Plausible reconstruction and rendering of semi-procedural landscapes

Author:
Óscar Argudo Medrano

Director:
Carlos Andújar Gran (CS)

Codirector:
Antonio Chica Calaf (CS)

Defense date: September 10, 2014

Master in Innovation and Research in Informatics
Computer Graphics and Virtual Reality

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

Contents

List of Figures

List of Tables

Abstract

In recent years, there have been numerous advances in modeling the world and visualizing it using virtual globe applications. While the resolution of these 3D scenes has improved significantly – specially in urban areas – the amount of detail is still insufficient for its inspection from points of view close to the ground level such as the height of an average person. Increasing this detail usually involves scanning again the area of interest with more accurate equipment or taking more samples per area. Therefore, the acquisition process is expensive because it involves a substantial amount of time, human resources and expensive equipment.

This project presents a new approach to increase the visual richness of rural and forest areas while avoiding the abovementioned costs. We build on the premise that in dense vegetation areas we do not need to reconstruct exactly the actual trees but a plausible model that resembles the kind of vegetation found there. We call our approach a semi-procedural reconstruction, since the trees are procedurally built using real information as guidance.

Our method requires just an input photograph of the kind of tree or shrub that we want to introduce to the terrain scene and it produces a tree model in a representation that allows for efficient level of detail, thus able to render forest scenes with several thousands of exemplars in real time.

The reconstruction process uses the silhouette of the tree crown and inflates a mesh which has this silhouette and minimizes the thin-plate bending energy by solving a linear system of equations. Then, using an example-based texture synthesis algorithm, we generate texture data in all directions around this mesh. This texture is used both to color the mesh and to perturb its surface relief, adding more detail to the smooth surface produced by the minimization system. This perturbation is proportional to the luminance of the texture at different resolution levels.

Once we have the final perturbed mesh, we store it using a radial representation in a cubemap we call the radius cubemap. This cubemap will be the basis of our level of detail rendering algorithms. Farthest trees will be rendered by instancing a proxy geometry and raycasting this cubemap to compute the intersection of the viewing ray and the crown geometry. Nearest trees will be rendered from instancing a variable number of billboards depending on their projected size on the screen. To ensure shape matching between both representations, these billboards are cropped to the silhouette of the tree crown by checking the radius cubemap.

The tests we performed show that our method offers a plausible reconstruction of tree models from photographs, and our representation lets us render tens of thousands of tree crowns in real time. The visual richness of the terrain model is largely augmented with the inclusion of our models, and can be used for realistically rendering points of view close to the ground and navigating a forest scene that resembles the reality.

Chapter 1

Introduction and motivation

During the last years, there have been significant advances in the areas of modeling, reconstruction and visualization of urban scenes. The use of photogrammetry[1] based on aerial or satellite photographs, as well as the use of laser techniques such as LIDAR[2], has allowed the creation of Digital Terrain Models (DTMs) that can be sent remotely and inspected in real time through virtual globe applications like Google Earth [17] or NASA World Wind [31].

Even though the resolution of these 3D scenes has improved noticeably in the last years, reaching in some urban areas resolutions up to 1m for the DTMs and 25cm/pixel for the corresponding aerial photographs (according to publicly available data from the Institut Cartogràfic i Geològic de Catalunya [21]), this level of detail is only sufficient for aerial points of view, but it does not have enough information for a visual simulation from points of view close to the surface - for example, from the eyes of a human observer moving through the scene.

Some technologies like Google Street View [18] solve this problem and allow detailed inspection by using a fixed set of 360º panoramic photographs. This constraints the navigation to jumping between fixed positions and do not provide a smooth interpolation between the viewpoints. Moreover, each panorama is an independent image that has to be sent to the user, so the amount of redundant data to be sent can be huge.

Another important aspect of the current datasets is their cost of acquisition. Some of the technologies rely on expensive equipment, like LIDAR. Acquiring the data usually involves a displacement to the site, mounting the sensors on some vehicle (a car or a plane for aerial views) and moving through the scene to be captured. In order to improve the level of detail, a new acquisition with more samples per surface area has to be carried out. This implies either using a better equipment and/or performing a more in-depth scan of the area, which takes more time and increases the cost.

This is why we want to propose new methods and tools to help the reconstruction of terrains and to later visualize them. In particular, for this project we focused on vegetation for rural scenarios with large areas of forest. The applications mentioned above do not represent the trees or use a very simple and unnatural representation for them. We want to build a tool that would allow a user to take pictures from the trees present in the terrain area being modeled, such that the vegetation layer of this model is procedurally

[1] Making measurements from photographs, for example the height of a terrain.
[2] A technology that measures distances using the time of flight of a laser beam.

built taking this information into account. This way, the trees shown during the navigation will look more familiar, increasing its realism and the presence felt by the user.

Trees have a set of characteristics that make them specially hard to reconstruct and to efficiently visualize. The next chapter will discuss these and the relevant solutions proposed so far. Chapter 3 provides an overview of our proposed reconstruction method. We will then explain in detail how the algorithm produces a crown model and how we use the results to render it (chapters 4 and 5). Finally, we will see how we use these results in a real application, a terrain viewer (chapter 6). Finally, the report ends with a gallery of visual results and some performance tests, the conclusions and ideas for future work.

Chapter 2

Previous work

This chapter will analyze and discuss different existing methods used to generate tree models as well as how to represent them efficiently. First, we will analyze the synthetic or procedural methods of generating trees, since they were the first methods to appear. Afterwards, we will see the opposite approach: digitizing real trees. Finally, we will focus on how to render trees, specially in large scenes.

2.1 Synthetic generation of trees

For a comprehensive and detailed survey on procedural and rule-based methods of tree generation, the book “Digital Design of Nature: Computer Generated Plants and Organics” by Oliver Deussen and Bernd Lintermann [14] is a very useful reference. In this section we briefly summarize some of the methods described in chapters 4, 5 and 6 of this book. Therefore, the works cited are classified in the same way: procedural and rule-based methods.

2.1.1 Procedural modeling

Procedural methods are algorithms that try to reproduce usually a certain type of plant or species using a set of parameters and rules.

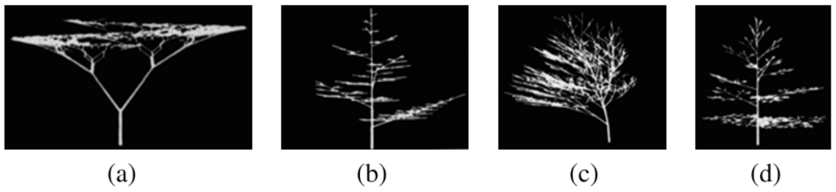


Figure 2.1: Branching structures by Aono and Kunii: (a) binary branching with angles 35°and -35°, (b) single branching with angle -70°, (c) use of an inhibitor to simulate wind, (d) branching angle depending on age. Image from [14].

The first branching patterns were defined using cellular automata on regular grids by Stanislaw Ulam in 1966. One year later, Dan Cohen proposed the first continuous branching procedural model using a set of rules [9]. The first branching structures in 3D were introduced by Honda in 1971 [20], using also a very simple recursive algorithm characterized by a small set of parameters: branching angles and length ratios of consecutive branch segments. Aono and Kunii [2] extended Honda's work to support the generation of a variety of branching patterns, statistical variations of angles, attraction and inhibition.

Reeves and Blau [46] focused on the realistic appearance of the tree model instead of the botanical correctness of the method. Starting with the main trunk, the branches are generated recursively. The parameters for this process depend on a distribution for each type of tree, and include characteristics like tree width and height, first branch height, branching angle and mean branch length. Since the generated structure looks very regular, a later post-process randomly simulates effects like gravity, winds or sunlight direction by bending and warping branches. Finally, leaves (particles) are added to the branches with no sub-branches. Leaves are determined by shape, orientation, spacing, density, color and location.

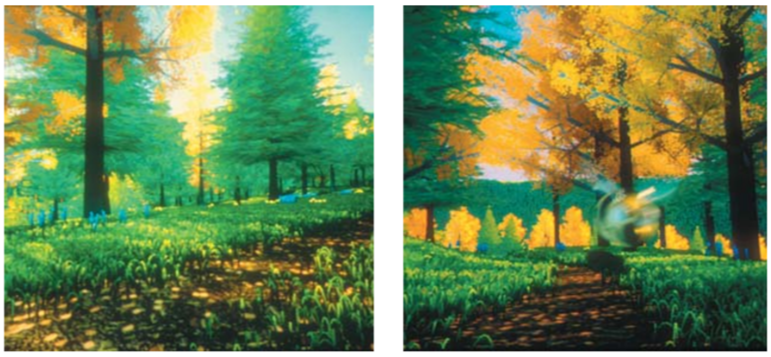


Figure 2.2: Reeves and Blau particle system used for rendering trees. Image from [14].

