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

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

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

C. Pradal [a 1], F. Boudon [a,*1], C. Nouguier [a], J. Chopard [b], C. Godin [b]

[a] *CIRAD, Virtual Plants INRIA Project-Team, UMR DAP, Montpellier, F-34398 France.*

[b] *INRIA, Virtual Plants INRIA Project-Team, UMR DAP, Montpellier, F-34398 France.*

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 modeling 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 are 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 ranging from tissues to plant communities.

Key words: Graphic library, Virtual plants, Crown envelopes, Plant architecture, Canopy reconstruction, Plant scene-graphs

[1] These two authors contributed equally to the paper

* Corresponding author.
Email address: `frederic.boudon@cirad.fr` (F. Boudon).

1 Introduction

The representation of plant forms in computer scenes has 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. [1–9]. 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. [10–16]. 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 [17,1,3]. 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 [19]. An alternative implementation of a L-system-based software for plant modeling was designed by W. Kurth [20] 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), [21,22]. Similarly to L+C, this system has been defined on 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 [2,23] 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 [5,24] 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 systems, like XFrog, or sketch-based systems 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. With this aim, our team developed the AMAPmod software [25] several years ago, 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 [26–29]. The PlantGL library was developed to support geometric processing of plant scenes in VPlants, for applications ranging from computer graphics [30,31] to different areas of biological modeling [32–34,15,35,36]. 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 enable users to concentrate on application issues rather than on technical details, and iv) the large set of available Python scientific packages can be freely and easily accessed by modelers in their applications. From a 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 the 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 design

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* [37].
- *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 available script languages, Python was found to present a unique compromise between the following features: it is (a) open-source; (b) available on the main operating systems; (c) object-oriented; (d) simple to use with syntax sufficiently intuitive for non-computer scientists (e.g. for biologists); (e) interactive: it allows direct testing of pieces of code without requiring a compilation process; and (f) has excellent support for integrating code written in compiled languages (e.g. C, C++, Fortran). Additionally, the Python community is large and very active and a large number of scientific libraries are available and can be imported into a program at any stage of model development.

2.2 Software architecture

The overall architecture layout of `PlantGL` is shown in Figure 1, including several layers of encapsulation and abstraction. The geometric, algorithmic and GUI libraries lie in the core of `PlantGL`. The geometric library encapsulates a *scene-graph* data structure and a taxonomy of *geometric objects*. The *algorithm* library contains tools to manipulate and display geometric structures (for instance `OpenGL` rendering). The GUI is developed as a set of `Qt` widgets

and can be combined with the previous components to provide visualization facilities. On top of this first layer, a set of wrappers create interfaces of the `C++` classes into the `Python` language using the `Boost.Python` library [38]. Because we design our `C++` libraries in an oriented-object manner, the `C++` class interfaces were totally compatible with the `Python` framework. All these interfaces are gathered into a `Python` module named `PlantGL`. Additionally, some automatic conversion tools with standard python structures were added into the wrappers in order to reinforce compatibility and integration. This module is thus integrated seamlessly with standard `Python` and enables further abstraction layers, written in `Python`. Extension of the library can be done using either `C++` or `Python`.

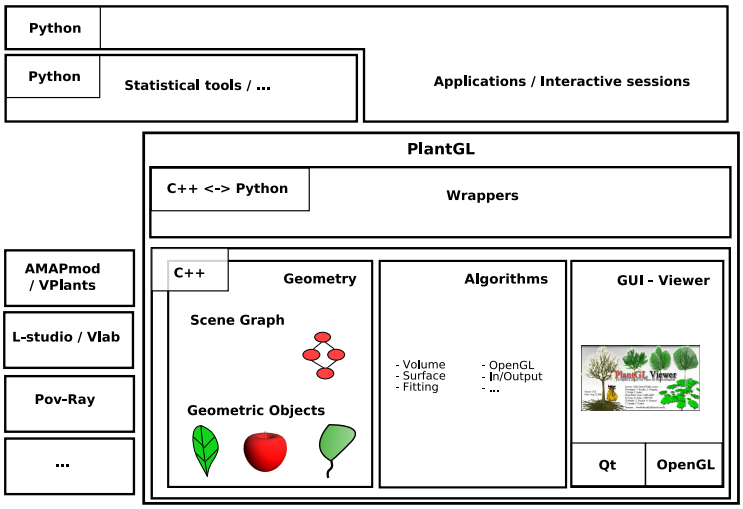


Fig. 1. Layout of the `PlantGL` architecture. It contains three C++ components: a geometric, an algorithmic and a GUI library. On top of this, wrappers implement interfaces with the `Python` language. `PlantGL` primitives can be used by modellers to develop scripts, procedures and applications in the embedding language `Python`. Data structures can be imported from and exported to other plant modelling software systems mentioned on the bottom left.

2.3 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*.

2.3.1 Scene-graphs

Scene-graphs are a common data structure used in computer graphics to model scenes [39]. 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 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) and an appearance node (e.g. which defines a color, a material or a texture). Non terminal nodes correspond to grouping or transformation nodes that allow the user to set the position, orientation and size of a geometric object. They are used to build complex objects from simple 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. 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. The discretization process is explicitly controlled by parameters of the models that indicate how many discrete primitives have to be created.

`PlantGL` contains a number of geometric models to represent points, curves, surfaces and volumes. They range from simple classical models such as *Cylinder*, *Sphere*, and *NURBS* to more specific ones, e.g. *Generalized Cylinder*, *Hull*, *etc.* Models dedicated to plant representation will be detailed in section 3.

2.3.2 Algorithms

In `PlantGL`, algorithms are separated from data structures for flexibility and reuse purposes. Given the heterogeneity of the nodes contained in the scene-graph, an algorithm applied to a scene needs to adapt its execution according to the type of node it is applied to. For this, we implemented the *visitor* design pattern [37] that makes it possible to keep algorithms outside node objects definition by delegating the mapping from node to algorithms to a separate visitor object called an *action*. It is thus possible to add new algorithms by implementing new actions without modifying the node classes or loosing performance (more details in appendix A).

In the actual implementation, `PlantGL` supports approximatively 40 *actions* that can be applied on a scene graph. The main ones can be classified into the following categories : *converters*, for instance from parametric to explicit representations; *renderers*; algorithms for the *characterization* of a scene using volume, surface or center of inertia; *fitting* algorithms to compute global rep-

resentations from a set of detailed shapes using for instance bounding volumes (sphere [40], box), convex hull [41]; *ray casting* using CPU or GPU; *space partitioning* e.g. in an Octree; *etc.* This last structure make it possible to determine quickly spatial neighbours to test for possible intersections between geometries, for instance during organ positionning in plant model generation [42]. A family of algorithms also makes it also possible to build and export a scene in different scene description formats: some classical ones, such as `PovRay`, `Vrml`, *etc.* or of some plant dedicated systems like `AMAPmod/VPlants`, `LStudio/VLab`, `AMAPsim` and `VegeStar`. This reinforces interoperability of PlantGL with other modeling tools.

