm [54] consists of computing a common knot vector U for all curves by adding all the different knots of each curve knot vector. Although this method is efficient, this result in a large number of knots and thus a large number of control points. In general, if the average number of knots is m and the number of profiles is k, the total number of knots for each profile can be in $O(km)$, instead of $O(m)$. To reduce the number of knots, the algorithm is modified as follows: a knot is added only if the distance to the already added closest knot is superior to a given tolerance. However, the homogenized profiles are an approximation of the original profiles. The resulting profiles are

$$P_k(u) = \sum_{i=0}^{n} N_{i,p}(u) P_{i,k} \tag{4}$$

where p is the common degree of the profiles, n is the common number of control points, the $\{N_{i,p}(u)\}$ are the pth-degree B-spline basis function defined on the common knot vector U, and $P_{i,k}$ is the ith control point of the kth profile P_k.

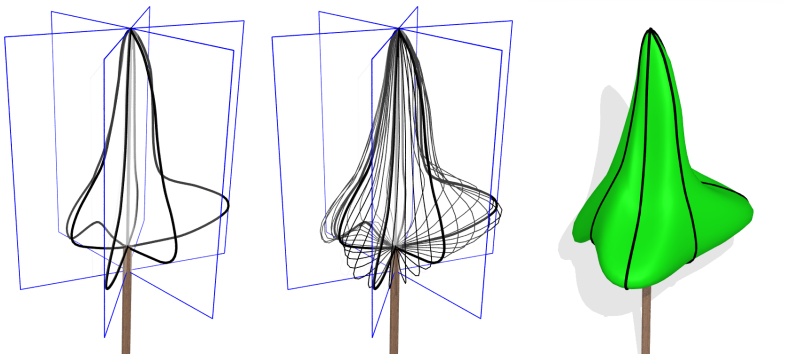
Using the homogenized profiles, a variational section Q can be defined. For each angle α around the rotation axis, Q defines a planar section of the skin surface (see Figure 7.b). Q is computed by interpolating all the profiles P_k at the given parameters α_k. It is defined as

$$Q(u, \alpha) = \sum_{i=0}^{n} N_{i,p}(u) Q_i(\alpha) \tag{5}$$

where the $Q_i(\alpha)$ are a variational form of control points.

Q_i are computed by an interpolation through the $P_{i,0}, \ldots, P_{i,K}$ points at the parameters $\alpha_0, \ldots, \alpha_K$ using a global interpolation method described in [54]. Let q be the chosen degree of the interpolation such as $q < K$. $Q_i(\alpha)$ are defined as

$$Q_i(\alpha) = \sum_{j=0}^{K} N_{j,q}(\alpha) R_{i,j} \tag{6}$$



(a) (b) (c)

Figure 7: Skinned Hull algorithm. (a) User define a set of planar profiles in different planes around the z axis (in black). (b) Profiles are interpolated to compute different sections (in grey). (c) The surface is computed. Input profiles are iso-parametric curves of the resulting surface.

where the control points $R_{i,j}$ are computed by solving interpolation constraints that results in a system of linear equations:

$$\forall i \in [0,n], \forall k \in [0,K], P_{i,k} = Q_i(\alpha_k) = \sum_{j=0}^{K} N_{j,q}(\alpha_k) R_{i,j} \tag{7}$$

The knot vector V of the Q_i is defined directly from the $\{\alpha_k\}$.

Geometrically, the surface is obtained by rotating $Q(u,\alpha)$ about the z axis, α being the rotation angle. It is defined as

$$S(u,\alpha) = (\cos\alpha Q_x(u,\alpha), \sin\alpha Q_x(u,\alpha), Q_z(u,\alpha)). \tag{8}$$

3.2.2 Foliage distributions

In particular studies such as light interception in eco-physiology or pest propagation in epidemiology, only the arrangement of leaves in space is relevant. This lead us to define adequate means to specify leaf distributions independently of branching systems.

A first simple strategy consists of creating random foliage according to different statistical distributions. This can be easily expressed in `PlantGL` as illustrated by the following example. Let us consider a random canopy foliage whose leaf distribution varies according to the height in the canopy. The following code sketches the creation of three horizontal layers of vegetation with different statistical properties.

```
from PlantGL import *
from random import uniform
sc = Scene()
for i in range(nbleaves):
  p = uniform(0,1)
  if p <= p1:
    height = uniform(bottom_layer1,top_layer1)
```

```
  elif p1 < p <= p2:
    height = uniform(bottom_layer2,top_layer2)
  else:
    height = uniform(bottom_layer3,top_layer3)
 # random position and orientation of leaves at the chosen altitude
 pos = (uniform(xmin,xmax),uniform(ymin,ymax),height)
 az, el, roll = uniform(-pi,pi),uniform(0,pi),uniform(-pi,pi)
 sc += Translated(pos,EulerRotated(az,el,roll,leaf_symbol)))
```

Figure 8 shows the resulting random foliage (trunk generation is not described in the code).

Figure 8: A layered canopy foliage. The probabilities for a leaf to be in the first, second or third layer are respectively 0.1, 0.7 and 0.2.

Plant foliage may also exhibit specific leaf arrangement with regular and deterministic properties. If the regularity corresponds to a form of spatial periodicity at a given scale, a procedural method similar to the previous one for random foliage can be easily used. However, some plants like ferns or pines show remarkable spatial organization of their foliage with several levels of aggregation and similar structures repeated at the different scales [14]. Such fractal spatial organizations can be captured by 3D *Iterated Function Systems* (IFS) [56].

An IFS is a set of contracting affine transformations $\{T_i\}_{i\in[1,M]}\}$. The union of these transformations define a contracting operator F such that:

$$F = T_1 \cup T_2 \cup \ldots \cup T_M. \tag{9}$$

The F operator may be applied iteratively to an arbitrary initial 3D geometric model, I, called the *initiator*. At the nth iteration the obtained object L_n has a pre-fractal structure composed of a number of elements increasing exponentially with n (while the size of each element decreases at an equivalent rate).

$$L_n = F^n(I) \tag{10}$$

When n tends to infinity, the iteration process tends to a fractal object L_∞, called the attractor. The attractor only depends on F (and not on the initiator) and has a known fractal dimension that depends on the contraction factors and number of F transformations, e.g. [57].

IFSs have been implemented in `PlantGL` as a transformation primitive. They may be used to construct reference virtual plant foliages with *a priori* determined self-similar structures and dimensions, Figure 9, e.g. [14, 33].

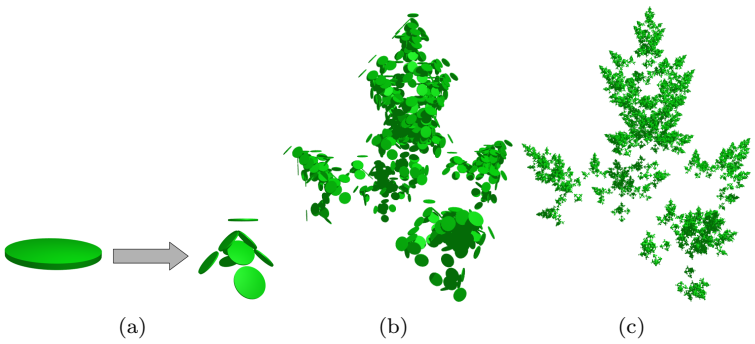


Figure 9: Construction of a fractal foliage using an IFS. (a) The initiator is a disc (representing a leaf shape). In the first iteration, the initiator is duplicated, translated, rotated and resized according to the affine transformations that compose F, leading to L_1. (b-c) IFS foliages L_3 and L_5 at iteration depths 3 and 5. The theoretical dimension of this foliage is 2.0

3.3 Branching system modelling

3.3.1 Scene-graphs for plants

In `PlantGL`, the construction of a branching systems comes down to instantiate a *p-scene-graph*. A p-scene-graph corresponds to an adaptation of the PlantGL scene-graph to the representation of plant branching structures. For this, a particular set of nodes in the p-scene-graph, named structural nodes, are introduced and represent the different components of the plant. These nodes are organized as a tree graph (as described in [58]) in which two types of edges can be specified to distinguish branching (`+`) and succession (`<`) relationships between parent and child nodes, Figure 10.a. In addition, each structural node is connected with transformation, geometry and appearance nodes that define the representation of each component. In p-scene-graphs, transformations can either be specified in an absolute or relative mode. In the absolute mode, transformations are expressed with respect to a common global reference frame whereas, in the relative mode, transformations are expressed in the reference frame of the parent component in the tree graph.