Oppenheimer [34] uses a method inspired by fractals that recursively calls itself to simulate the branching along a trunk and to generate smaller branches onto large ones, using transformations controlled by a set of parameter distributions such as branching angle, branch to parent size ratio, branches per stem segment... Random variations of the parameters are needed to alleviate the effects of self-similarity of fractals. For the first time, curved sections are used for a more realistic look of the trunk and its branches. Trunk and branches are modeled with generalized cylinders. The tree bark is simulated using a saw tooth function that is modulated by adding Brownian noise.

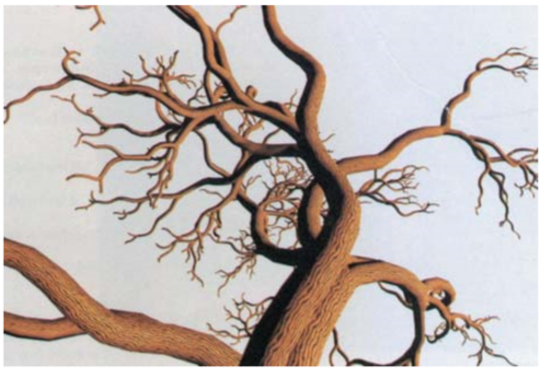


Figure 2.3: Oppenheimer fractal tree. Image from [14].

Previous methods do not have a natural appearance of branching. Bloomenthal [3] focused on the geometrical aspects of tree modeling. Tree skeleton control points are connected using interpolating splines and the surface reconstructed by connecting disks perpendicular to the spline. To reproduce branching faithfully, saddle surfaces (ramiforms) are constructed between the two branching strands (compare figures 2.3 and 2.4). Roots are modeled using "blobs", geometric surfaces defined by a distance to a set of objects, in this case the ramification skeleton of the roots.

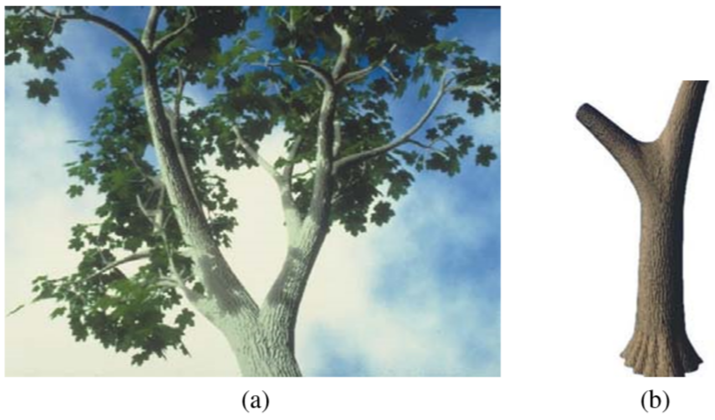


Figure 2.4: (a) Bloomenthal's Mighty Maple. (b) Detailed view of the bark, branching saddle and "blobby" roots. Image from [14].

De Reffye et al. [11] proposed another procedural method inspired by botanical growth rules. The growth of the shoots is simulated in discrete time steps, and each bud carries the probability of dying (stop branching), resting or branching out. They admit that it takes a considerable knowledge of both botany and of their model to create images with great fidelity to nature.

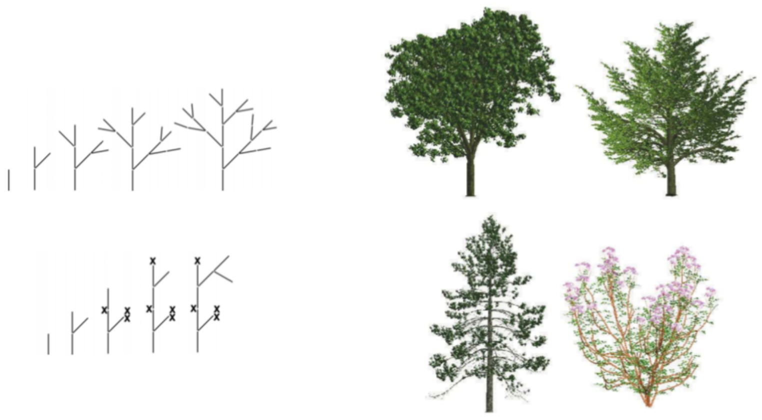


Figure 2.5: Left, randomized growth simulation using two different sets of parameters (dead shoots are marked with ×). Right, some results. Image from [14].

Weber and Penn's work [53] tries to find another method that captures the approximate appearance of the trees instead of using botanical principles. Their procedural method uses a very large amount of parameters (about 80) which is listed in the appendix of their article. Fundamental parameters are, for instance, the overall appearance of the tree (as an enclosing geometry), the size of the lower part of the tree without branches, the number of branching levels, and the shape of the foot of the trunk. For each of the maximal three branching levels, additional parameters are indicated, using an average value with a respective range of variation: vertical branching angle to the father branch, deviation angle, length relative to the father, number of branches, and phototropism (termed as a curve parameter). The results, however, look very realistic.

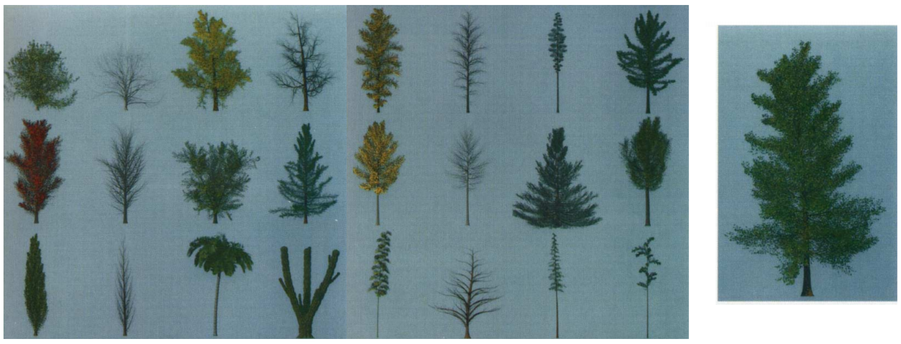


Figure 2.6: Trees generated by Weber and Penn, and a detailed view of a Black Tupelo tree. Images from [53].

In a more recent work, Runions et al. [48] explore a modeling approach they call Space Colonization, in which the competition of branches for space - instead of the definition of a recursive branching process - plays the major role. This idea, which was already present in the early cellular automata models, was also present in the work of Rodkaew et al. [47]: particles were distributed on the shape of a tree crown, and their paths were attracted both to their neighbors and towards the tree base. The tracked converging paths of these particles formed the tree. However, this approach generates branches from the tips downwards, disregarding the biological processes. The Space Colonization algorithm of Runions et al. follows a natural, base-to-leaves order.

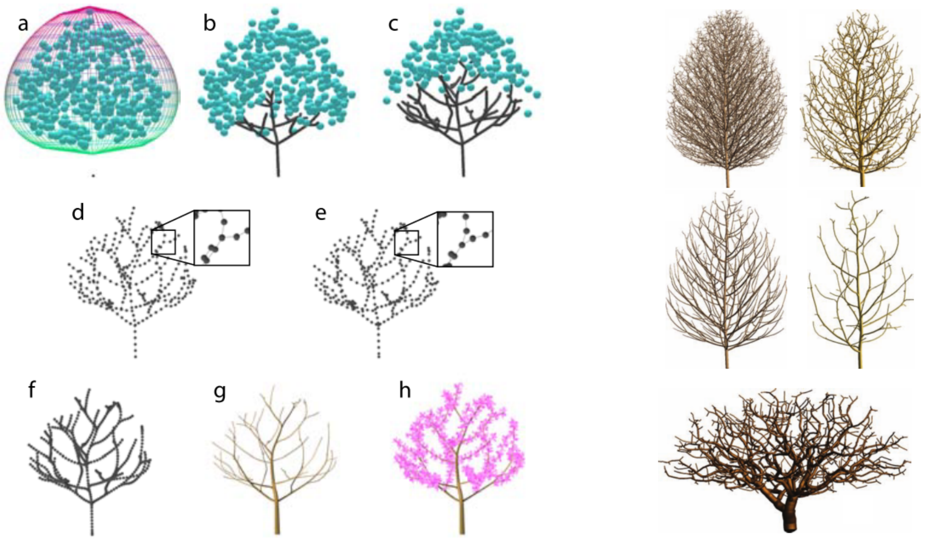


Figure 2.7: Overview of Runions et al. method: (a) initial envelope with attraction points, (b, c) two steps of the skeleton generation, (d) node decimation reduces the amount of geometry, (e) node relocation improves the overall appearance, (f) subdivision creates more smoothly curved limbs, (g) tree geometry, (h) addition of organs. Right, trees generated using this algorithm. Images from [48].