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

2.3.3 Visualization tools

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

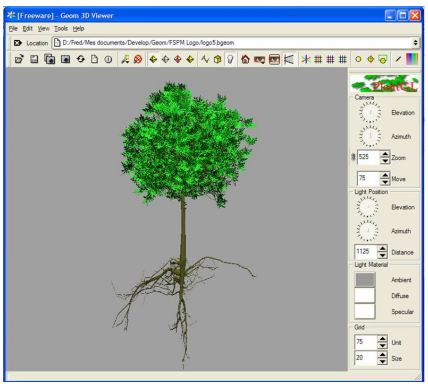


Fig. 2. Visualization of the detailed representation of a 15 year old walnut tree [11] with the PlantGL viewer.

The viewer and scenes are multi-threaded so that a user can manipulate a 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 are returned to the `Python` process using multi-threaded communications (see details in appendix A). This enables graphical control of a `Python` modelling process through the Viewer.

2.4 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. First, `PlantGL` provides a declarative language, `GML` [43], similar to `VRML` [44] with extensions for plant geometric objects. `GML` mainly adds persistence capabilities to `PlantGL` in both ascii and binary versions. For `C++` applications, it is possible to link directly with the library. In this case, `PlantGL` features can also be extended by adding new objects and new actions. Additionally, the full `PlantGL API` is accessible from the `Python` language. Combination of high level tools and languages such as `PlantGL` and `Python` allows rapid prototyping of new models and applications, as illustrated in section 4.

3 Construction of Plant Models

The design of `PlantGL` is focused on modelling and rendering of vegetative scenes. In this section, we present the specific geometric models and modelling tools of `PlantGL` useful for the representation of plant components at different scales. In an increasing computational complexity, we examine sequentially the representation of simple plant organs, tree crown, branching systems and cellular representation of tissues.

3.1 Plant organs

To represent simple organs, `PlantGL` contains a set of classical geometric models. For instance, some cylinders and frustums can be used to represent internodes, 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 user to express a full range of modelling techniques. For instance, the pine cone (Figure 3.a) is built by placing each scale using the golden angle [45]. The following code [2] sketches the placement of each scale.

[2] In all the code excerpts presented in this paper, standard python constructs are formated in bold to emphasize the structure of the code and the names of the

(a) (b) (c)

Fig. 3. (a) a pine cone, (b) a cactus and (c) a tulip. Models were created with simple `Python` procedures that create and position organs, like petals and leaves.

```
from plantgl import *
pine = Scene()
scale_smb = TriangleSet(...) # Geometric symbol of a pine scale
delta_angle = pi /(1+sqrt(5)) # golden angle between each scale
nb_scale, max_pine_radius, max_scale_size = 160, 50, 1.5
bottom_height, top_height = 10, 90 # global dimensions of the pine
def distToTrunk(u): # with u in [0,2], u < 1 for the bottom part
  if u < 1: return max_pine_radius*u
  else: return max_pine_radius*((2-u)**2)
def scaleSize(u):
  if u < 1: return max_scale_size*log(1+u,2)
  else: return max_scale_size*log(3-u,2)
def scaleHeight(u):
  if u < 1: return u * bottom_height
  else: return bottom_height + (u - 1) * top_height
for i in range(nb_scale):
  u = 2*i/nb_scale
  pine += AxisRotated((0,0,1),i*deltaAngle,
    Translated((distToTrunk(u),0,scaleHeight(u)),
      Scaled(scaleSize(u),scale_smb))
```

In this example, pine scales are identified by a normalized u position along the trunk with $u \in [0,1]$ for the bottom part and $u \in [1,2]$ for the top part. For a scale at position u, the functions `distToTrunk`, `scaleSize` and `scaleHeight` give the distance to the trunk, its size and its height respectively. Size of successive scales in the spiral follows a logarithmic law in this case. From this information and the golden angle, a geometric representation is built in the loop of the 5 last lines of code. Other algorithmic arrangements, such as the ones underlying Figure 3.b and c, can easily be made with python.

---

structures and functions provided by `PlantGL` are underlined.

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 [10] or competition for space [46]), 3D tree model reconstruction from photographs [7], and interactive rendering [47]. They are one original key application of `PlantGL`. In this section, we describe in detail three envelope models that were specifically designed to represent plant volumes: asymmetric, extruded and skinned hulls.

Asymmetric Hull - This envelope model, originally proposed by Horn [48] and Koop [49], then extended by Cescatti [10], makes it possible to easily define asymmetric crown shapes. The envelope is defined using six control points in six directions and two shape factors C_T and C_B that control its convexity (see Figure 4). With a few easily controllable parameters, a large variety of realistic crown shapes can be achieved.

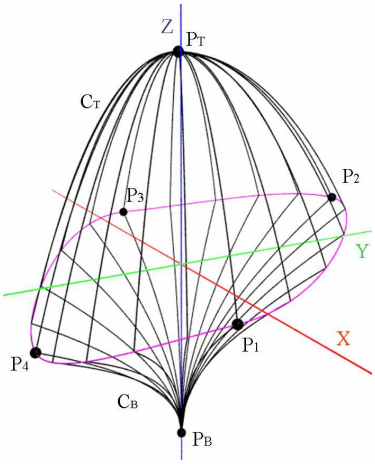


Fig. 4. Asymmetric hull parameters.

The first two control point, P_T and P_B, represent top and base points of the crown respectively. The four other points P_1 to P_4 represent the different radius of the crown in two orthogonal directions. P_1 and P_3 are constrained to lie in the xz plane and P_2 and P_4 in the yz plane. These points define a peripheral line L at the greatest width of the crown. For the x and y dimensions, L is composed of four elliptical quarters. The height of points of L is defined as an interpolation of the heights of the control points (see appendix B.1 for detailed equations).

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 of degrees C_T and C_B respectively. 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).

Fig. 5. Examples of asymmetric hulls showing plasticity of this model to represent tree crowns (inspired by Cescatti [10]).

Cescatti proposed 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 that makes it possible to build envelopes from photographs or drawings. For this, reference images can be displayed in the background of the different editor views. An illustration of this shape usage can be found for the interactive design of bonsai trees [30].

Extruded Hull - The use of extruded envelope for plant modelling was originally proposed by Birnbaum [50]. Such an envelope is defined with a horizontal and a vertical profile. This first profile can be acquired from the projection of a tree crown on the ground, given for instance by shadow on the field; and the second one from a lateral view of the tree.