The topological relationships specified in the p-scene-graph express physical connections between plant components. To give a valid representation, such relationships can be translated as geometrical connections of the components representations. For this, a set of constraints, namely *within-scale constraints*, can formalize these geometric connections [60]. These constraints may specify for instance continuity relationship between the geometric parameters of connected components (end points, diameters, etc.). These constraints are used to ensure the consistency of the overall representation.

P-scene-graphs are further extended by introducing a multiscale organization in the structural nodes, which makes it possible to augment the multiscale tree graphs (MTG) used in plant modelling [58] with graphical informations [61]. Such multiscale graphs correspond to recursively quotiented tree graphs. It is possible to define multiscale organization from a simple detailed tree graph

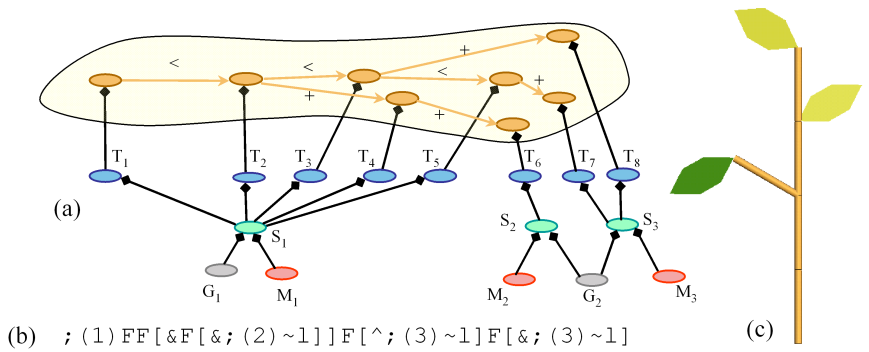


Figure 10: A p-scene-graph. (a) The scene-graph with structural nodes in orange, transformation in blue, shape in green, appearance in red and geometry in grey. (b) The corresponding L-systems bracketed string from which it has been constructed. The '`F`' symbols are geometrically interpreted as cylinders, '`~l`' as leaf symbols, '`&`' and '`^`' as orientation operations and '`;`' as color specifications, [59]. (c) The corresponding geometric model.

(namely the support graph) by specifying quotient functions that will partition the nodes into groups corresponding to macroscopic components in the MTG.

The different scales included in the multiscale p-scene-graph give different views with different resolutions of the same plant object. Since they correspond to different views of the same reality, the associated geometric models must respect particular coherence constraints. For this, a set of *between-scale constraints* is defined that relate the model parameters at one scale with the model parameters at other scales. Between-scale constraints may specify for instance that all the components of a branching system must be included in some macroscopic envelope. These constraints may be either used in a bottom-up or top-down fashion. In top-down approaches, macroscopic representations may be used to bound the development of a plant at a more microscopic level. Such a strategy was used for instance in [29] for the design of bonsai tree using L-systems. In bottom-up approaches, a detailed representations of a plant is used to compute a more macroscopic representation. For this, a set of fitting algorithms has been implemented in `PlantGL` that makes it possible to compute the bounding envelope of a set of geometric primitives using for instance convex hulls [38] or minimal bounding sphere [39], *etc.* These envelope representations can thus be used to characterized globally the geometry of a branching system at different scales, for instance for computing the fractal dimension of a plant [14].

3.3.2 Construction of branching structure models

P-scene-graphs can be created either by generative procedures written in Python or by importing plant structures from other plant modeling softwares.

From a generative perspective, we particularly emphasized in the current version of `PlantGL` the connection with `L-studio/VLab` [5], a widely used L-system based modelling framework for plant growth modelling. A L-system [1] is a particular type of rewritting system that may be used to simulate the develop-

ment of a tree like structure. In this framework, the tree structure is encoded as a bracketed string of symbols called modules that represent components of the structure. To represent attributes, these modules may bear parameters. Particular bracket symbols mark the beginning and the end of branches. The plant growth is formalized by a set of production rules that describes the change over time of the string modules. Starting from an initial string, namely the axiom, modules of the string are replaced according to appropriate rules. A derivation step corresponds to the rewriting in parallel of every modules of the string. Therefore, the development of a tree structure throughout time is modelled by a series of derivation steps. To associate a geometric interpretation with the L-system's output string, some modules are given a graphical meaning and a LOGO-style turtle is used to interpret them, [62]. For this, the string is scanned sequentially from left to right and particular modules are interpreted as actions for the turtle. The turtle state is characterized by a position, a reference frame and additional graphical attributes. Some modules make the turtle move forward and draw graphical elements (cylinders, *etc.*) or change its orientation in space. A stack mechanism makes it possible to store the turtle state at the beginning of a branch and to restore it at the end.

In order to interface `L-studio/VLab` with `PlantGL`, import procedures have been implemented in PlantGL. The strings representing the branching systems generated with cpfg are stored in text files. These strings are then imported into PlantGL with the dedicated primitive `Lstring`. To interpret the L-system modules as commands for the creation of a p-scene-graph, a particular turtle has been implemented in PlantGL that follows the `Lstudio/VLab` specification [1] (Figure 10). Since the p-scene-graph is accessible in Python, it is then possible to interactively explore and analyse the resulting geometric structure with the set of algorithms available in PlantGL for the manipulation of a scene-graph or for the analysis of a plant topological structure using other toolkits such as `AMAPmod/VPlants` [63]. The following code sketches the import of a L-system string in `PlantGL`, its conversion into a scene-graph and some basic manipulation of the result such as display and wood volume computation.

```
from PlantGL import *
lstring = Lstring('plant.str')
turtle = PglTurtle()
lstring.apply(turtle)
sg = turtle.getSceneGraph()
Viewer.display(sg)
vol = volume(sg)
```

A more complex example of such a coupling between `Lstudio/cpfg` and `PlantGL` is illustrated in section 4.3.3. It uses PlantGL's hulls defined in section 3.2.1 to constrain L-systems generation of branching structure. Other connections with modelling platforms such as `AMAPmod/VPlants` [60], `VegeSTAR` [64] are also available and make it possible to create 3D plant models from measured data (see Figure 11).

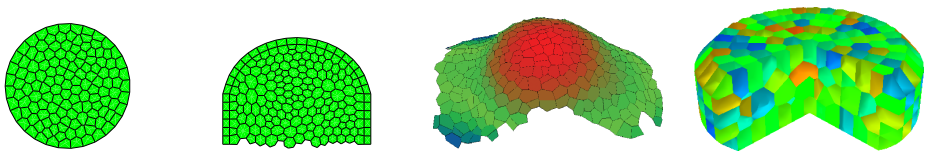
3.4 Tissue models

In `PlantGL`, a plant tissue is considered as a collection of connected regions. A region may represent either a single cell or a set of cells. Unlike branching systems, the neighborhood relationship between regions cannot be simply

(a) (b) (c)

Figure 11: Example of branching systems in `PlantGL`. (a) A beech tree simulated with an L-system using `Lstudio/VLab` and imported in `PlantGL`. This model is used in the application presented in section 4.3.3. (b) Black tupelo tree generated procedurally in `Python` using `PlantGL` according to the generative procedure proposed by Weber and Penn in [3]. and (c) the root system of an oak tree [12].

represented by tree graphs since the connection networks between regions usually contain cycles. To model correctly the neighborhood relationship between regions, we need to take into account the hierarchical organization of region connections: for example, in 3 dimensions, two 3-D cells are connected through a 2-D wall, two walls are connected through a 1-D edge and two edges are connected through a 0-D vertex. More generally, the connection between two or more elements of dimension $n+1$ is an element of dimension n. Such a hierarchical organization defines an *abstract simplicial complex* [65]. Similarly to *p-scene-graphs* for branching systems, scene-graphs representing tissues, called *t-scene-graphs*, are defined by augmenting simplicial complexes representing cell networks with geometrical properties. Each structural node of the simplicial complex is associated with transformation, geometry and appearance nodes.



a b c d

Figure 12: Tissue models.
(a) 2D tissue, (b) 2D transversal cut, (c) 3D surface tissue from [32], (d) 3D tissue

A set of geometric algorithms have been designed to simplify the manipulation of the *t-scene-graphs* during simulations.