The Space Colonization algorithm of Runions et al. (see figure 2.7) starts with a definition of the tree crown envelope, and fills it with hundreds or thousands of attraction points (a). These points signal the availability of empty space for growth and will be removed when a branch grows close to them. The tree skeleton is formed by an iterative process beginning at the base node of the tree. In each iteration, new nodes extend and branch the skeleton towards nearby attraction points (b, c). The process ends when all attraction nodes have been removed, when no skeleton nodes have attraction points close to their radius of influence or when a maximum number of iterations is reached. Then, the tree skeleton may be edited (d, e, f). The geometry of the branches (g) is generated using generalized cylinders as in [3]. Finally, the leaves and organs are attached to the model (h). Figure 2.8 provides more details about the branching algorithm.

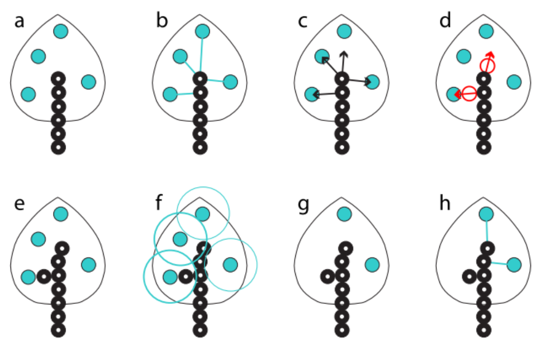


Figure 2.8: Space Colonization algorithm. (a) The tree consists now of six nodes (black) and four attraction points (red). (b) Each attraction point is associated with its closest node inside a radius of influence. (c) Influence vectors from a node to its associated attraction points. (d) Normalized sum of influence vectors (red arrows) provides new node locations (e). The neighborhoods of the two leftmost attraction points have been penetrated (f), so they are removed (g). A new iteration begins (h). Image from [48].

Palubicki et al. [36] also propose a self-organizing model for plant development. In addition to the space colonization, they also model the competition for light using a shadow propagation algorithm, proposed also by Palubicki in [35]. In this algorithm, the space is discretized using a grid, and each bud casts a penumbra pyramid onto voxels underneath. The optimal growth direction is then computed as the negative gradient of the shadow values of the cells, or by selecting the voxel inside the growth range with the minimum shadow value. The results of space and light availability are used to control the fate of the buds: produce a new branch, produce a flower, remain dormant or abort.

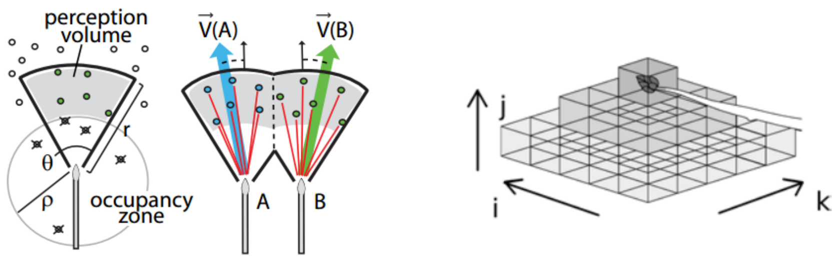


Figure 2.9: Palubicki et al. self-organizing trees. Left, evaluation of space available for colonization. Right, shadow propagation. Images from [36].

2.1.2 Rule-based modeling: L-systems

The L-systems take their name from their author Aristid Lindenmayer, who described morphological forms of plants using string rewriting systems [26], a subset of rule-based systems. In a rule-base system, contrary to a procedural method, a formal rule basis is used to transform an initial state into a final state by applying a number of changes or substitutions. One of the essential differences with procedural methods is the parallelism of the replacements, which allows the modeling of mechanisms such as the exchange of signals which are otherwise difficult to achieve. Often, L-systems provide a compact description for complex final conditions. Prusinkiewicz and Lindenmayer's book "The algorithmic beauty of plants" [40] examines many aspects of plant modeling using L-systems. Since this is one of the most popular methods to generate synthetic vegetation, a brief introduction follows. A description of other rule-based methods such as Iterated Function Systems or the Object Instancing paradigm can be found in [14].

An L-system is a string rewriting system defined using a formal grammar $G = (V, \omega, P)$, where V is an alphabet, ω a nonterminal string (axiom), and P a finite set of replacement rules (productions). A production $p \in P$ is written as $\phi ::= \chi$, where $\phi, \chi \in V^+$. If ϕ has length 1, i.e. is a character of V, the L-system is *context free*. Otherwise, if $\phi = \tau a \nu$, where $a \in V, \tau, \nu \in V^+$ and the length of τ and ν is k and l, we speak of a context sensitive (k, l)-system. If for any $\phi \in V^+$ there exists at most one production rule with ϕ at the left side, the L-system is deterministic. Otherwise, when random processes are needed, stochastic L-systems extend the grammar $G = (V, \omega, P, \pi)$ with the set of probabilities $\pi : p \rightarrow (0, 1]$. When ϕ is found on the left side of one or several rules, one of them is randomly selected according to their probabilities.

The plant geometry is obtained from the final string. To interpret this string, the turtle metaphor was proposed by Prusinkiewicz [40]: a virtual turtle is moved over the drawing plane (or 3D space) and during this movement a line may be drawn tracking its position. For example, we can define the following commands:

F	move into current direction d units, drawing a line
f	move into current direction d units, without drawing a line
$+$	increase current angle by δ
$-$	decrease current angle by δ
[	store current state (position and direction) on the stack
]	load state (position and direction) from the top of the stack

Table 2.1: Example set of commands for L-system drawing

And use them to draw the following L-system: $G = (V, \omega, P)$ with alphabet $V = \{F, +, -\}$, axiom $\omega = F - -F - -F$ and a set of one production rule $P = \{F ::= F + F - -F + F\}$. Setting $d = 1$ and $\delta = 60°$, the axiom produces an equilateral triangle. The defined rule is the generator for the Koch curve fractal. And each derivation step produces a new iteration of the Koch snowflake:

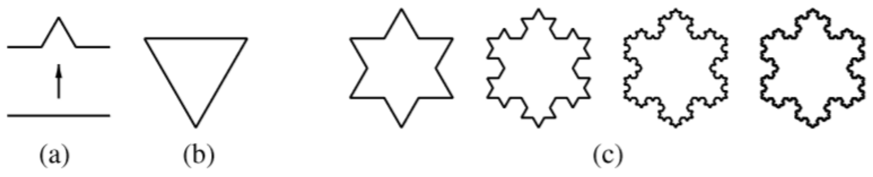
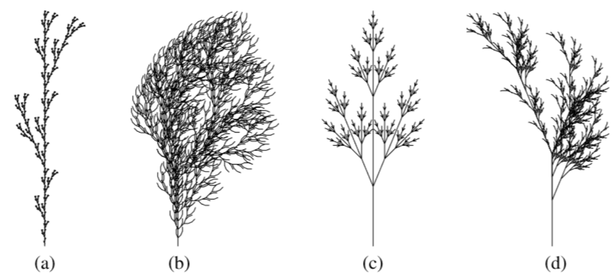


Figure 2.10: Koch snowflake generated using the described L-system. (a) production rule interpretation, (b) axiom interpretation, (c) result after 1, 2, 3 and 7 rewritting operations. Image from [14].

Branching structures require a push-down automaton to process the string sequence. Using the symbols [and] we can push and pop the turtle state into a stack structure. Also, we can add non-drawable symbols to represent branching nodes instead of rewriting the previous branches. The following figure shows some examples:



Part	n	δ	ω	P
(a)	5	25.7°	F	$F ::= F[+F]F[-F]F$
(b)	4	22.5°	F	$F ::= FF - [-F + F + F] + [+F - F - F]$
(c)	7	25.7°	X	$X ::= F[+X][-X]FX, F ::= FF$
(d)	5	22.5°	X	$X ::= F[[X] - X]F[+FX] - X, F ::= FF$

Figure 2.11: Branching structures generated using L-systems: (a) and (b) are based on edge rewriting, (c) and (d) use the non-drawable symbol X to simulate node rewriting. Image from [14].

Obviously, L-systems are also extended to 3D using additional symbols and parameters. The examples seen so far always produce the same structures. In order to introduce the variability present in natural structures, stochastic and parametric L-systems can be used. As defined before, in a stochastic L-system the same left side appears in more than one rule and the rule to be applied is selected randomly according to a set of probabilities. A parametric L-system allows dynamic modification of the parameters during expansion. The commands use a parameter vector $\mathbf{w}$, so $F(\mathbf{w})$ would mean into current direction $d(\mathbf{w})$ units, and so on.