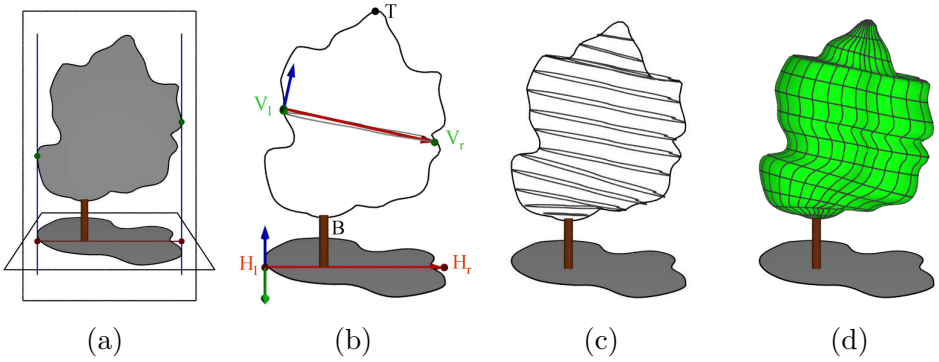
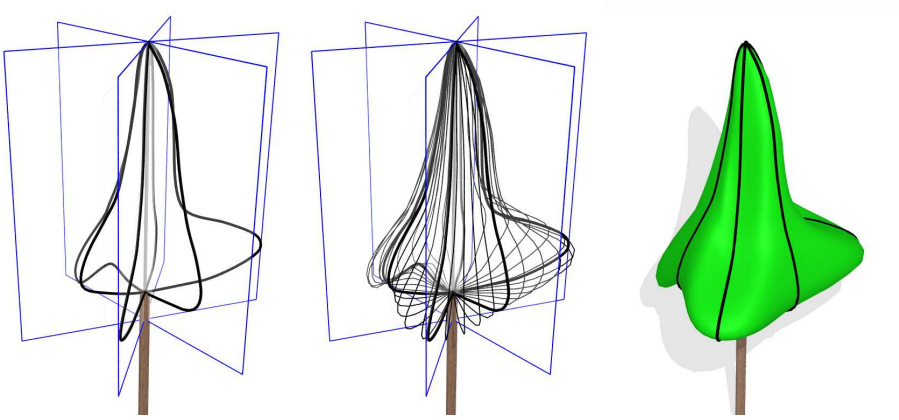


Fig. 6. Extruded Hull reconstruction. (a) Acquisition of a vertical and a horizontal profile. (b) Transformation of the horizontal profile into a section of the hull. (c) Computation of all sections (d) Mesh reconstruction.

The envelope is reconstructed by sweeping the horizontal profile inside the vertical profile (see Figure 6). 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 horizontal profiles inside V. Equations are given in appendix B.2.

An illustration of the reconstruction of such an envelope from photographs is given in section 4.1.

Skinned Hull - The previous envelope models account for variable radius of the crown in different directions but have limited input allowing variation of the profiles shapes to be captured. To alleviate this difficulty, we propose a new flexible profile based envelope model, namely *skinned hull*, which is defined as the interpolation of a set of vertical profiles given for any angle around the z-axis.



(a) (b) (c)

Fig. 7. Skinned Hull reconstruction. (a) The user defines 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.

The skinned hull is inspired from skin surfaces [51–53]. It generalizes the notion of surface of revolution with a variational section being an interpolation of the various profiles (see Figure 7 and appendix B.3 for equations). For the particular case of a single profile, the surface is a surface of revolution.

3.2.2 Foliage distributions

In particular studies such as light interception in eco-physiology or pest propagation in epidemiology, only the arrangements 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. In this example, leaves are generated above an horizontal rectangle delineated by `(xmin,xmax)` and `(ymin,ymax)`. Layers L_i are positioned between bottom and top heights (e.g. between `heights[`$i-1$`]` and `heights[`i`]`). To simulate different leaf distributions in the foliage, we asso-

ciate an increasing probability to each height for a leaf to be above this height, bottom height of first layer having $p_0 = 0$ and top height of last layer having $p_3 = 1$. For the three layers, a leaf will thus have a probability p_1 to be in the first layer, $p_2 - p_1$ in the second layer, and $1 - p_2$ in the third one.

```
from random import uniform
sc = Scene()
heights = [ 1, 2, 3, 4 ]          # Height of the layers limits
leaf_symbol = TriangleSet(...) # Leaf geometry to instantiate
for i in range(nbleaves):
 p = uniform(0,1)
 if p <= p1:
  height = uniform(heights[0],heights[1])
 elif p1 < p <= p2:
  height = uniform(heights[1],heights[2])
 else:
  height = uniform(heights[2],heights[3])
 # 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).

Fig. 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 (giving $p_1 = 0.1$ and $p_2 = 0.8$).

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 [15]. Such fractal spatial organizations can be captured by 3D *Iterated Function Systems* (IFS) [54].

An IFS is a set of contracting affine transformations $\{T_i\}_{i\in[1,M]}$. This family of contractions defines a contracting operator F for all bounded set of points

X in $\mathbb{R}^3$ such that:

$$F(X) = T_1(X) \cup T_2(X) \cup \ldots \cup T_M(X). \quad (1)$$

The F operator may be applied iteratively to an initial arbitrary 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).

$$L_n = F^n(I) \quad (2)$$

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. [55].

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, e.g. [15,34]. The following code shows the construction of the IFS of Figure 9. The affine transformations are defined as 4x4 matrices that are build here from a translation (`t`), a scaling factor (`s`) and a rotation defined from some euler angles (`a`). The initiator is a disk that represents a simple leaf shape.