- A first algorithm makes it possible to define the geometry of an element of dimension $n+1$ from the geometric information of its components of dimension n. For example, the polyhedral geometry of a cell is derived from the polygonal geometry of its walls. The overall consistency of the geometry of all elements in the tissue is thus ensured by specifying only the geometry of its simplicial (smallest) elements.
- A second algorithm implements cell division. A cell (or more generally a region) is divided in two daughter cells. The geometry of these daughter cells may be specified by the user using standard `PlantGL` algorithms (that compute main axis, shape volume or surface, shape orientation, ...) to reflect the biological characteristics of cell division geometry (main orientation of the cell, smallest separation wall, orthogonality between walls, ...).
- A third algorithm has been designed in order to mesh tissue models. Such meshing operation discretizes *t-scene-graphs* into triangles elements. Resulting meshes can be used either to visually display the tissue or in conjunction with finite elements methods to solve differential equations representing physiological processes (diffusion, reaction, transport,...) or mechanical stresses for example.

T-scene-graphs can be obtained either from a file, from images of biological tissues [32], or using procedural algorithms. `PlantGL` provides a set of procedural algorithms that generate regular, grid-based tissues (based on rectangular or hexagonal grids) and non-regular tissues containing cells with random sizes. Random tissues are generated using a randomly placed set of points representing cell centers. The Delaunay triangulation (2D or 3D) of this set of points is then computed. This is done by using an external computational geometry library, CGAL [66], available in `Python`. Cell neighborhood is defined by this triangulation and walls correspond to its dual representation (Voronoï diagram). These procedural algorithms result in relatively simple tissue structures. More complex *t-scene-graphs* can be obtained by simulation of tissue development. Starting from an initial simple tissue, a growth algorithm modifies the shape of the tissue. A cell is divided each time its volume reaches a given threshold. This combination of growth and division is maintained up to the desired final shape (see 4.3.1 for example).

4 Applications and Illustrations

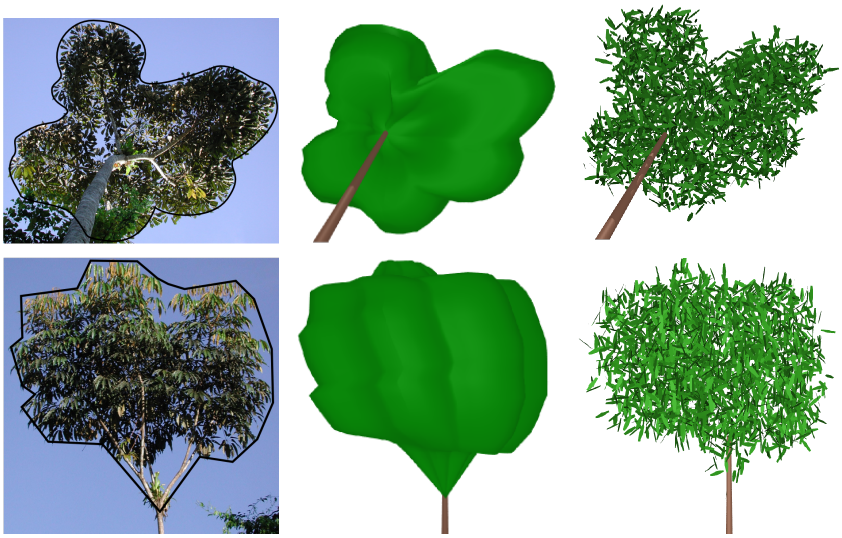
The `PlantGL` library has already been used in a number of modeling applications by our group (e.g. [32, 33, 14, 30, 35]) and other plant research groups (e.g. [31, 34]). The library allows modellers to address graphic and geometric issues in the different phases of a modeling approach, i.e. observation, analysis, simulation and model evaluation. In this section, we aim to illustrate how `PlantGL` provides a set of useful transversal tools to address various questions in these different phases. In particular, we stress the use of envelope- or grid-based approaches which is original in `PlantGL` and opens new application areas. In these applications, we illustrate how `PlantGL` can be assembled with other `Python` libraries to achieve high-level operations on plant structures, thus opening the way to the definition of a powerful plant modeling platform.

4.1 Plant Canopy Reconstruction

In plant modeling, 3D digitizing of plant structure has become a topic of increasing importance in the last decade. Various methodologies have been used to digitize plants at different levels of detail [67] for leaves, [68, 60] for branching systems, and also [52, 6, 48] for tree crowns. Among these approaches, the reconstruction of 3D models of large/tall trees (like trees of tropical forest for example) remains a challenging problem. This is mainly due to the difficulty of acquiring information in the field, and to capture the intricate structure of such plants. In this section, we show how the new envelope-based tools provided in `PlantGL` can be used in this aim.

4.1.1 Using PlantGL to build-up large crowns

First approaches attempted to use parametric models to estimate the geometry of tree crows in the context of light modeling in plant stands [9]. More recently, non-parametric visual hulls have been used in different application contexts to characterize the plant volume based on photographs [6, 49, 48].



(a) (b) (c)

Figure 13: Reconstruction of the crown envelope of a *Schefflera Decandra* tree with an extruded hull build from two photographs. Sketches are done with an internal curve editor. Photo courtesy of Y. Caraglio.

`PlantGL` makes it possible to easily combine these approaches by using for example photographs and parametric envelope models to estimate plant canopy volumes.

Based on a set of images of a tree (being either photographs or botanical drawings) with known camera positions and orientations, the modeler must first choose a parametric envelope models. Then, (s)he defines a number of crown profiles. For the extruded hull for instance, two closed curves that encompass the entire crown in vertical and horizontal planes are required. This step is

usually made by manually outlining the foliaged tree region using an internal curve editor. Profiles are then used to build the tree silhouette hull.

From this estimated crown envelope, the modeler can then infer a detailed crown model by making new assumptions about the leaf distribution inside the crown. Figure 13 illustrates the use of a simple uniform random distribution of leaf positions and orientations. The following code shows how the example for random foliage generation of section 3.2.2 can be adapted to take into account complex crown shapes.

```
sc = Scene()
for i in range(nbleaves):
  pos = (uniform(xmin,xmax),uniform(ymin,ymax),
                uniform(zmin,zmax))
  if inside(hull,pos) :
    sc += Translated(pos,leaf_symbol))
```

Using `Python`, it is possible to define foliage distribution using more complex algorithms, such as those described in [5, 6, 29].

4.1.2 Using PlantGL to assess plant mock-up accuracy

Another important problem in canopy reconstruction is to assess the accuracy of 3D plant mockups obtained from measurements. A family of solutions consists of comparing equivalent synthesized descriptions of both the real and the virtual plants. In this family, hemispherical views are particularly interesting since they directly measure a physical characteristic of the plant, namely the amount of intercepted light.

Figure 14 illustrates this approach [69]. An hemispheric picture is taken from the real plant, while an equivalent virtual picture is computed with the same camera position on the reconstructed plant. White areas in both pictures directly reflect the amount of light that reach different positions under or inside the crown [70]. The amount of intercepted light is summarized with the canopy openness index defined as the ratio between non intercepted pixels and total number of pixels on the picture.