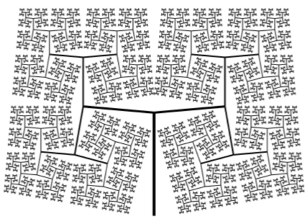
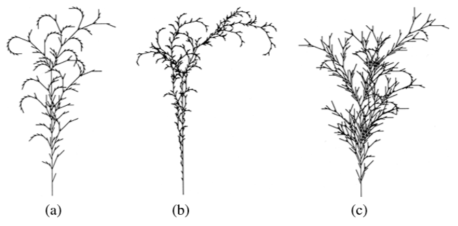


Figure 2.12: Binary tree generated from a parametric L-system with a single production rule $A(s) ::= F(s)[+A(s/R)][-A(s/R)]$ applied 10 times to the axiom $\omega = A(1)$. $\delta = 85°$, $R = 1.456$. Image from [14].

Context-sensitive L-systems allow signal exchanges that can control the growth in different parts of the plant. In order to make the context more visible, traditionally L-systems used the notation $\tau < A > \nu ::= \chi$, where $A \in V$ and $\tau, \nu, \chi \in V^+$. See figure 2.13.

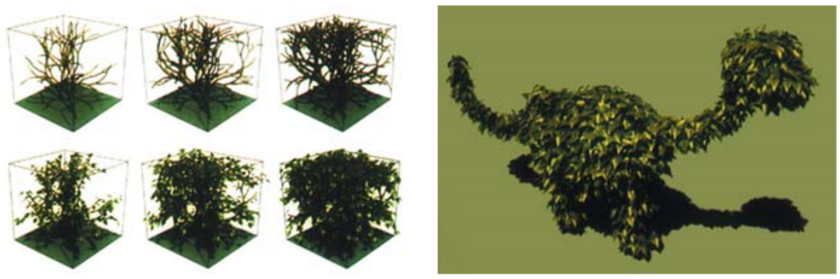


Part	(a)	(b)	(c)
n	30	24	26
δ	22.50	25.75	22.50
Ignore:	+-F	+-F	+-F
ω	F1F1F1	F0F1F1	F1F1F1
$0 < 0 > 0 ::=$	1	1	0
$0 < 0 > 1 ::=$	1[-F1F1]	0	1[-F1F1]
$0 < 1 > 0 ::=$	1	0	1
$0 < 1 > 1 ::=$	1	1F1	1
$1 < 0 > 0 ::=$	0	1	0
$1 < 0 > 1 ::=$	1F1	1[+F1F1]	1F1
$1 < 1 > 0 ::=$	1	1	1
$1 < 1 > 1 ::=$	0	0	0
$* < - > * ::=$	-	-	-
$* < + > * ::=$	+	+	+

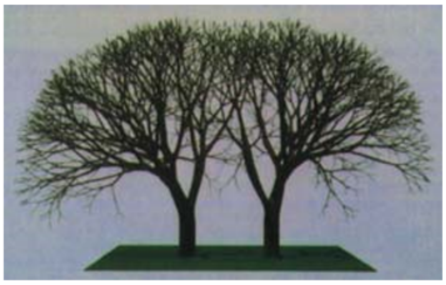
Figure 2.13: Context-sensitive L-systems examples. Note (a) and (b) are almost identical but produce very different results. Image from [14].

L-systems have also been extended to use time information in timed L-systems and differential L-systems [39], which permit the simulation of growth procedures through the use of differential equations for the parameter set. Environment-sensitive L-systems [41] can simulate local aspects of the environment (for example, pruning) with an inquiry symbol based on the position. Open L-systems [30] also enable communication modules for sending and receiving parameters, so interaction between branches or trees such as light competition can be simulated. Finally, user-defined functions and modeling tools can also be inserted [42], facilitating the reproduction of a particular plant.

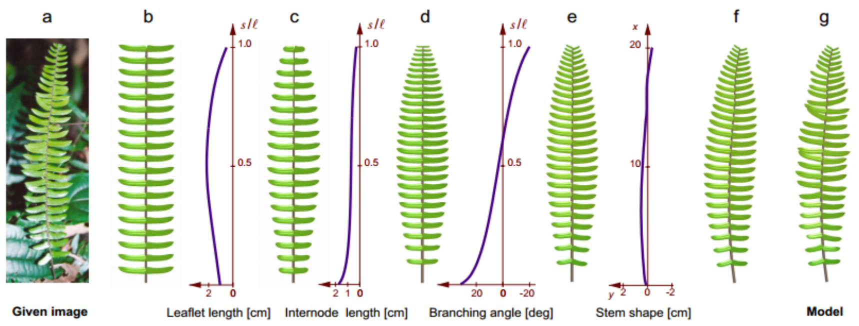
(a) Animation with differential L-systems



(b) Pruning with environment-sensitive L-systems



(c) Tree interaction with open L-systems



(d) Modeling a fern leaf using positional functions

Figure 2.14: Examples of various extensions of L-systems. Images from [14].

2.1.3 Rule-based object production systems

The rule-based procedures such as the L-systems we have seen are a very powerful method that allows the generation of a large variety of plants. However, the use of L-systems is not very intuitive, and the production of a precise plant is a difficult process even for skilled users. Small changes can cause a complete modification of the total shape, and after a parameter change the whole expansion process must be worked through, and the geometry production is expensive. On the other hand, procedural methods have the opposite characteristics: only a very limited number of plants can be modeled with a method, but the parametrization is usually straightforward and intuitive.

Therefore, some tools like the Xfrog modeling system by Lintermann and Deussen [27] offer a combination of the two approaches in the so-called rule-based object production approach. Here, a plant is represented by connecting components. Each component generates parts of the plant geometry such as leaves, stems or simple geometric primitives by using procedural methods. These components are connected in a directional graph and using multiplication components, which represents the rule system.

Simple	Creates and transforms simple geometries (cubes, spheres, cylinders, torii, point sets). All other components derive from this one.
Revolution	Produces a surface of revolution from the user-specified outline.
Leaf	Produces the geometry for leaves and petals.
Horn	Basis for stems, branches, twigs... Produces point sets or generalized cylinders.
Tree	Produces a generalized cylinder like the Horn, but can also be instructed to multiply its children nodes as branches of its stem. Its parameters can control the branching structure (angles, sizes, ...)
Hydra	Multiplies subsequent components and places them in a star-like arrangement.
Wreath	Arranges its successors on a ring.
Phiball	Multiplied components are arranged on a section of a sphere by the golden section algorithm (like sunflower seeds).
FFD	Free form deformation of the shape by user-defined functions.
Hyperpatch	Free form deformation of the shape by moving control points of a 3D Bézier patch (degree 1 to 3).
World	Allows the definition of functions for gravitropism and phototropism.
Camera	Scene camera transform and projection.

Table 2.2: Components of the system, from [27]

The following figure shows some examples of plants modeled using this system, as well as their associated graphs:

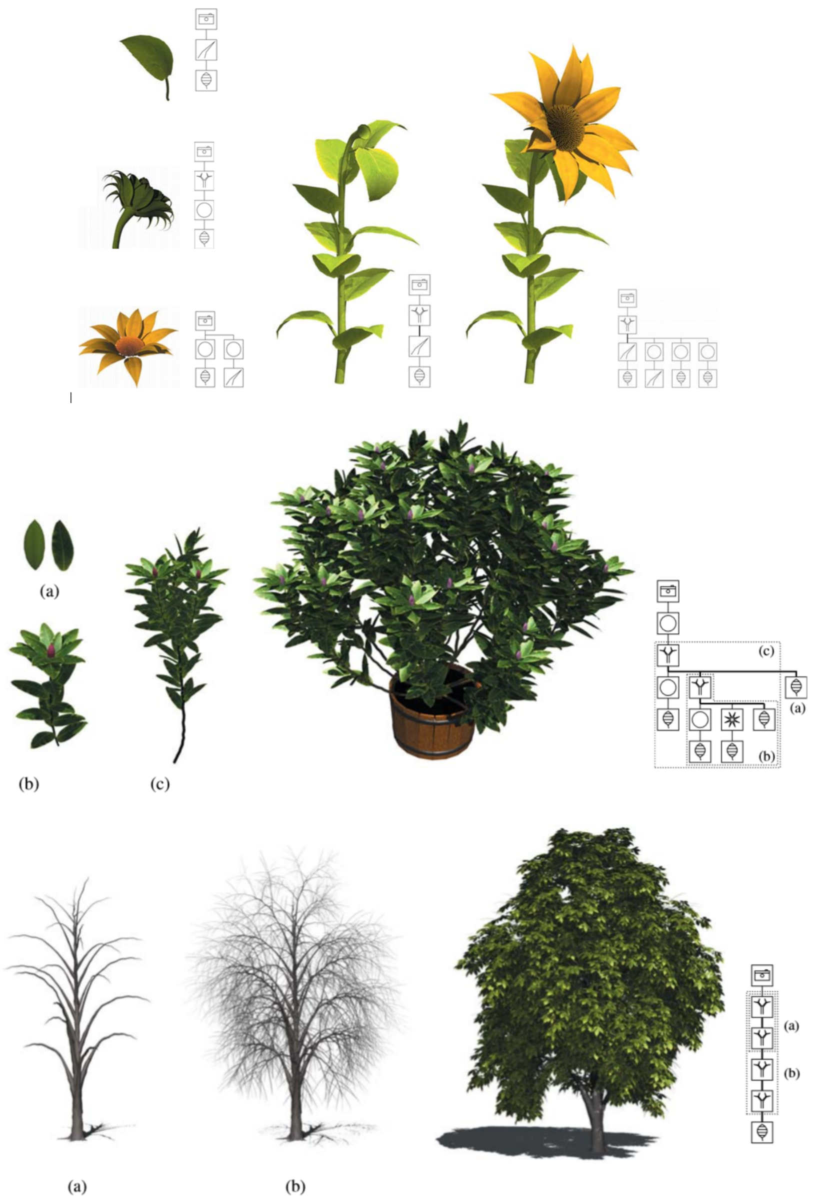


Figure 2.15: Examples of plants generated with the system. Images from [14].

2.2 Reconstruction of real trees

Instead of automatically generating a tree as we have seen in the previous section, some works use real data - such as photographs or point clouds - to generate the models. Obviously, these methods have the advantage of producing realistically looking results and often do not need to spend some trial and error time tuning parameters of an algorithm nor expertise in the rule definition. The disadvantage is that they need to capture input data. Moreover, the traditional methods of scanning objects cannot be applied because the small geometries of all the leaves introduce many high frequency details. Trees have an interesting structure at different scale levels, at the upper scale of its branches and at the lower scale of the leaves. Therefore, several approaches have been introduced to reconstruct tree models from photographs or scanned point clouds.

2.2.1 Trees from photographs

Most of the reconstruction methods we have found use a picture set as input, probably due to their ease of acquisition.

In the work of Shlyakhter et al. [49], the trunk and major branches are recovered from a set of instrumented photographs, and then an *L-system* is applied to grow the small branches and leaves. The input consists in a set of 4 to 15 images with known camera poses, which are then manually segmented. The second stage uses the silhouettes obtained previously for each photograph and computed the visual hull of the tree. To do so, each silhouette polygon is back-projected using the camera information and a cone-like shape is obtained. The intersection of all these 3D shapes is the approximation of the tree shape. This visual hull is used to compute the tree skeleton as an approximation of the medial axis, giving the main trunk and first levels of branching. To complete the tree with small branches and fine level detail, an L-system is used. Their method uses the skeleton as starting axiom with buds at the last two levels of branching. Also, they ensure similarity with the pictured tree by pruning branches that grow outside the visual hull shape.

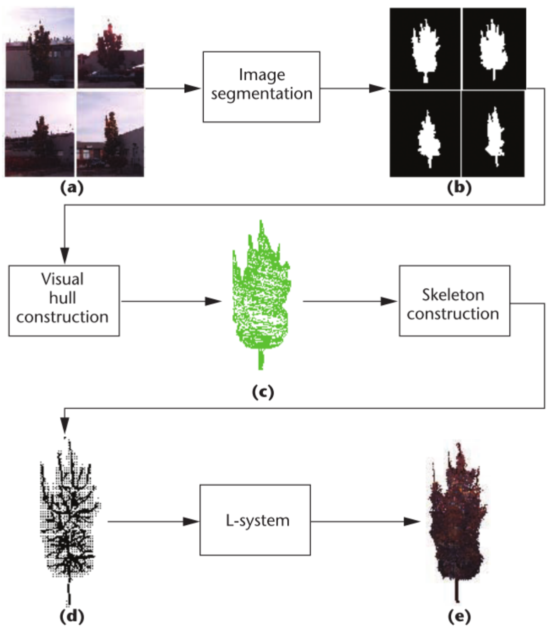


Figure 2.16: Shlyakhter et al. method overview: input images (a) are segmented (b) and a visual hull (c) constructed from their silhouettes. This hull is used to compute the tree skeleton (d) from which the L-system will produce the finer details (e). Image from [49].

Reche et al. [45] developed a volumetric approach for individual tree reconstruction and rendering from calibrated photographs. For each image (they use about 20-30), an alpha mask is extracted. This process requires user intervention to segment foreground and background, and produces an approximation of the visibility coverage for each pixel. Then, a recursive grid containing an opacity value at each cell is constructed from these masks. For each pixel of each alpha mask, the set of intersected cells is found using ray casting. From an optimization procedure, an estimation of the opacity of all the cells is found after a few iterations. In order to render this volume, each cell is assigned a small billboard extracted from the original photographs (using a heuristic approach to overcome opacity in the inner cells). Finally, they use a direct volume rendering algorithm. It is worth to mention that the authors note their method is particularly suited for trees with relatively sparse foliage.

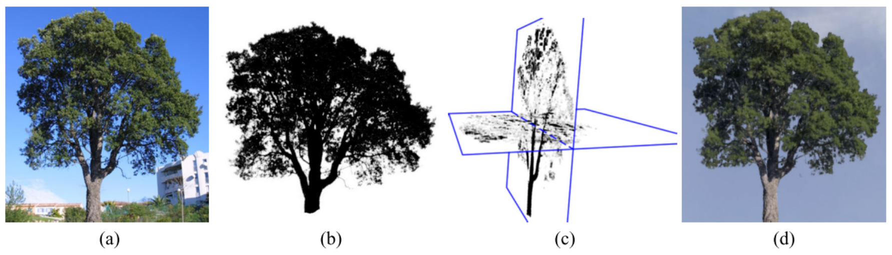


Figure 2.17: Reche et al. method overview: one of the input images (a), its transparency mask (b), two slices of the volume grid containing opacities (c) and a synthetic volume rendering (d). Image from [45].

Neubert et al. [32] reuse this idea of a volume grid and simulate a 3D particle flow to model the branches. Their method starts from an automatic matting of each input image, which can be seen as a tree density estimation. On these images, a set of branch seed points is randomly selected from the silhouette or user defined, and a path to the trunk is found. This defines a skeleton for each image that the authors call attractor graph. Then, using the density images and an algorithm like the one presented by Reche et al [45], a coarse density grid is computed. This tree density grid is used to initialize the positions of a set of 500-2000 particles. Then, a particle simulation will move these particles and create the tree branches. The force acting at each step on each particle is defined by the attraction of nearby particles as well as the direction field computed from the set of 2D attractor graphs. Particles are merged when they get close. Once the branching structure has been computed, the model of the branches is generated and populated with leaves extracted from photographs according to the density grid. The user is also able to sketch regions to add additional branching or foliage coverage.

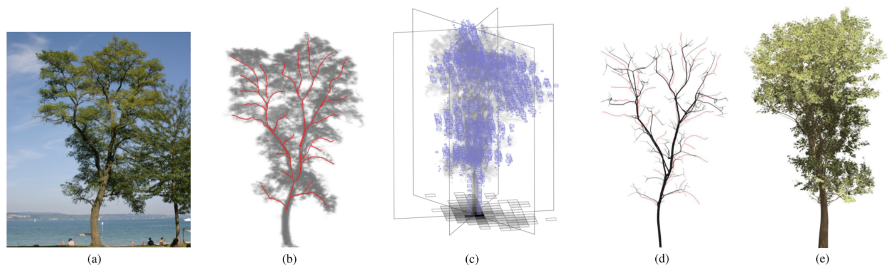


Figure 2.18: Neubert et al. method overview: one of the input images (a), its density estimation and attractor graph in red (b), the density grid (c), branching structure obtained from the particle simulation (d) and final rendering (e). Images from [32].

In Tan et al. [52], a structure from motion algorithm is run to produce a 3D point cloud from a set of 10-20 captured photographs. These points are segmented onto branches or leaves, depending on their color and position, and a set of visible branches is extracted. The branch reconstruction process assumes the trunk and a representative set of branches are visible. Therefore, the occluded branches can be generated using the visible ones as replication block examples. Finally, to populate the tree with leaves, input images are analyzed to produce a set of regions with homogeneous colors, which will be the candidate leaves (usually represented as ellipsoids). Then, these regions are projected to find a close

3D point from the previously extracted point cloud or a branch and the leaf is attached to the model. After all the input images have been processed, more leaves are synthesized to the lowest density areas. Although the method can be fully automatic, the user can interact during the matting, branch editing and the leaves segmentation/clustering.