```
transformations = [Matrix4.fromTransform(t,s,a) for t,s,a in
 [((0,0,2),1/3,(0,0,0)),          ((0,0.5,1.5),1/3,(45,90,0)),
 ((0,-0.5,1.5),1/3,(45,270,0)),((0.5,0,1),1/3,(45,0,0)),
 ((-0.5,0,1),1/3,(45,180,0)),    ((0,1,.5),1/3,(45,90,0)),
 ((0,-1,.5),1/3,(45,270,0)),     ((1,0,0),1/3,(45,0,0)),
 ((-1,0,0),1/3,(45,180,0))]]
i = IFS(4,transformations,Disk(1))
```

3.3 Branching system modelling

3.3.1 Scene-graphs for plants

In `PlantGL`, the construction of a branching system comes down to instantiating 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

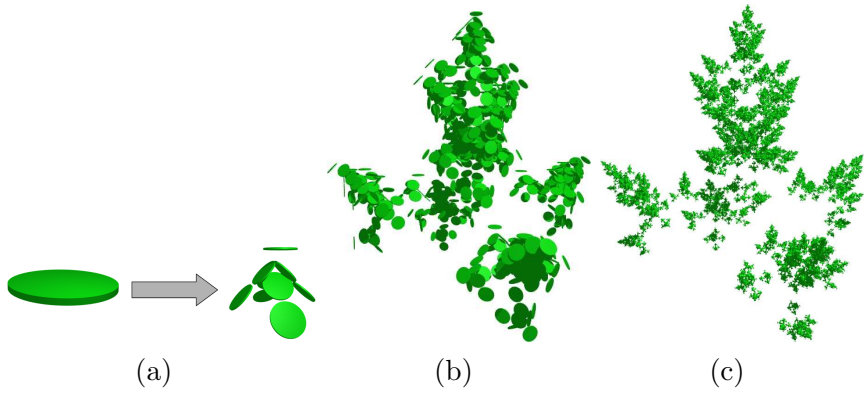


Fig. 9. Construction of a fractal foliage using an IFS. (a) The initiator is a disc (representing a leaf shape). In the first iteration, the initiator is duplicated, translated, rotated and resized according to the affine transformations that compose F, leading to L_1. (b-c) IFS foliages L_3 and L_5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

nodes are organized as a tree graph (as described in [56]) in which two types of edges can be specified to distinguish branching (+) and succession (<) relationships between parent and child nodes (see 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.

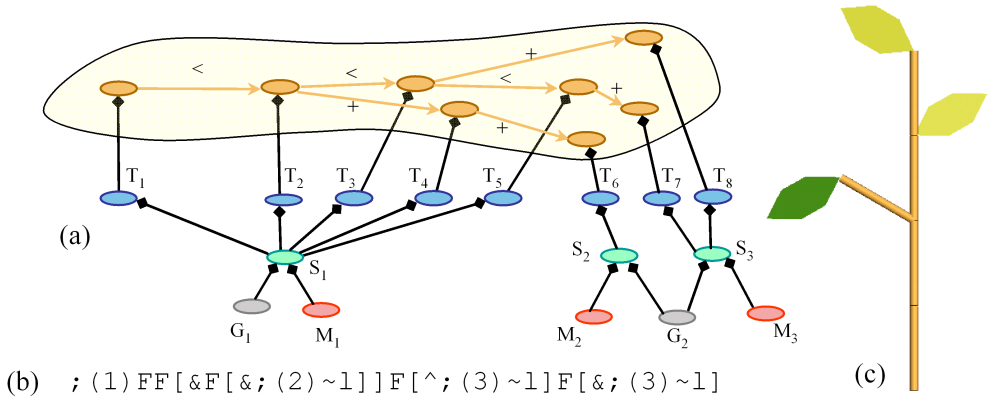


Fig. 10. A p-scene-graph. (a) The scene-graph with structural nodes in orange, transformation in blue, shape in green, appearance in red and geometry in grey. (b) The corresponding L-systems bracketed string from which it has been constructed. The 'F' symbols are geometrically interpreted as cylinders, '~l' as leaf symbols, '&' and '^' as orientation operations and ';' as color specifications, [57]. (c) The corresponding geometric model.

The topological relationships specified in the p-scene-graph express physical connections between plant components. For this, a set of constraints, namely

within-scale constraints, can formalize these connections in term of geometry [26]. 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 [56] with graphical informations [58]. Such multiscale graphs correspond to recursively quotiented tree graphs [56]. It is possible to define multiscale organization from a simple detailed tree graph (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 representation may be used to bound the development of a plant at a more microscopic level. Such a strategy was used for instance in [30] 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 [41] or minimal bounding sphere [40], *etc.* These envelope representations can thus be used to characterize globally the geometry of a branching system at different scales, for instance for computing the fractal dimension of a plant [15].

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 software.

From a generative perspective, we particularly emphasized in the current version of `PlantGL` the connection with `L-studio/VLab` [6], a widely used L-system based modelling framework for plant growth modelling. An L-system [3] is a particular type of rewriting system that may be used to simulate the

development 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 module of the string. Therefore, the development of a tree structure through 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, [59]. 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 [3] (see 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` [60]. 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.

```
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` [26], and `VegeSTAR`

(a) (b) (c)

Fig. 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 [4] and (c) the root system of an oak tree [13].

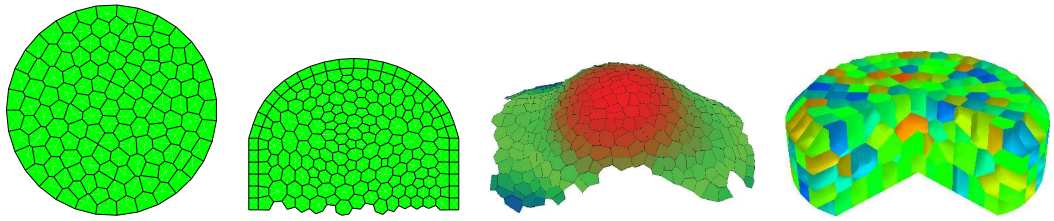
[61] are also available and make it possible to create 3D plant models from measured data (see Figure 11).

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 represented by tree graphs since the connection networks between regions usually contain cycles. To model the neighborhood relationship between regions correctly, 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* [62]. 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 set of geometric algorithms has been designed to simplify the manipulation of the *t-scene-graphs* during simulations.

- The first algorithm makes it possible to define the geometry of an element of dimension $n+1$ from the geometric information of its components of



a b c d

Fig. 12. Tissue models. (a) 2D tissue, (b) 2D transversal cut, (c) 3D surface tissue from [33], (d) 3D tissue.

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 smallest elements.

- The 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, ...).
- The third algorithm has been designed to refine *t-scene-graphs* into small triangular elements. Resulting meshes can be used either to visually display the tissue or in conjunction with finite element 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 [33], 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` [63], 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. [33,34,15,31,36]) and other plant research groups (e.g. [32,35]). The library allows modellers to address graphic and geometric issues in the different phases of a modeling process, i.e. observation, analysis, simulation and model evaluation. In this section, we aim to illustrate how PlantGL provides a set of useful efficient 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

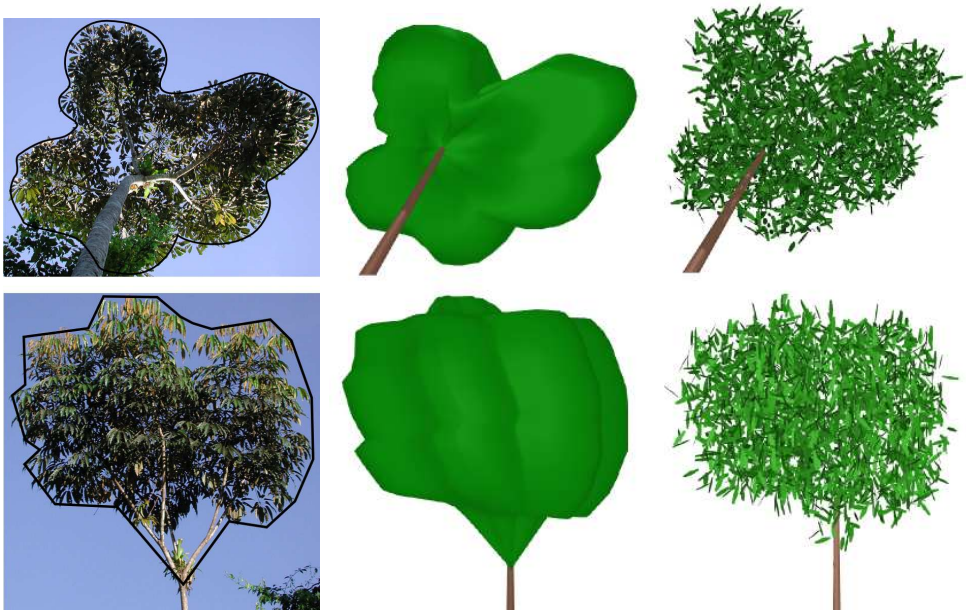
In plant modeling, 3D digitizing of plant structure has become a topic of increasing importance in the last decade. Various methodologies have been used to digitize plants at different levels of detail for leaves, for branching systems, and also for tree crowns [64,65,26,50,7,46]. Among these approaches, the reconstruction of 3D models of large/tall trees (like trees of a tropical forest for example) remains a challenging problem. This is mainly due to the difficulty of acquiring information in the field, and in capturing the intricate structure of such plants. In this section, we show how the new envelope-based tools provided in PlantGL can be used for 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 crowns in the context of light modeling in plant stands [10]. More recently, non-parametric visual hulls have been used in different application contexts to characterize the plant volume based on photographs [7,47,46].