4.2 Analysis of Plant Geometry

Plant geometry is a parameter of paramount importance in the modeling of plant-environment interaction. However, plants usually show complex geometric shapes with numerous components, highly organized but with non-deterministic structure. Characterizing this "irregularity" of plant shapes with few high level parameters is thus a determinant issue of modeling approaches. In many applications in forestry, horticulture, botany or eco-physiology, analysis of plant structures are carried out to find out adequate ways of capturing their intricate geometry in simple models. In this section, we illustrate the use of grids and envelopes defined in `PlantGL` in order to achieve such analysis.

4.2.1 Grid-Based Analysis

Fractal geometry was introduced to analyse the geometry of markedly irregular structures that can be either mathematically constructed or found in nature [72]. Several parameters have been introduced for this purpose, such as fractal dimension and lacunarity. These parameters are intended to capture the essence

Figure 14: Assessment of plant model reconstruction using hemispheric view [69]. The *Juglans nigra x Juglans regia* hybrid walnut plant model is reconstructed from partial measurements: wood structure is manually digitized while leaves are generated from distribution functions. On the left, an hemispheric photograph of a real walnut from the ground. On the right, reconstructed mockup using `AMAPmod/VPlants` exported in the `Pov-Ray` [71] to compute an hemispheric picture at the same position. Here, the evaluated canopy openness from real photograph is of 40 % and 49 % for the virtual tree.

of irregularity, i.e. the way these structures physically occupy space as resolution decreases. Several estimators of these parameters exist. They consist of paving the original object in different manners with tiles of different sizes and study the variation of the number of tiles with tile size.

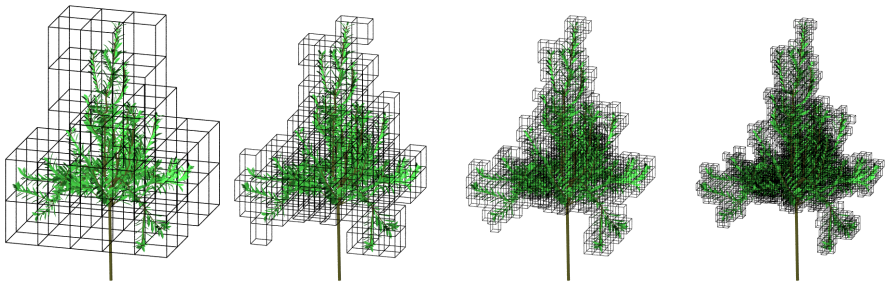


Figure 15: The box counting method applied to the foliage of a digitized apple tree [73]. The tree is 2m height with around 2500 leaves. Global bounding box has a volume of 10 m^3 and serve as initial voxel. A grid sequence is then created by subdividing uniformly this bounding box into sub-voxels. At each scale, intercepted voxels are counted to determine fractal dimension (here of the order of 2.1).

For plant structures, fractal properties, such as fractal dimension, have been computed in different contexts, e.g. [74]. They frequently rely on the fractal analysis of 2D photographs. However, more recently, several works showed the possibility to compute more accurate 3D-estimators using detailed 3D-digitized plant mock-ups of real plants [75, 14, 33].

`PlantGL` makes it possible to carry out such computation in a flexible way. For example, to implement the box-counting estimator of the fractal dimension [72], the `PlantGL` "grid" object can be used to count the number of 3D cells

of a given size containing vegetation. If $N(\delta)$ denotes the number of occupied 3D cells of size δ, the box-counting estimator of the fractal dimension D_δ of the object is defined as:

$$D_\delta = \lim_{\delta \to 0} \frac{\ln N(\delta)}{\ln \frac{1}{\delta}}. \tag{11}$$

D_δ is estimated from the slope of the regression between $\ln N(\delta)$ and $\ln \frac{1}{\delta}$ values. The following code sketches the implementation of such an estimator using `Python` and `PlantGL`.

```
from scipy import stats
def boxcounting(scene,maxdivision):
  grid = Grid(scene,maxdivision)
  voxels = [log(grid.intercepted_voxels(div))
            for div in range(maxdivision)]
  delta=[log(1./i) for i in range(maxdivision)]
  slope,itcept,r,ttp,stderr=stats.linregress(voxels,delta)
  return slope # slope of the regression
```

Figure 15 illustrates the application of the box counting method on the foliage of a 3D-digitized apple tree [73]. `PlantGL` can be used in a similar way to compute various other fractal properties of plants [14, 33]

4.2.2 Envelope-Based Analysis

Parametric envelopes provided in `PlantGL` can also be used to analyse volumetric properties of plant crowns. For example, in order to quantify the development of a plant crown over time, envelopes can be adjusted to the crown of the developing tree at different ages and their surface or volume can then be estimated.

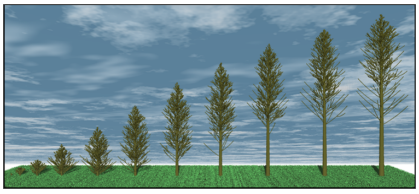
Figure 16 illustrates this approach together with the possibility to import plant data from other softwares. The growth of an eucalyptus was simulated at various ages using the `AMAPsim` software [22], Figure 16.a. Results were imported in `PlantGL` as MTGs. The convex hull of the plant crown was then computed at each age using the fitting algorithms provided by `PlantGL` for envelopes as described in [14]. Here is how this series of steps can be carried out in `PlantGL`:

```
import pylab
def hullanalysis(ages,trees):
  hulls=[fit('convexhull',i) for i in trees]
  wood_sf=[surface(i) for i in trees]
  hull_sf=[surface(i) for i in hulls]
  delta_wood_sf=[wood_sf[i+1]-wood_sf[i] for i in range(len(wood_sf)-1)]
  delta_hull_sf=[hull_sf[i+1]-hull_sf[i] for i in range(len(hull_sf)-1)]
  pylab.plot(ages[1:],delta_wood_sf)
  pylab.plot(ages[1:],delta_hull_sf)
```

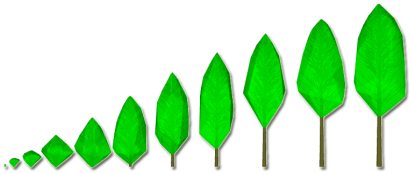
Based on such data, various investigations about the crown development can be made. Curves showing the variation of crown surface/volume throughout time can be analysed, comparison at a more microscopic scale with the leaf area variation can be made 16.c, etc.

4.3 Simulations based on plant geometric models

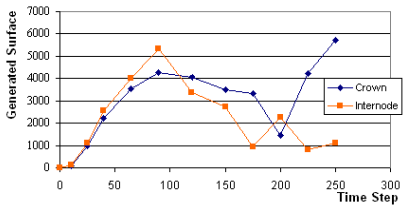
Using flexible geometric models of plant is not restricted to the analysis of plant structure. They can be used as well for the simulation of various physical or



(a)



(b)



(c)

Figure 16: (a) An eucalyptus trees simulated at various age (from 1 to 8 months on a scale from 10 to 250) with the `AMAPsim` software, exported in PlantGL and rendered with the `Pov-Ray` software [71] (b) Global representations of eucalyptus crown at the various ages (c) Increments of wood and hull surfaces in time. The different degrees of correlation and their associated time period enable us to identify the various phase of the crown development.

physiological processes that take place *within* or *in interaction with* the plant structure. Here, we present three applications that demonstrate the use of `PlantGL` at different scales, ranging from organ to communities.