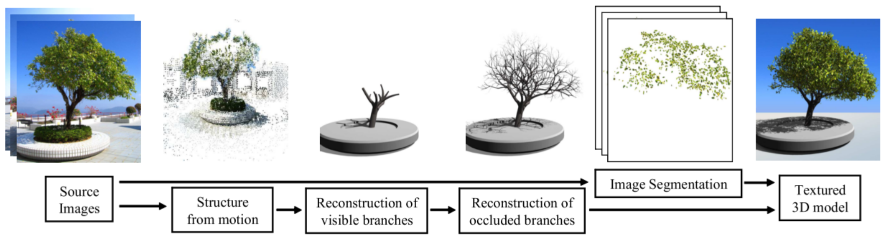

Source Images
Structure from motion
Reconstruction of visible branches
Reconstruction of occluded branches
Image Segmentation
Textured 3D model

Figure 2.19: Tan et al. method overview. Image from [52].

In a more recent article, Tan et al. [51] describe a new method that allows tree modeling from a single image and some user interaction. On the image plane, the user first draws one stroke following the shape of the crown, and the foliage is segmented from the closed region of this stroke using a Gaussian mixture model. Then, more strokes are drawn to mark the visible branches. The interface automatically tries to follow visible branches close to the user strokes, and the user can correct this result adding or removing strokes. The set of 2D branches is converted to 3D using a greedy strategy: adjusting their orientation such that distance between them is as large as possible. Like in their previous work, a library of branching structures is built from this set of visible branches and possibly user defined patterns. Then, an iterative process selects a branch and replaces it with a subtree. This process tries to minimize the distance from the branches to a set of point attractors sampled on the segmented foliage region in the input image. In order to ensure a balanced 3D growth, a set of extrapolated 3D attractors is also computed by rotation 90 degrees the branch joints. The iterative process alternates between minimizing distance to image attractors and distance to these 3D points. Finally, the leaves are added using rectangles textured with the input image foliage. Branches and leaves that project outside the segmented foliage region are pruned.

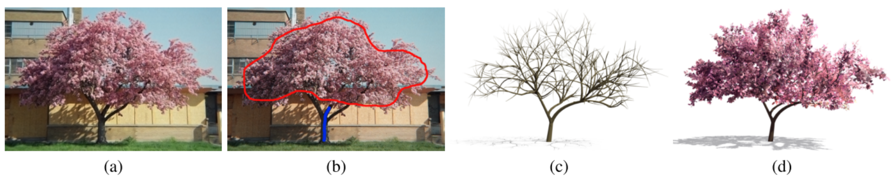


Figure 2.20: Tan et al. single image method overview: input image (a), user-drawn strokes (b) for crown (red) and branches (blue), generated branch structure (c) and final tree model (d). Image from [51].

Sun et al. [50] proposed a method to combine rule-based modeling and image-based reconstruction to create lightweight trees from two input images. From one of the source images, the tree branches are segmented using an automatic algorithm and thinned to find the 2D skeleton of the tree trunk. The result is a set of branch segments, which are then classified as real branching points or fake branching relationships introduced due to occlusions. Once the branching structure of the 2D image is recovered, the second image is used to find the branch correspondence and recover the 3D position of the start and end points of the branches. This branching structure is then analyzed to produce an axiom, a set of productions and parameters (length and width of branch, branching angle and divergence angle) for a parametric L-system. The final model is produced from this

L-system. They also generate a Web3D file from this L-system that allows an efficient transmission of the model.

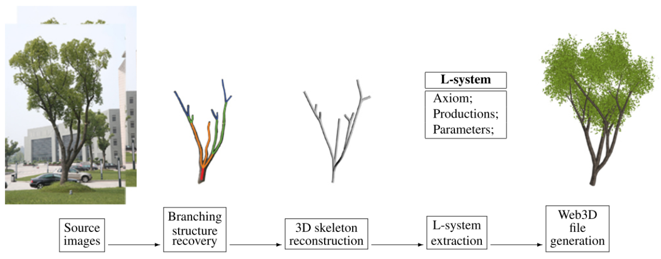


Figure 2.21: Sun et al. method overview. Image from [50].

For the case of plants with well differentiated leaves, Quan et al. [44] showed a method to realistically reconstruct the plant model. Running a structure from motion algorithm on the input images (30-45) a 3D point cloud is obtained. Then, the individual leaves are segmented using a graph-based optimization using the 3D point positions and their projections on the images to identify continuous regions. The user also assists this process by refining the segmentation. Next, an example leaf is selected by the user. A flat version of it is fitted onto each group of points segmented as a leaf to determine its orientation, and then deformed to match the boundary. Input images are used to modify the texture. Finally, the user is asked to model the branch structure of the plant.

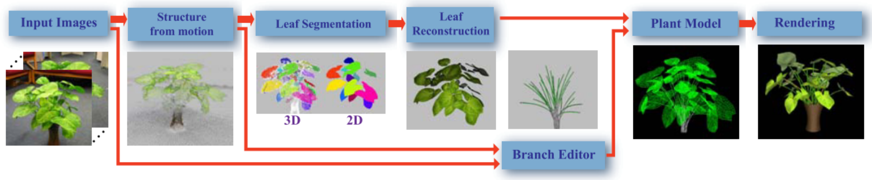


Figure 2.22: Quan et al. method overview. Image from [44].

Quan et al. method is able to produce accurate leaves, but does not scale to large dense foliage areas. On the other hand, the previously described methods or the methods we will see in section 2.2.2 capture large-scale branching structure of trees but arbitrarily synthesize the fine details. In a recent work, Bradley et al. [5] proposed a method aimed to reconstruct details as they appear on the real tree.

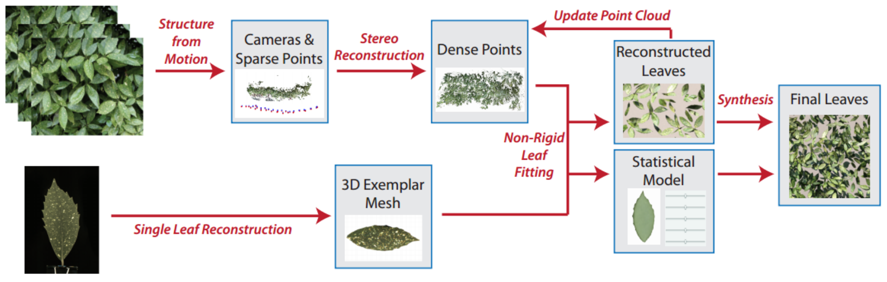


Figure 2.23: Bradley et al. method overview. Image from [5].

In their work, the input consists of a small number of overlapping images of the foliage to reconstruct and at least one exemplar leaf, which they obtain by scanning a real leaf pruned from the plant. From a structure from motion algorithm, they estimate the camera parameters and an initial sparse 3D point set. Then, a custom multi-view stereo algorithm generates a dense point cloud. Then, the exemplar leaves are aligned to the point cloud data. During this alignment, the leaf exemplars are allowed some deformation to match the leaf points shape. From this deformations, a statistical model of shape variations is learned. Similarly, a color statistical model can also be built. Finally, the statistical models are used to synthesize new leaves which could not be reconstructed or were occluded in the point set.

2.2.2 Trees from scanned point clouds

Other authors propose solutions based on tree reconstruction from a point cloud. Some of the approaches we have already seen in section 2.2.1 ran a structure from motion algorithm to the input photographs and obtained a 3D point cloud [44, 52, 5]. However, others start directly from a laser-scanned point cloud. In recent years, these techniques have become more popular due to the increasing number of scanned datasets produced, specially from city streets. This section gives a few insights onto some representative works.

Xu et al. [58] method first builds a graph connecting each point to its neighbors. Then, starting from the root point (which is manually selected or automatically by taking the lowest point) the shortest distance path from it to each other point is computed. This set of lengths is then quantized, and connecting the centroids of neighboring bins produces the skeleton of the tree. Since several connected components may appear in the initial graph, an appropriate position and angle is found for each of the main branches to be attached to the main trunk. The points in the subgraphs that are not connected to the main skeleton are considered to be leaves, and clustered according to some species-specific parameters to define leaf locations. Then, fine branches are synthesized to reach these positions. Finally, the tree mesh is constructed around the skeleton and the leaves are textured.