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

Using a set of images of a tree (being either photographs or botanical drawings) with known camera positions and orientations, the modeler has to define a number of crown profiles according to the selected envelope model. 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



(a) (b) (c)

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

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 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.

```
hull # reconstructed hull
bbx = BoundingBox(hull)
foliage = Scene()
def random_position():
  return (uniform(bbx.xrange()),uniform(bbx.yrange()),
          uniform(bbx.zrange()))
for i in range(nbleaves):
  pos = random_position()
  while not inside(hull,pos) :
    pos = random_position()
  foliage += Translated(pos,leaf_symbol))
```

Using this code, the generation of the foliage of Figure 13.c composed of 2000

leaves is made in 1 sec. on a Pentium IV 2.0 GHz[3]. Using `Python`, it is possible to define foliage distribution using more complex algorithms, such as those described in [6,7,30].

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 [66]. A 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 [67]. The amount of intercepted light is summarized with the *canopy openness index* defined as the ratio between white pixels and total number of pixels on the picture.

Fig. 14. Assessment of plant model reconstruction using hemispheric view [66]. 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 to `Pov-Ray` [68] to compute a hemispheric picture at the same position. Here, the evaluated canopy openness is 40% for the real photograph and 49 % for the virtual tree.

[3] The computation times indicated in the remainder of this paper are for the same hardware configuration.

4.2 Analysis of Plant Geometry

Plant geometry is a parameter of paramount importance in the modeling of plant-environment interactions. 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 [69]. Several parameters have been introduced for this purpose, such as fractal dimension and lacunarity. These parameters are intended to capture the essence 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 studying the variation of the number of tiles with tile size.

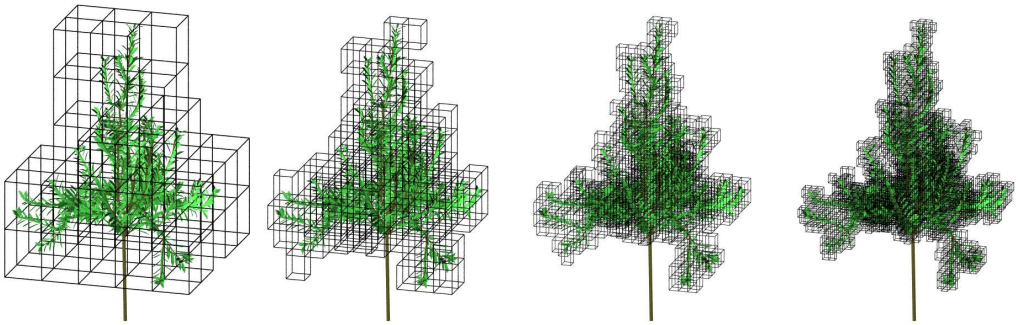


Fig. 15. The box counting method applied to the foliage of a digitized apple tree [70]. The tree is 2m height with around 2500 leaves. Global bounding box has a volume of 10 m^3 and serves as the initial voxel. A grid sequence is then created by subdividing uniformly this bounding box into sub-voxels. 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. [71]. 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 [72,15,34].

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 [69], 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:

$$D_\delta = \lim_{\delta \to 0} \frac{\ln N(\delta)}{\ln \frac{1}{\delta}}. \quad (3)$$

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):
  nbvoxels = [log(Grid(scene,div).nb_intercepted_voxels())
              for div in range(maxdivision)]
  delta=[log(1./i) for i in range(maxdivision)]
  slope,itcept,r,ttp,stderr=stats.linregress(nbvoxels,delta)
  return slope # slope of the regression
```

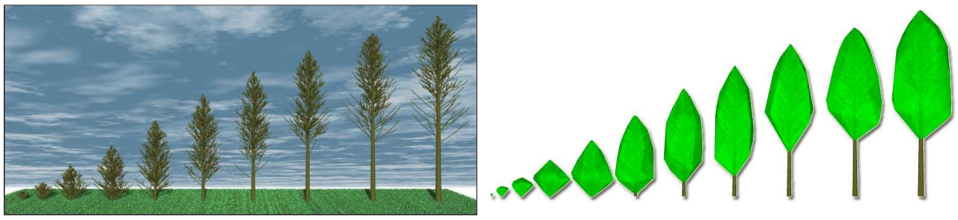
Figure 15 illustrates the application of the box counting method on the foliage of a 3D-digitized apple tree [70]. For this example, computation of the 30 grids of decreasing sizes took 0.7 sec. PlantGL can be used in a similar way to compute various fractal properties of plants [15,34]

4.2.2 Envelope-Based Analysis

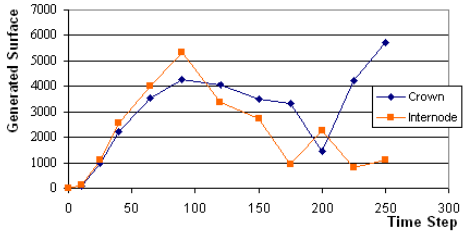
Parametric envelopes provided in PlantGL can also be used to analyse volumetric properties of plant crowns. For example, in order to quantify the development of a plant crown over time, envelopes can be adjusted to the crown of the developing tree at different ages and their surface or volume can then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

Figure 16 illustrates this approach together with the possibility to import plant data from other software. The growth of a eucalyptus was simulated at various ages using the AMAPsim software [23], 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. 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]
```



(a) (b)



(c)