4.3.1 Simulation at organ scale

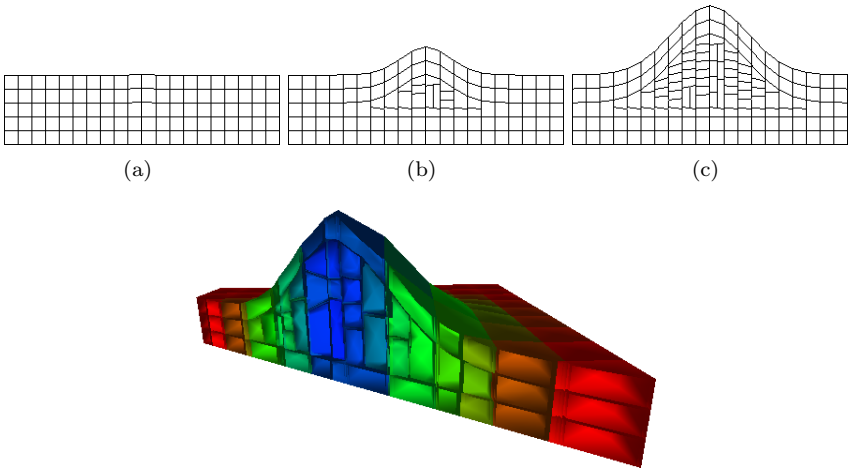
Due to the recent advances in plant cellular and developmental biology, the modeling of plant organ development is considered with a growing interest by the plant research community: leaf [76, 77, 78], shoot apical meristem [79, 80, 81, 82, 83], flower [84]. PlantGL provides flexible data structures and algorithms that make it possible to develop 2D or 3D simulations of tissue development.

As a matter of illustration, let us consider for instance a tissue whose vertices are submitted to a known velocity field (coming from kinetic observations for example). Due to the velocity field and to boundary conditions, the speed of the external vertices of the tissue is known. The speed vector of the internal vertices is then interpolated from these known values. At each time t of the simulation, each vertex is then moved according to this speed vector. This process progressively modifies the cell size, and consequently the overall tissue shape. During the growth, if a cell has a surface (or volume in 3D) that reaches a predefined threshold, it divides in two children cells. Different algorithms implementing cell division are available on a tissue object, *e.g.* [85]. Here follows the code of such a tissue growth in `PlantGL` for a particular choice of the cell division algorithm (results are presented on figure 17):

```
tissue = createGridTissue( (20,5) )
def speed_field (pos, t) :
```

```
    yspeed=a*[1+b*t*pos.x^2/(h+b*t)^2]*exp(-pos.x^2/(h+b*t))
    return Vector(0,yspeed)

for t in range(time_begin,time_end,delta_time):
    for pos in tissue.positions() :
        pos+=speed_field(pos,t)*delta_time
    for cell in tissue :
        if cell.size() > max_cell_size:
            cellDivide(cell,algo=MAIN_AXIS)
```



(a) (b) (c) (d)

Figure 17: (a,b,c) 2D geometrical representation of a growing tissue at three different times. (d) 3D simulation of bump formation on a tissue

4.3.2 Simulation at plant scale

In biological applications, virtual plants are frequently used to carry out virtual experiments where data is difficult to measure or when the interaction between the studied processes is too complex. This is particularly true for the study of light interception by plants: light cannot be measured in a real canopy with high accuracy and the amount of light rays that can go through a canopy is a complex function of the tree architecture. The following example illustrates the use of `PlantGL` in the context of model assessment and how high-level geometric operations used in light interception models can be simply performed with `PlantGL`.

The STAR (Surface to Total Area Ratio) is a key eco-physiological parameter used in light interception models [86]. It is a directional quantity defined by ratioing the surface of the projection of a tree foliage S_Ω in a particular direction Ω to its total leaf surface S.

$$STAR_\Omega = \frac{S_\Omega}{S} \tag{12}$$

This directional index can be integrated over all the sky vault to characterize the overall light interception of a tree.

However, since the total leaf area of a real plant is often expensive to measure, approximate values of the STAR are often used in place of the exact one in eco-physiological applications. For this, the directional STAR is estimated from simple measures of the plant volume and leaf density and by making simplifying assumptions on the actual spatial distribution of leaves in the canopy [87]. In this case, the plant is supposed to be an homogeneous volume with small leaves uniformly distributed within the crown looking like a "turbid medium". In this context, a light beam b of direction Ω_b has a probability $p_0(b)$ to be intercepted :

$$p_0(b) = exp\,(-G_{\Omega_b}.LAD.l_b) \tag{13}$$

where G_{Ω_b} is a coefficient characterizing the spatial distribution of leaf orientations in the crown volume, LAD is the Leaf Area Density in the volume and l_b the length of the beam path in the crown volume. Assuming the B beams constitute a regular and dense sampling of the whole volume, the approximated directional STAR of the turbid volume, $\widehat{STAR}_\Omega$, can then be computed as [88]:

$$\widehat{STAR}_\Omega = \sum_{b=1}^{B} S_b(1 - p_0(b))/S \tag{14}$$

where S_b is the cross section area of a beam. This model-based definition of the STAR can be compared to the above real STAR to evaluate the quality of light model assumptions. The resulting difference characterizes the error due to the model underlying hypotheses (homogeneity/randomness of the foliage distribution, negligibility of leaf size, ...) with respect to the actual canopies [87].

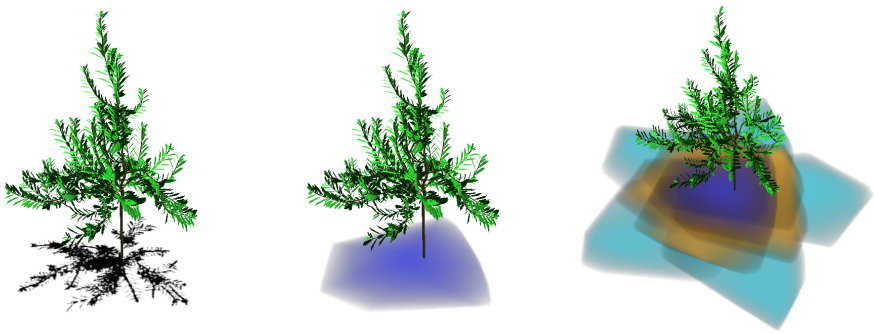
In `PlantGL`, both STAR quantities, *i.e.* the real and approximated STARs, can be computed from a plant mockup using the library high-level functions. The real STAR of a given virtual canopy can be computed by counting the number of vegetation pixels in a virtual picture obtained by projecting virtual plant canopies using an orthographic camera [87] and multiplying by the size of a pixel. This would be expressed as follows in `PlantGL`:

```
def star(leaves,dir):
  Viewer.display(leaves)
  Viewer.camera.setOrthographic()
  Viewer.camera.setDirection(dir)
  proj, nbpixel, pixelsize = Viewer.frameGL.getProjectionSize()
  return proj / surface(leaves)
```

For the approximated STAR, the envelope of the tree crown must first be computed. Then, a set of beams of direction Ω are cast and their interceptions and resulting length in the crown volume are computed. A sketch of such a code would be as follows:

```
def star(leaves,g,dir,up,right,beam_radius):
  hull= fit('convexhull',leaves)
  lad = surface(leaves) / volume(hull)
  bbx = BoundingBox(hull,dir,up,right)
  Viewer.display(hull)
  pos = bbx.upperRightCorner()
  interception = 0
  for rshift in range(bbx.size().y/beam_radius):
        for upshift in range(bbx.size().z/beam_radius):
```

```
                ray = Viewer.castRay(pos-rshift*right-upshift*up,dir)
                p0 = 1
                for intersection in ray:
                        length = norm(intersection[1]-intersection[0])
                        p0 *= exp(-g*lad*length)
                interception += (1-p0)*(beam_radius**2)
return interception / surface(leaves)
```



(a) (b) (c)

Figure 18: Light intercepted by an apple tree represented as shadow on the ground. Intensity of the colors represent intensity of interception. STAR can be computed as a ratio between the surface of the shadow and the plant leaf surface (a) Light intercepted from top direction using ray casting (b) Light intercepted from same direction using Beer-Lambert hypothesis. (c) Light interception sampled from different directions. The different colors are used to mark the difference between various elevations of ray direction.