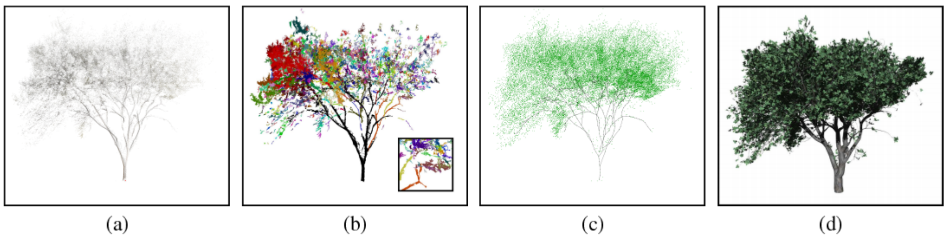


Figure 2.24: Xu et al. method overview: original scanned point set (a), connected graph components identified (b), reconstructed skeleton and identified leaf positions (c), final model (d). Image from [58].

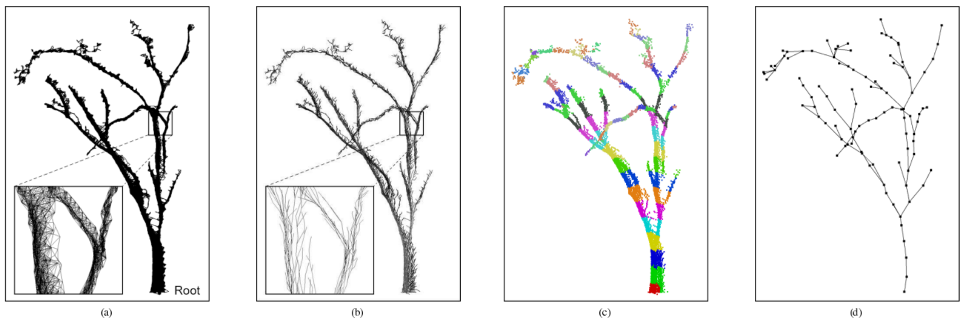


Figure 2.25: Xu et al. main skeleton production: neighbors graph (a), shortest distance paths to root (b), clustered distances to root (c) and skeleton formed by connecting neighboring bins (d). Image from [58].

Livny et al. [29] define a fully automatic method to reconstruct a scanned point set containing one or more trees. Like in the method we saw before, all scanned points are connected to their neighbors with weighted edges according to their Euclidean distances and the minimum-weight spanning tree is extracted from this graph. If the input point set contains different trees, the selected root points of each one are connected with zero-weight edges and removed after the spanning tree has been created. Then, these tree skeletons obtained need to be smoothed to obtain a more natural look. They generate an orientation field for each skeleton by minimizing the sum of directional differences between adjacent edges weighted by edge importance. The importance of an edge is equal to the size of its subtree. This orientation field will be used to optimize the spatial embedding of the skeleton, by formulating a least squares problem. Finally, the skeletons are inflated into tree geometry and branches are populated with textured leaves. The authors note that trees that have very dense crowns, leading to large-scale occlusion of the interior branches, cannot be faithfully reconstructed.

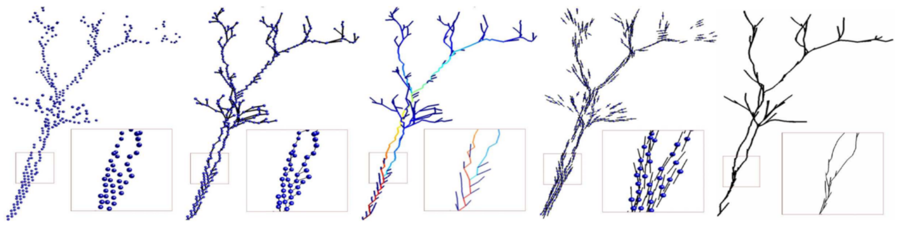


Figure 2.26: Livny et al. trunk reconstruction, from left to right: input points, minimal-weight spanning tree, colored importance weights of the edges, orientation field, smoothed tree structure. Image from [29].

In an extension to their work, Livny et al. [28] propose a lobe-based representation to reconstruct and lightly store and transmit trees. They show how to convert a point cloud a lobe-based representation and how to reconstruct the tree by texturing the lobes and producing the branch geometry. Using three parameters that depend on the tree species, they follow the edges in the tree graph and identify points in which the average edge length is bigger than the expected diameter of the branches. These points are unlikely to be connected to the branching structure, so they cluster them in lobes. Each lobe is represented as the triangulated surface produced from its α-shape[1]. In order to reconstruct the tree from the lobes representation, each species has a defined set of branch patches with anchor points. Starting from a branch patch inside the lobe, new fitting patches are anchored iteratively until the lobe volume is covered depending on the desired level of detail.

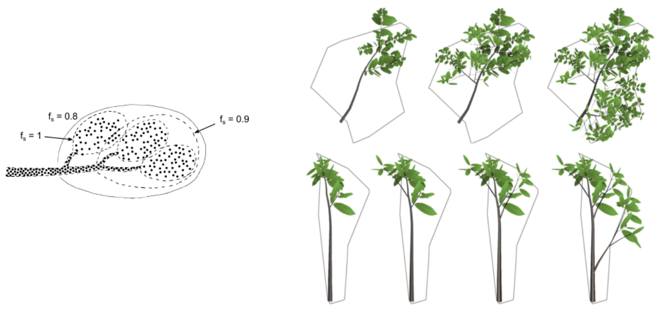


Figure 2.27: Linvy et al. texture lobes. Left, the effect of one of the parameters on the final shape of the lobes. Right, lobe reconstruction from patches. Images from [28].

[1] α-shapes are extensions of convex hulls that allow non-convex envelopes to be created

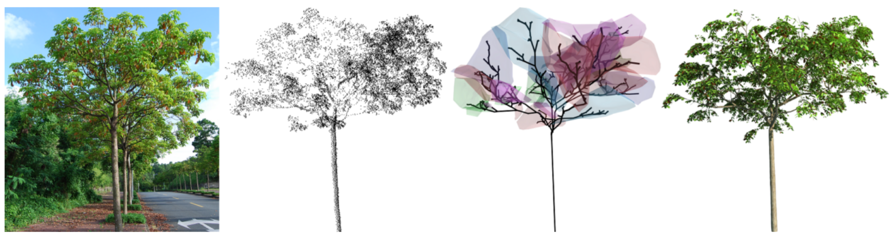


Figure 2.28: Linvy et al. texture lobes, left to right: comparison picture, point set, lobe-based representation, and full tree after lobe texturing. Images from [28].

2.2.3 Sketch-based reconstruction

Instead of capturing a real tree, other authors have researched the possibility to allow artists to directly draw the desired shape and appearance of the desired tree. Here we will discuss a pair of these approaches, since they provide some interesting ideas on how to infer a 3D structure from a 2D sketch and how to add fine details.

Okabe et al. [33] propose a user interface to sketch trees. To model a tree, the user starts drawing in 2D the branch structure. When the user finishes, the sketched branches are converted into a 3D tree skeleton. The density of branches can be increased by the user successively clicking on parent branches. Next, the user can start sketching and placing leaves, and the system automatically infers and proposes leaf arrangement patterns the user can choose to fill the branch. The arrangement of a branch can be copied to the rest of branches of the tree. An additional feature allows the user to duplicate the tree by sketching a trunk and a silhouette shape, and the branches of the copy are adjusted to fit this shape.

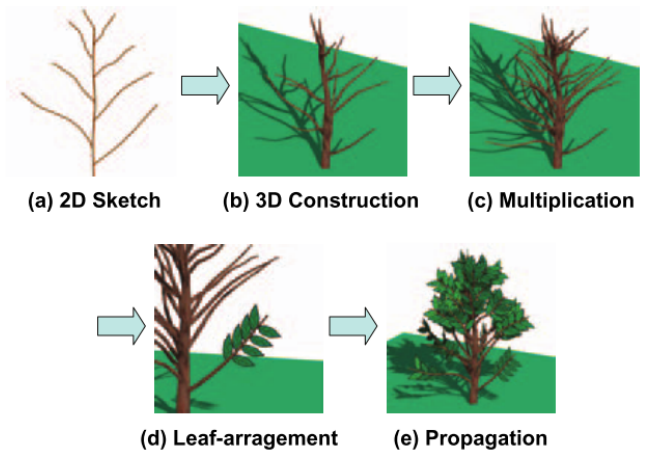


Figure 2.29: Okabe et al. sketching process. Image from [33].