Fig. 16. (a) A eucalyptus tree simulated at various age (from 1 to 8 months on a scale from 10 to 250) with the `AMAPsim` software, exported to PlantGL and rendered with `Pov-Ray` software [68] (b) Global representations of eucalyptus crown at the various ages (c) Increments of wood and hull surfaces over time. The different degrees of correlation and their associated time period enable us to identify the various phases of the crown development.

```
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 through time can be analysed as shown in Figure 16.c. Comparison at a more microscopic scale with the leaf area variation can thus be made.

4.3 Simulations based on plant geometric models

The use of flexible geometric models of plants is not restricted to the analysis of plant structure. They can be used as well for the simulation of various physical or 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 [73–75], shoot apical meristem [76–80], and flower [81]. 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 model the development of a bump formation in a tissue for instance to simulate development of a primordium at a shoot apex. We assume that the tissue is composed of polygonal cells in 2D (respectively polyhedral cell in 3D) delimited by 1D walls in 2D (respectively by polygonal walls in 3D).

The bump formation of a primordium at the surface of a meristem can be approximated (in cylindrical coordinates) by :

$$z(r,t) = \frac{h(t)}{1 + e^{k(r - r_m(t))}} \tag{4}$$

where h is the height of the bump, r_m is the radius of the bump at half its height and k is a shape factor that defines the slope of the bump. h and r_m vary throughout time at a rate ρ_h and ρ_r respectively. From this equation we can derive a time dependent velocity field for surface vertices. The velocity of the internal vertices is an interpolation between the surface velocity and the velocity of the most inner vertices. To simplify, we assume that the most inner vertices (at altitude z_{min}) are fixed and their speed is then equal to zero. At each time t of the simulation, each vertex is moved according to its velocity. This process progressively modifies the cell size, and consequently the overall tissue shape. During the growth, if a cell has a volume (or surface in 2D) that reaches a predefined threshold, it divides into two children cells. Different algorithms implementing cell division are available on a tissue object, *e.g.* [82]. Here follows the code of such a tissue growth in `PlantGL` for a particular choice of the cell division algorithm :

```
tissue = createGridTissue( (20,5) )
def surface_altitude (r, t) :
    return h(t)/(1+exp(k*(r-rm(t))))

def velocity_field (pos, t) :
    r=norm(pos.x,pos.y)
    tmp=exp(k*(r-rm(t))
    surface_speed=(rho_h(t)+k*h(t)*rho_r(t)*tmp/(1+tmp))/(1+tmp)
    zspeed=surface_speed*(pos.z-zmin)/(surface_altitude(r,t)-zmin)
    return Vector3(0,0,zspeed)

for t in range(time_begin,time_end,delta_time):
```

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

Results of the simulation are presented on figure 17. This growth simulation of the tissue, composed of 100 cells, using 100 time iterations took 4.5 sec. Note that interpolation has been slightly modified to account for a constant size of the two external layers of cells.

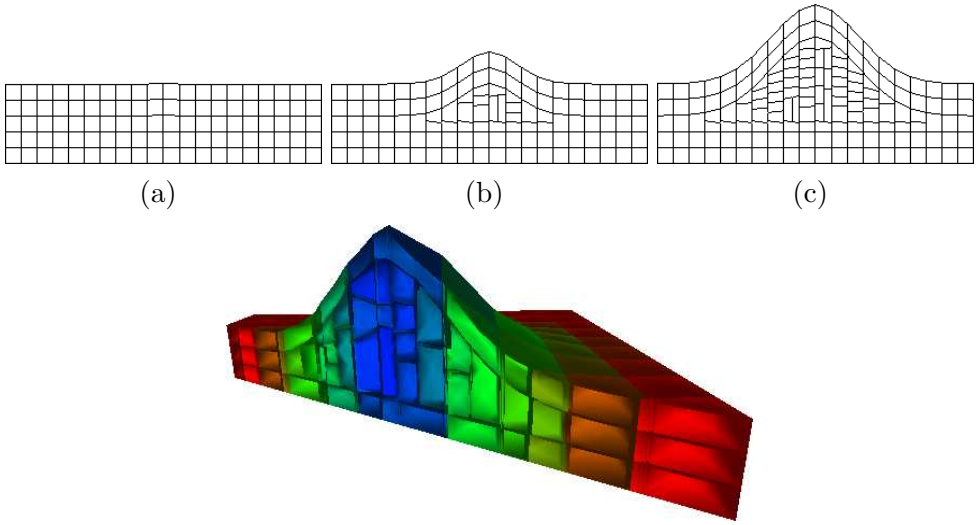


Fig. 17. (a,b,c) 2D geometrical representation of a growing tissue at three different times. (d) 3D simulation of bump formation on a tissue

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. While the canopy openness index presented in section 4.1.2 gives an estimation of the ratio of light intercepted by the plant from one view point, this example addresses the problem of integrating such a measure on the whole plant structure. Additionally, it illustrates the use of `PlantGL` in the context of model assessment and shows how high-level geometric operations used in light interception models can be simply performed with `PlantGL`.

Light intercepted by a plant can be characterized by STAR values, namely

Surface to Total Area Ratio [83]. This eco-physiological parameter is a directional quantity defined by taking the ratio of the area of the projection of a tree foliage S_Ω in a particular direction Ω to its total leaf area S. The STAR is thus the mean irradiance of a leaf area unit.

$$STAR_\Omega = \frac{S_\Omega}{S} \tag{5}$$

The directional STAR index can be integrated over all the sky vault to characterize the overall light interception of a tree. For this, a set of directions given by the turtle sky discretization [84] can be used and associated STAR values are averaged after weighting by the standard overcast sky radiance distribution [85].

Since the total leaf area of a real plant is often expensive to measure, approximate values of the STAR are often used 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 [86]. In this case, the plant is supposed to be a 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 :

$$p_0(b) = exp\left(-G_{\Omega_b}.LAD.l_b\right) \tag{6}$$

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 [87]:

$$\widehat{STAR_\Omega} = \sum_{b=1}^{B} S_b(1 - p_0(b))/S \tag{7}$$

where S_b is the cross section area of a beam. This model-based definition of the STAR can be compared to STAR from Equation 5 to evaluate the quality of light model assumptions. The resulting difference characterizes the error due to the model's underlying hypotheses (homogeneity/randomness of the foliage distribution, negligibility of leaf size, ...) with respect to the actual canopies [86].

In `PlantGL`, both STAR quantities, *i.e.* the projection-based and turbid-medium-

based STARs, can be computed from a plant mockup using the high-level library functions. The projection based 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 [86] 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)
```