4.3.3 Simulation at community scale

Detailed plant models, at the level of branches and leaves, do not always correspond to the most adequate level for expressing knowledge in plant models. `PlantGL` provides a number of means to deal with abstract representation of plants at different scales. In particular, the various envelope models defined in section 3.2.1 can be used as abstract means to model plant crown bulk. Such models are useful for instance in the modeling of plant communities, where competition for space has been shown to be a key structuring factor [89].

In the following example, we illustrate how natural scenes containing thousands of plants distributed in a realistic manner can be built with `PlantGL`, taking into account competition for space. It is inspired by [30] which is an extension of [90, 91] to the use of more complex crown shapes.

The ecosystem synthesis starts with the generation of a set of coarse individuals with height, crown radius and crown base height determined from density and allometric functions.

Individuals are *fagus* beech trees with different classes of ages. Allometric functions of the *Fagacées* model [92] are used to determine the heights and radius values as a function of tree age. The spatial distribution of these plants is generated using a stochastic point process. For this, we use a Gibbs process

[93, 94] defined as a pairwise interaction function $f(p_i, p_j)$, that represents the cost associated with the presence of two given plants at positions p_i and p_j respectively. Positive cost values will lead to repulsion between trees while negative ones lead to attraction. A realization of this process is intended to minimize of the global cost $F = \sum_{i \neq j} f(p_i, p_j)$, defined as the sum of the costs associated with each pair of points. The Gibbs process is simulated with a classical depletion-replacement iterative algorithm [95].

Classically, the cost function is used to model neighbor competition and is defined as a function of the crown radii and positions of the trees. The cost function of two trees i and j, characterized by shapes with constant radius, may be chosen for instance proportional to the difference between the sum of the crown radii and the distance between p_i and p_j. For asymmetric shapes, the same function can be used where radii of trees now correspond to the radius of each envelope in the direction defined by the tree positions p_i and p_j. In addition, both the position and the different parameters of the crown envelope can now be changed in the depletion-replacement algorithm. Figure 19 illustrates the 3D output of such a process.

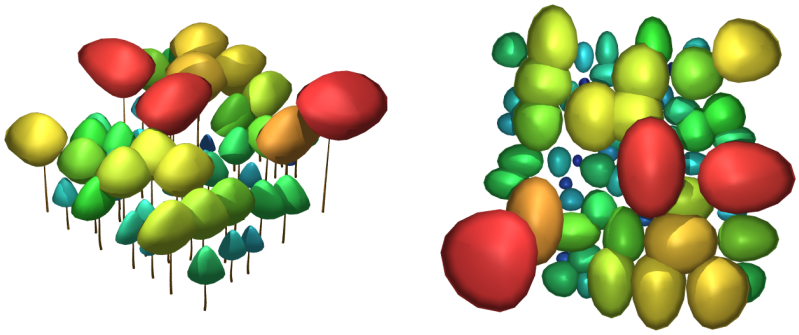


Figure 19: A front and top view of a generated stand at the crown scale. Different colors are used to differentiate various layers of vegetation.

From this set of coarse individuals, detailed plant representations can be inferred and assembled into a complete scene. For this, different generation methods either available in `PlantGL` or outside of the software can be used. In our example, we generated the beech trees using the L-systems models using generative procedure described in [29].

Bushes and flowers were generated using `PlantGL` and `Python` as presented in section 3.3 and added to the scene. Finally, a digitized walnut tree [68] was also added to illustrate how scenes may be created in `PlantGL` using a range of classical date sources.

The final rendering was made with `Povray` [71]. Each plant geometric models were thus converted and assembled in this format. Figure 20 illustrates the resulting scene.

5 Conclusion

In this paper, we presented a new open-software library for the geometric modeling of plants built on the top of the `Python` programming language. The

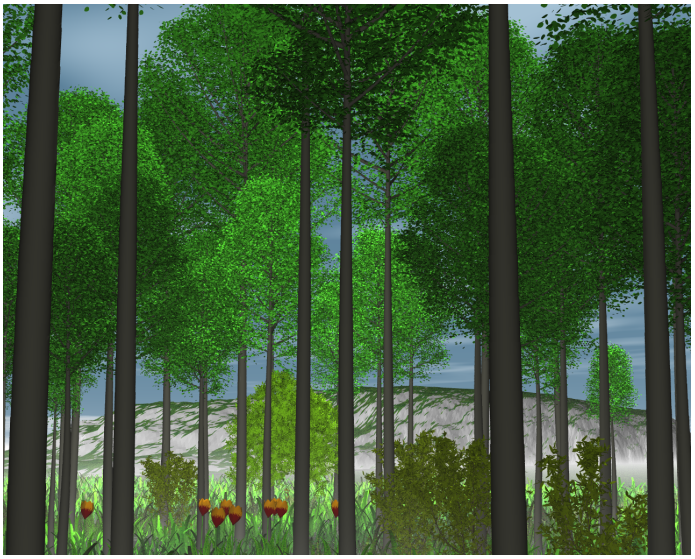


Figure 20: A community of plants generated from a Gibbs process [30]. The scene is made of trees coming from different sources: beech trees of different sizes and ages where generated using the ecosystem model presented in section 4.3.3 which were built using the interaction between `PlantGL` and LStudio/Vlab, A walnut tree corresponding to a 3D digitizing of a real plant built using AMAPmod/VPlants, virtual bushes flowers and grass created procedurally in `Python` with `PlantGL`

library provides a set of geometric models that are useful to represent various types of plant structures at different scales, ranging from tissues to plant communities. In particular, it contains original geometric components such as dedicated parametric envelopes for crown shape representation and tissues for representing plants at cell scale in 2D or 3D. Branching systems can be created either procedurally or by importing them from plant growth simulation platforms, such as `LStudio/VLab`. The resulting plant geometric models can be easily analysed using `Python` and `PlantGL` high level algorithms. The different features of the `PlantGL` library have been illustrated on applications involving plants at different scales and showing its use at various stages of a modeling process.

Acknowledgments.

The authors thank P. Prusinkiewicz for making the `LStudio/VLab` software kindly available to them, D. Da Silva, P. Barbier de Reuille and Y. Caraglio for their contribution to some images, and H. Sinoquet, E. Costes, F. Danjon, C.-E. Parveau and J. Traas for making 3D digitized plants or tissues available to them. This work has been partially supported by ANR projects *NatSim* and *Virtual Carpel*.

References

[1] P. Prusinkiewicz, A. Lindenmayer, The algorithmic beauty of plants, Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[2] P. de Reffye, C. Edelin, J. Françon, M. Jaeger, C. Puech, Plant models faithful to botanical structure and developmentr, in: SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques, New York, NY, USA, 1988, pp. 151–158.

[3] J. Weber, J. Penn, Creation and rendering of realistic trees, in: SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques, ACM Press, New York, NY, USA, 1995, pp. 119–128.

[4] B. Lintermann, O. Deussen, Interactive modeling of plants, Computer Graphics and Applications, IEEE 19 (1) (1999) 56–65.

[5] P. Prusinkiewicz, L. Mündermann, R. Karwowski, B. Lane, The use of positional information in the modeling of plants, in: SIGGRAPH'01, Computer Graphics, ACM, Los Angeles, California, 2001, pp. 36–47.

[6] I. Shlyakhter, M. Rozenoer, J. Dorsey, S. Teller, Reconstructing 3d tree models from instrumented photographs, IEEE Comput. Graph. Appl. 21 (3) (2001) 53–61.

[7] N. B., T. Franken, O. Deussen, Approximate image-based tree-modeling using particle flows, ACM Transactions on Graphics (Proc. of SIGGRAPH 2007) 26 (3).

[8] P. Tan, G. Zeng, J. Wang, S.-B. Kang, L. Quan, Image-based tree modeling, ACM Transactions on Graphics (Proc. of SIGGRAPH 2007) 26 (3).

[9] A. Cescatti, Modelling the radiative transfer in discontinuous canopies of asymmetric crown. I. model structure and algorithms, Ecological Modelling 101 (1997) 263–274.

[10] H. Sinoquet, P. Rivet, C. Godin, Assessment of the three-dimensional architecture of walnut trees using digitising, Silva Fennica 31 (3) (1997) 265–273.

[11] C. Godin, Representing and encoding plant architecture: a review, Annals of Forest Science 57 (05-juin) (2000) 413–438.

[12] F. Danjon, H. Sinoquet, C. Godin, F. Colin, M. Drexhage, Characterisation of structural tree root architecture using 3d digitising and amapmod software, Plant and Soil 211 (2) (1999) 241–258.

[13] J. B. Evers, J. Vos, C. Fournier, B. Andrieu, M. Chelle, P. C. Struik, Towards a generic architectural model of tillering in gramineae, as exemplified by spring wheat (triticum aestivum)., New Phytol 166 (3) (2005) 801–812.

[14] F. Boudon, C. Godin, C. Pradal, O. Puech, H. Sinoquet, Estimating the fractal dimension of plants using the two-surface method. an analysis based on 3d-digitized tree foliage, Fractals 14 (3) (2006) 149–163.

[15] R.-S. Smith, C. Kuhlemeier, P. Prusinkiewicz, Inhibition fields for phyllotactic pattern formation: a simulation study, Canadian Journal of Botany 84 (2006) 1635–1649.

[16] A. Lindenmayer, Mathematical models for cellular interactions in development, I & II, Journal of Theoretical Biology (1968) 280–315.

[17] R. Karwowski, P. Prusinkiewicz, Design and implementation of the l+c modeling language, Electronic Notes in Theoretical Computer Science 86.

[18] P. Prusinkiewicz, R. Karwowski, B. Lane, The l+c plant modeling language, in: J. V. et al. (Ed.), Functional-Structural Plant Modelling in Crop Production, Springer, 2007, p. in press.

[19] W. Kurth, Growth grammar interpreter grogra 2.4: A software for the 3-dimentional interpretation of stochastic, sensitive growth grammar in the context of plant modelling, Introduction and reference manual, Forschungszentrum Waldokosysteme der Universitat Gottingen (1994).

[20] O. Kniemeyer, G.-H. Buck-Sorlin, K. W., A graph grammar approach to artificial life, Artificial Life 10 (4) (2004) 413–431.

[21] G.-H. Buck-Sorlin, O. Kniemeyer, W. Kurth, Barley morphology, genetics and hormonal regulation of internode elongation modelled by a relational growth grammar, New Phytologist 10 (4) (2005) 413–431.

[22] J.-F. Barczi, P. de Reffye, Y. Caraglio, Essai sur l'identifcation et la mise en oeuvre des paramétres nécessaires à la simulation d'une architecture végétale : le logiciel amapsim., in: J. Bouchon, P. de Reffye, D. Barthélémy (Eds.), Modélisation et Simulation de l'Architecture des Végétaux, INRA Editions, 1997, pp. 205 – 254.

[23] O. Deussen, B. Lintermann, Digital Design of Nature. Computer Generated Plants and Organics., Springer-Verlag, 2005.

[24] C. Godin, E. Costes, Y. Caraglio, Exploring plant topological structure with the amapmod software: an outline, Silva Fennica 31 (1997) 355–366.

[25] C. Godin, E. Costes, H. Sinoquet, A method for describing plant architecture which integrates topology and geometry, Annals of Botany 84 (1999) 343–357.

[26] E. Costes, H. Sinoquet, J.-J. Kelner, C. Godin, Exploring within-tree architectural development of two apple tree cultivars over 6 years, Annals of Botany 91 (2003) 91–104.

[27] P. Ferraro, C. Godin, P. Prusinkiewicz, Toward a quantification of self-similarity in plants, Fractals 13 (2) (2005) 91–109.

[28] Y. Guédon, Y. Caraglio, P. Heuret, E. Lebarbier, C. Meredieu, Analyzing growth components in trees, Journal of Theoretical Biology.

[29] F. Boudon, P. Prusinkiewicz, C. Federl, P.and Godin, R. Karwowski, Interactive design of bonsai tree models, Computer Graphics Forum (Proc. of Eurographics '03) 22 (3) (2003) 591–591.

[30] F. Boudon, G. Le Moguedec, Déformation asymétrique de houppiers pour la génération de représentations paysagères réalistes, Revue Electronique Francophone d'Informatique Graphique (REFIG) 1 (1).

[31] F. Danjon, T. Fourcaud, D. Bert, Root architecture and wind-firmness of mature pinus pinaster., New Phytol 168 (2) (2005) 387–400.

[32] P. Barbier de Reuille, I. Bohn-Courseau, C. Godin, J. Traas, A protocol to analyse cellular dynamics during plant development, The Plant Journal 44 (2005) 1045–1053.

[33] D. Da Silva, F. Boudon, C. Godin, O. Puech, C. Smith, H. Sinoquet, A critical appraisal of the box counting method to assess the fractal dimension of tree crowns., Lecture Notes in Computer Sciences (Proceedings of the 2nd International Symposium on Visual Computing).

[34] G. Louarn, Y. Guédon, J. Lecoeur, E. Lebon, Quantitative analysis of the phenotypic variability of shoot architecture in two grapevine cultivars (vitis vinifera l.)., Annals of Botany 99 (3) (2007) 425–437.

[35] J. Chopard, C. Godin, J. Traas, Toward a formal expression of morphogenesis: a mechanics based integration of cell growth at tissue scale, in: Proceedings of the 7th International Workshop on Information Processing in Cell And Tissues, Oxford, UK, 2007, p. in press.

[36] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[37] P. S. Strauss, R. Carey, An object-oriented 3d graphics toolkit, in: SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques, ACM Press, New York, NY, USA, 1992, pp. 341–349.

[38] C. B. Barber, D. P. Dobkin, H. Huhdanpaa, The quickhull algorithm for convex hulls, ACM Trans. Math. Softw. 22 (4) (1996) 469–483.

[39] E. Welzl, Smallest enclosing disks (balls and ellipses), New Results and New Trends in Computer Science 555 (1991) 359–370.

[40] E. Andres, P. Nehlig, J. Françon, Supercover of straight lines, planes and triangles, in: DGCI '97: Proceedings of the 7th International Workshop on Discrete Geometry for Computer Imagery, Springer-Verlag, London, UK, 1997, pp. 243–254.

[41] R. Carey, G. Bell, The Annotated VRML 2.0 Reference Manual, Addison-Wesley, 2002.

[42] F. Boudon, C. Nouguier, C. Godin, Geom module manual. I. user's guide, Document de travail du Programme Modélisation des plantes 3-2001, CIRAD (12/2001 2001).

[43] J.-G. Schneider, O. Nierstrasz, Components, scripts and glue., in: L. Barroca, J. Hall, P. Hall (Eds.), Software Architectures - Advances and Applications, Springer, 1999, Ch. 2, pp. 13–25.

[44] T. E. Oliphant, Python for scientific computing, Computing in Science and Engineering 9 (3) (2007) 10–20.

[45] H. Sowizral, Scene graphs in the new millennium, IEEE Comput. Graph. Appl. 20 (1) (2000) 56–57.

[46] S. Meyers, More Effective C++: 35 New Ways to Improve Your Programs and Designs, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[47] Boost c++ librairies (1998–2006).
URL `http://www.boost.org/`

[48] J. Phattaralerphong, H. Sinoquet, A method for 3d reconstruction of tree crown volume from photographs: assessment with 3d-digitized plants, Tree Physiology 25.

[49] A. Reche-Martinez, I. Martin, G. Drettakis, Volumetric reconstruction and interactive rendering of trees from photographs, ACM Trans. Graph. 23 (3) (2004) 720–727.

[50] H. Horn, The adaptive geometry of trees, Princeton University Press, Princeton, N.J., 1971.

[51] H. Koop, Silvi-star: A comprehensive monitoring system, Forest Dynamics (1989) 229.