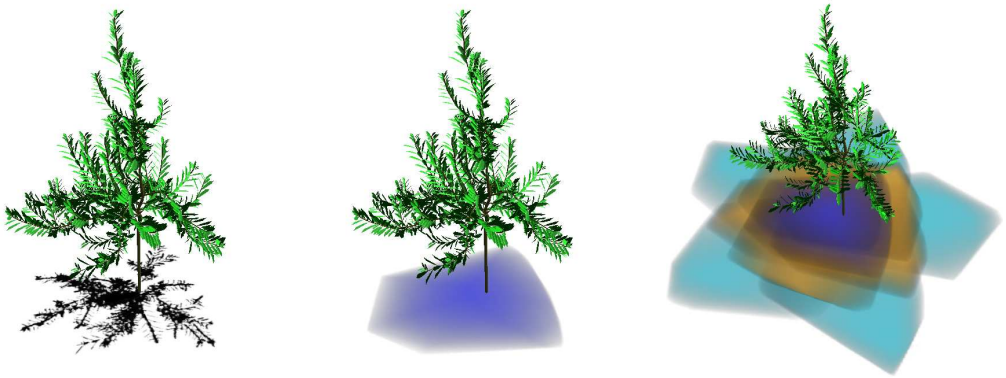
For the turbid medium based STAR, the envelope of the tree crown must first be computed. Then, a set of beams of direction Ω are cast and their interceptions and resulting length in the crown volume are computed. A sketch of such a code would be as follows:

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

with `intersection` containing the in and out intersection points of the ray and a hull and `intersections` being a list of such structure. The STAR with ray casting took 0.3 sec for each direction to be computed with `PlantGL` and the approximated one took 10 sec using 50000 rays.

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 ways to deal with abstract representation of



(a) (b) (c)

Fig. 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 area of the shadow and the plant leaf area. (a) Light intercepted from top direction using ray casting. STAR in this direction is equal to 0.23 (b) Light intercepted from same direction using Beer-Lambert hypothesis. STAR in this case is equal to 0.34. (c) Light interception sampled from different directions. The different colors are used to mark the difference between various elevations of ray direction. Sky integrated STAR is equal in the case of ray casting to 0.44 and in the case of Beer-Lambert to 0.58. This confirms that the turbid medium hypothesis over-estimates the STAR measure [86].

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 [88].

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 [31] which is an extension of [89,90] 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 [91] 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 [92,93] 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 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 [94].

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.

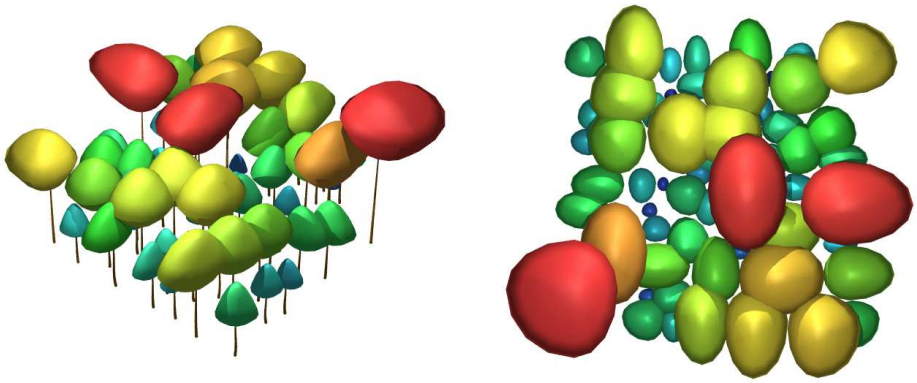


Fig. 19. A front and top view of a generated stand at the crown scale. Different colors are used to differentiate various layers of vegetation.

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 with the L-systems models using the generative procedure described in [30]. 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 [65] 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` [68]. Each plant geometric model was converted and assembled in this format. Figure 20 illustrates the resulting scene. The computation of the distribution of the 56 trees on the terrain using the depletion-replacement algorithm with 4000 iterations is made using pure `Python` and took 5 min. Generation of individual tree structure using L-systems took 20 sec per tree. The final `Povray` rendering took 5 min.

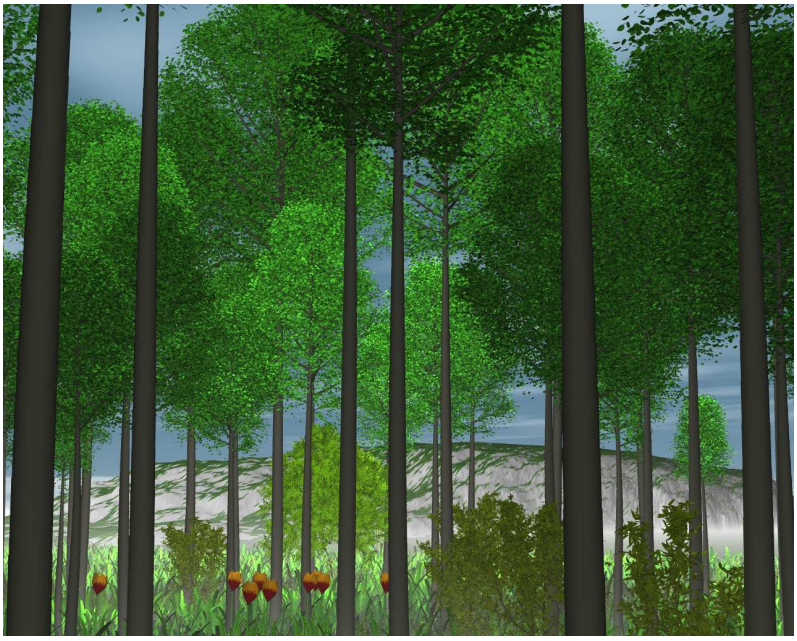


Fig. 20. A community of plants generated from a Gibbs process [31]. The scene is made of plants from different sources: beech trees of different sizes and ages where generated using the ecosystem model presented in section 4.3.3. A walnut tree corresponding to a 3D digitizing of a real plant built using AMAPmod/VPlants, virtual bushes flowers and grass created procedurally in `Python` with `PlantGL`

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 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 kindly making the `LStudio/VLab` software available to them, D. Da Silva for STAR computation on the apple tree and the shadow images, 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, J. Hanan, Development models of herbaceous plants for computer imagery purposes, in: SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques, ACM, New York, NY, USA, 1988, pp. 141–150.

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

[3] P. Prusinkiewicz, A. Lindenmayer, The Algorithmic Beauty of Plants, Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[4] 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.

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

[6] 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.

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

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

[9] 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).

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

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

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

[13] 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.

[14] 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.

[15] 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.

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

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

[18] R. Karwowski, P. Prusinkiewicz, Design and implementation of the L+C modeling language, Electronic Notes in Theoretical Computer Science 86.

[19] 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.

[20] W. Kurth, Growth Grammar Interpreter GROGRA 2.4: A software tool 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).

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

[22] 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.

[23] J.-F. Barczi, P. de Reffye, Y. Caraglio, Essai sur l'identifcation et la mise en oeuvre des paramétres nécessaires àla 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.

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

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

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

[27] 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.

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

[29] Y. Guédon, Y. Caraglio, P. Heuret, E. Lebarbier, C. Meredieu, Analyzing growth components in trees, Journal of Theoretical Biology 248 (2007) 418–447.

[30] 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.

[31] 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) (2007) 9–19.

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

[33] 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.

[34] 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) 4291 (2006) 751–750.

[35] G. Louarn, Y. Gudon, 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.