[52] P. Birnbaum, Modalités d'occupation de l'espace par les arbres en forêts guyanaise, Master's thesis, Université Paris VI (1997).

[53] C. D. Woodward, Skinning techniques for interactive b-spline surface interpolation, Comput. Aided Des. 20 (10) (1988) 441–451.

[54] L. A. Piegl, W. Tiller, The Nurbs Book, 2nd Edition, Springer, 1997.

[55] L. A. Piegl, W. Tiller, Surface skinning revisited, The Visual Computer 18 (4) (2002) 273–283.

[56] M. Barnsley, Fractals Everywhere, Academic Press, Boston, 1988.

[57] K. Falconer, Techniques in fractal geometry, John Wiley and Sons, 1997.

[58] C. Godin, Y. Caraglio, A multiscale model of plant topological structures, Journal of theoretical biology 191 (1998) 1–46.

[59] R. Mech, M. James, M. Hammel, J. Hanan, P. Prusinkiewicz, CPFG v4.0 user manual, The University of Calgary (2005).

[60] C. Godin, E. Costes, H. Sinoquet, A method for describing plant architecture which integrates topology and geometry, Annals of Botany 84 (1999) 343–357.

[61] F. Boudon, Représentation géométrique multi-échelles de l'architecture des plantes, Ph.D. thesis, Université de Montpellier II (2004).

[62] P. Prusinkiewicz, Graphical applications of l-systems, in: Proceedings on Graphics Interface '86/Vision Interface '86, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 1986, pp. 247–253.

[63] C. Godin, Y. Guédon, E. Costes, Exploration of a plant architecture database with the amamod software illustrated on an apple tree hybrid family, Agronomie 19 (3-4).

[64] B. Adam, N. Dones, H. Sinoquet, Vegestar v.3.1. a software to compute light interception and photosynthesis by 3d plant mock-ups, in: C. Godin (Ed.), Fourth International Workshop on Functional-Structural Plant Models, Montpellier, France, 2004, p. 414.

[65] E. H. Spanier, Algebraic Topology, McGraw-Hill, New York, 1966.

[66] CGAL, Computational Geometry Algorithms Library.
URL `http://www.cgal.org`

[67] H. Sinoquet, B. Andrieu, The geometrical structure of plant canopies: characterization and direct measurements methods, in: C. Varlet-Grancher, R. Bonhomme, H. Sinoquet (Eds.), Crop structure and light microclimate, Vol. 0, INRA Editions, Paris, 1993, pp. 131–158.

[68] H. Sinoquet, P. Rivet, C. Godin, Assessment of the three-dimensional architecture of walnut trees using digitizing, Silva Fennica 3 (1997) 265–273.

[69] C.-E. Parveaud, Propriétés radiatives des couronnes de noyers (*Juglans nigra x J. regia*) et croissance des pousses annuelles - influence de la géométrie du feuillage, de la position des pousses et de leur climat radiatif, Ph.D. thesis, Université de Montpellier II, France (2006).

[70] E. Casella, H. Sinoquet, A method for describing the canopy architecture of coppice poplarwith allometric relationships., Tree Physiology 23 (2003) 1153–1170.

[71] T. P.-R. Team, Persistence of vision raytracer (1991–2006).
URL `http://www.povray.org/`

[72] B. B. Mandelbrot, The fractal geometry of nature, W.N. Freeman, New York, USA, 1983.

[73] E. Costes, H. Sinoquet, C. Godin, J. J. Kelner, 3d digitizing based on tree topology : application to study the variability of apple quality within the canopy, Acta Horticulturae 499 (1999) 271–280.

[74] M. Barnsley, S. Demko, Iterated Function Systems and the Global Construction of Fractals, Royal Society of London Proceedings Series A 399 (1985) 243–275.

[75] A. L. Oppelt, W. Kurth, H. Dzierzon, G. Jentschke, D. L. Godbold, Structure and fractal dimensions of root systems of four co-occurring fruit tree species from *Botswana*, Annals of Forest Science 57 (2000) 463–475.

[76] J. Runions, T. Brach, S. Kuhner, Photoactivation of gfp reveals protein dynamics within the endoplasmic reticulum membrane, Journal of Experimental Botany 57 (1) (2006) 43–50.

[77] F. G. Feugier, Models of vascular pattern formation in leaves, Ph.D. thesis, Pierre et Marie Curie (2005).

[78] A.-G. Rolland-Lagan, E. Coen, S. J. Impey, J. A. Bangham, A computational method for inferring growth parameters and shape changes during development base clonal analysis, Journal of Theoretical Biology 232 (2005) 157–177.

[79] H. Jönsson, M. Heisler, G. V. Reddy, V. Agrawal, V. Gor, B. E. Shapiro, E. Mjolsness, E. M. Meyerowitz, Modeling the organization of the wuschel expression domain in the shoot apical meristem., Bioinformatics 21 Suppl 1.

[80] H. Jönsson, M. G. Hesler, B. E. Shapiro, E. M. Meyerowitz, E. Mjolsness, An auxin-driven polarized transport model for phyllotaxis, PNAS 103 (5) (2006) 1633–1638.

[81] P. Barbier de Reuille, I. Bohn-Courseau, K. Ljung, H. Morin, N. Carraro, C. Godin, J. Traas, Computer simulations reveal properties of the cell-cell signaling network at the shoot apex in arabidopsis, PNAS 103 (5) (2006) 1627–1632.

[82] C. Smith, On vertex-vertex systems and their use in geometric and biological modelling, Ph.D. thesis, University of Calgary (2006).

[83] M. Heisler, H. Jönsson, Modelling meristem development in plants, Current Opinion in Plant Biology 10 (2007) 92–97.

[84] A.-G. Rolland-Lagan, J. A. Bangham, E. Coen, Growth dynamics underlying petal shape and asymmetry, Nature 422 (13) (2003) 161–163.

[85] J. Nakielski, Tensorial model for growth and cell division in the shoot apex, in: A. Carbone, M. Gromov, P. Prusinkiewicz (Eds.), Pattern formation in biology, vision and dynamics, World Scientific, Singapore, 2000, pp. 252–267.

[86] P. Oker-Blom, H. Smolander, The ratio of shoot silhouette area to total needle area in scots pine, Forest Science 34 (1988) 894–906.

[87] H. Sinoquet, G. Sonohat, J. Phattaralerphong, C. Godin, Foliage randomness and light interception in 3-d digitized trees: an analysis from multiscale discretization of the canopy, Plant, Cell and Environment 28 (9) (2005) 1158–1170.

[88] H. Sinoquet, C. Varlet-Granchet, R. Bonhomme, Modelling radiative transfer within homogeneous canopies: basic concepts, in: C. Varlet-Grancher, R. Bonhomme, H. Sinoquet (Eds.), Crop structure and light microclimate, Vol. 0, INRA Editions, Paris, 1993, pp. 131–158.

[89] A. Franc, S. Gourlet-Fleury, N. Picard, Une introduction à la modélisation des forêts hétérogènes, ENGREF, Nancy, 2000.

[90] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, P. Prusinkiewicz, Realistic modeling and rendering of plant ecosystems, Computer Graphics 32 (Annual Conference Series) (1998) 275–286.

[91] B. Lane, P. Prusinkiewicz, Generating spatial distributions for multilevel models of plant communities, in: Proceedings of Graphics Interface 2002, Calgary, Alberta, 2002, pp. 69–80.

[92] G. Le Moguédec, J. Dhôte, Présentation du modèle *Fagacées*, Tech. rep., LERFOB, INRA, Nancy, France (2002).

[93] P. Diggle, Statistical analysis of spatial point patterns, Academic Press, London, UK, 1983.

[94] F. Goreaud, Apport de l'analyse de la structure spatiale en forêt tempérée à l'étude de la modélisation des peuplements complexes, Ph.D. thesis, ENGREF (2004).

[95] B. Rippley, Simulating spatial patterns: dependent samples from a multivariate density, Applied Statistic 28 (1979) 109–112.

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Centre de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399