[36] 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.

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

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

[39] 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.

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

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

[42] N. Greene, Voxel space automata: modeling with stochastic growth processing in voxel space, Computer Graphics 23 (3) (1989) 175–184.

[43] F. Boudon, C. Nouguier, C. Godin, GEOM module manual. I. User's guide, Document de travail du Programme Modlisation des plantes 3-2001, CIRAD (2001).

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

[45] H. Vogel, A better way to construct the sunflower head, Mathematical Biosciences 44 (1979) 179–189.

[46] J. Phattaralerphong, H. Sinoquet, A method for 3d reconstruction of tree crown volume from photographs: assessment with 3d-digitized plants, Tree Physiology 25 (2005) 122942.

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

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

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

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

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

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

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

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

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

[56] C. Godin, Y. Caraglio, A multiscale model of plant topological structures, Journal of Theoretical Biology 191 (1998) 1–46.

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

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

[59] 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.

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

[61] 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.

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

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

[64] 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.

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

[66] 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).

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

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

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

[70] 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.

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

[72] 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.

[73] 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.

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

[75] 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.

[76] 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.

[77] 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.

[78] 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.

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

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

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

[82] 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.

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

[84] J. A. Den Dulk, The interpretation of remote sensing, a feasibility study, Ph.D. thesis, Wageningen university (1989).

[85] P. Moon, D. Spencer, Illumination from a non-uniform sky, Transactions of the Illumination Engineering Society 37 (1942) 707712.

[86] 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.

[87] 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, INRA Editions, Paris, 1993, pp. 131–158.

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

[89] 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.

[90] 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.

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

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

[93] 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).

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

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

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

[97] M. Raynal, D. Beeson, Algorithms for Mutual Exclusion, MIT Press, Cambridge, MA, USA, 1986.

A Implementation issues

Different issues have to be addressed in order to implement an efficient scene-graph that preserves performance and flexibility. Literature on scene-graphs [95] offers various interesting solutions that were formalized as design patterns and which inspired our implementation.

Memory management - Scene-graphs have to deal with a 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 that the same object is referenced several times. Allocation and deallocation of this object in memory can thus be problematic. To address this issue, we use *Reference counting pointers* [96] which manages pointers to dynamically allocated objects. They are responsible for automatic deletion of the objects when no longer needed.

Visitor Actions - As stated in section 2.3.2, algorithm polymorphism is implemented using the visitor design pattern to avoid modification of object interfaces. 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. For this we need to determine at run time the actual type of a node before choosing the algorithm. To do that, each class of the hierarchy implement an *apply* method that takes an action object as an argument. The method makes a call to the action, passing the node as an argument, and the action executes the appropriate algorithm depending on the actual node type.

Wrappers - The seamless transition between `C++` and `Python` is ensured by the `Boost.Python` library [38]. 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 geometrical descriptions generally rely on large number of triangles. In order to minimize time cost for sending all the triangles to the GPU, the triangle points are packed into arrays that can be sent in one command to the card. Moreover, `OpenGL` commands 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 separate threads. For this, the `Qt` threads implementation is used. Inter-thread communication is made using `Qt` event dispatch mechanism which is extended with synchronized dispatch using mutual exclusion lock (also called mutex) [97].

B Mathematical details of envelope models

B.1 Asymmetric Hull

This envelope model is defined by six control points and two shape factors C_T and C_B. The four points P_1 to P_4 define the peripheral line L. For x and y, points of L form four elliptical quarters centered on the origin. For z, their height 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 $\sin^2\theta$ in the interpolation. 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

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

Points of L are connected to the top and base points with quarters of super-ellipses of degrees C_T and C_B respectively. Letting $P_l \in L$ and P_T be the top point of the envelope, the super-ellipse quarter connecting P_l and P_T is defined as

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

An equivalent equation is obtained for super-ellipse quarters using P_B and C_B instead of P_T and C_T.

B.2 Extruded Hull

The extruded envelope is defined from a horizontal profile H and a vertical profile V. First, two fixed points, B and T, at the top and bottom of the vertical profile respectively, are defined (see Figure 6.b). V is split into two open profile curves V_l and V_r, the left and the right part of V respectively, with their first and last points equal to B and T. A slice $S(u)$ is thus defined using two mapping points $V_l(u)$ and $V_r(u)$. Its span vector $\overrightarrow{S(u)}$ is set to $\overrightarrow{V_l(u)V_r(u)}$

On the horizontal profile H, two anchor points, H_l and H_r, are defined. A horizontal section of the extruded hull is computed at $S(u)$ so that the resulting curve fits inside V (see Figure 6.b and c). Finally, to map $\overrightarrow{H_lH_r}$ to $\overrightarrow{S(u)}$, the transformation is a composition of a *translation* 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 $\overrightarrow{S(u)}$, and a *scaling* by the factor $\left\|\overrightarrow{S(u)}\right\| / \left\|\overrightarrow{H_l H_r}\right\|$.

The equation of the Extruded Hull surface is thus:

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

B.3 Skinned Hull

The envelope of a skinned hull is a closed skinned surface which interpolates a set of profiles $\{P_k(u), k = 0, \ldots, K\}$ positioned at angle $\{\alpha_k, k = 0, \ldots, K\}$ around the z axis. Similarly to an extruded hull, all vertical profiles are split into two open profile curves homogeneously parameterized. We assume thus that all these profiles $P_k(u)$ are non-rational B-spline curves with common degree p and number n of control points $P_{i,k}$ (see [52] for homogenization details). From these profiles, a variational profile Q can be defined that gives a section for each angle α around the rotation axis (see Figure 7.b). It is defined as

$$Q(u,\alpha) = \sum_{i=0}^{n} N_{i,p}(u) Q_i(\alpha) \tag{B.4}$$

where the $N_{i,p}(u)$ are the pth-degree B-spline basis functions and the $Q_i(\alpha)$ are a variational form of control points. Q interpolates all the profiles P_k. Therefore, $Q_i(\alpha)$ are computed using a global interpolation method [52] on the control points $P_{i,k}$ at α_k with $k \in [0, K]$ For this, let q be the chosen degree of the interpolation such as $q < K$. $Q_i(\alpha)$ are defined as

$$Q_i(\alpha) = \sum_{j=0}^{K} N_{j,q}(\alpha) R_{i,j} \tag{B.5}$$

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

$$\forall i \in [0,n], \forall k \in [0,K], P_{i,k} = Q_i(\alpha_k) = \sum_{j=0}^{K} N_{j,q}(\alpha_k) R_{i,j} \tag{B.6}$$

Geometrically, the surface of the skinned hull is obtained by rotating $Q(u,\alpha)$

about the z axis, α being the rotation angle. It is thus defined as

$$S(u, \alpha) = (\cos \alpha Q_x(u, \alpha), \sin \alpha Q_x(u, \alpha), Q_y(u, \alpha)). \tag{B.7}$$