To convert the 2D sketch into a 3D tree model, the authors propose a greedy strategy. Branches are placed one by one, and the sibling order is random. They use the following constraints: the branch projection of branches onto the 2D plane must fit the sketch, branches must be located inside the 3D convex hull obtained from the sketch 2D convex hull by sweeping a circle along it, and the distance between branches must be as large as possible. Other typical relations such as the size and length of child branches with respect to the parent branch are also taken into account. By doing this method, the tree only resembles the sketch from a front view, lateral views can be very different. The authors make the observation that users often draw those branches pointing sideways from the main trunk and omit the branches pointing forward of backwards. Therefore, they propose to position the branches deviating only 45 degrees from the sketch plane, do the same again after rotating 90 degrees the sketch along the trunk direction, and finally merging the two resulting trees.

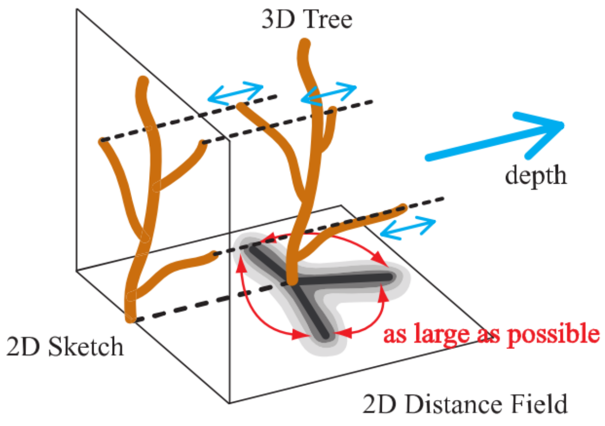


Figure 2.30: Okabe et al. branch positioning. Image from [33].

The approach of Wither et al. [57] builds on the methodologies used by botanists and artists when they create 2D drawings of trees incrementally. Also, instead of drawing the full tree with details, they usually draw the shape and specify details on certain areas using zoom rectangles. Thus, the authors design a system that starts from a crown silhouette and only needs to refine a subpart of the tree, the style of this refined area is copied to the rest of the tree. Starting from the trunk and crown silhouette sketch, the system infers the first level of branching by connecting the trunk with the centroids of the silhouette bumps using information from the crown geometric skeleton and botanical rules. Each detected bump on the sketched silhouette also indicates a crown of a sub-branching system. The system provides a simple version of these that serves as construction lines for the user to refine these regions as well as guides during style copying from one branch to another. The process finishes when the user draws the silhouette of a leaf. To construct the 3D model, the system also assumes like in the work of Okabe et al. [33] that users draw branches close to the sketch plane and generates new branches to cover all the angles while maximizing distances between branches.

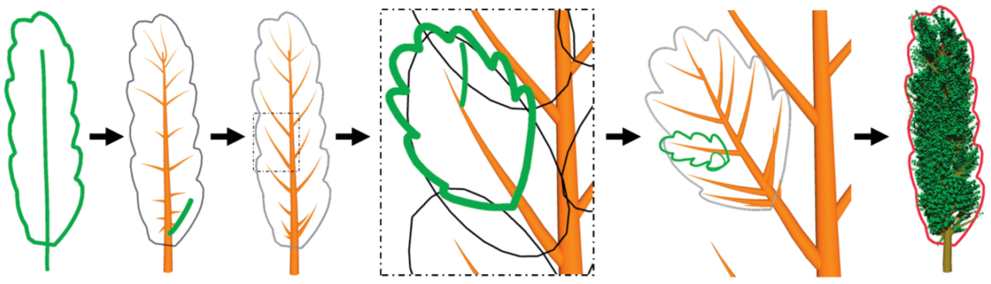


Figure 2.31: Wither et al. sketching system. The user first sketches the main trunk and the silhouette. Then the first level branching is inferred. The user redraws a branch, and this change is propagated to the rest of branches. When the user zooms in, the outline of the simple inferred sub-silhouette appears, serving as the basis to draw the structure of the next level. When the user zooms out, styles are transferred between branching systems and branches are positioned in 3D, resulting in a full 3D tree. Image from [57].

2.3 Tree representations

As we have seen, trees are complex objects and their representation usually requires a large number of polygons. Some characteristics that influence the rendering realism of a plant [4] are: the number and variety of organ shapes, the number of organs and the spatial organization of these organs depending on the species. Also, a plant aspect drastically varies with observation distance: at close scale we see a branching system with detailed organs, but at large distances the aggregation of individual details yields an overall impression of a fuzzy volume.

This is why most classical rendering optimization and simplifications techniques that are successfully applied on regulars objects usually produce non-adequate results for plants, and a wide range of techniques tailored for plants has been developed. Boudon et al. made a survey of these techniques in [4], and a short summary is provided here for completeness of this chapter.

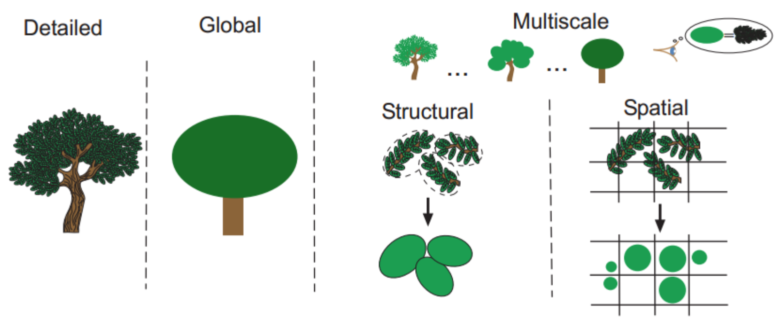


Figure 2.32: Classification of tree representations by Boudon et al. Image from [4].

2.3.1 Detailed representations

Detailed representations try to model the tree in the most realistic manner for close views. Trunk, branches and leaves usually require different modeling and rendering techniques. We have already discussed some representations in section 2.1.

Pioneering approaches used cylinders to represent trunks and branches. Bloomenthal [3] proposed the use of generalized cylinders and the saddle structure to improve the realism of branching points. He also proposed the use of implicit surfaces to represent "blobby" structures like the roots, and this approach has been extended and improved to represent the tree branches in later works. Finally, subdivision surfaces have been also used to obtain a smooth surfaces at the branching points.

The bark texture of trunk and branches has been either addressed as a texturing approach and as a mechanical simulation to generate it. In the texturing approaches, one of the main problems is the parametrization of junctions. Some proposed solutions are blending by interpolation between the texture of each branch, or using particle flow along branches to compute the parametrization. Realism of the bark rendering is usually improved using *bump mapping* or *displacement mapping*.

The foliage of the tree is often built from a few leaf representations, each of them being a textured polygon. Other approaches construct more complex leaves using splines, sweep surfaces [42] or from a skeleton or silhouette.

2.3.2 Global representations

These representations provide efficient rendering at distant views, by only representing the overall appearance of the tree. The most common tool to achieve a realtime rendering of forests or scenes with a large number of trees are billboards. Classical billboards use a single quad oriented towards the camera, but there is no parallax when the camera moves and a tree slightly behind another may pop in front depending on the angle. Cross billboards use a small set of fixed quads crossing each other, but artifacts appear when a quad is viewed from a grazing angle.

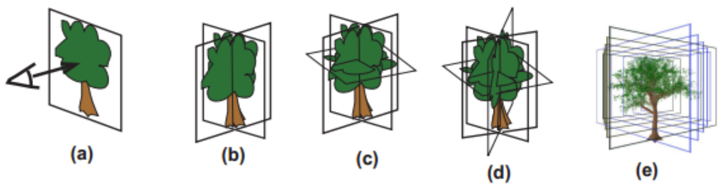


Figure 2.33: Tree representations using billboards: (a) classical, (b) cross billboard, (c) 3-cross billboard, (d) 4-cross billboard, (e) billboard slices. Image from [4].

Jakulin [22] extended the cross billboard representation for the crown, and used traditional geometry rendering for the trunk and limbs. In a preprocess step, several sets of parallel slices are created from various viewpoints, and for each set the crown leaves are assigned to the closest slice. The leaves of each slice are then rendered to a 2D texture. During rendering, the two slice sets closest to the viewing direction are selected, and the slices rendered using the correct transparency and blending. Another extension of billboards was proposed by Qin et al. [43] in a representation they called quasi-3D trees. This representation stores a set of 2D buffers containing the tree billboards for color, normal vectors, relative depths and shadowing.

A more recent approach by Bruneton and Neyret [7] proposes a representation for medium distance views called *z-field*. The tree is rendered from 181 views on the upper hemisphere, and they store the minimal and maximal depths, the ambient occlusion and the opacity. During rendering, the three closest precomputed views are used to reconstruct the tree shape and compute the shading. Although the modeled forests display a variety of realistic lighting effects such as view-dependent reflectance, slope-dependent reflectance, hotspots and silverlining[2], each different tree model requires about 50 MB.

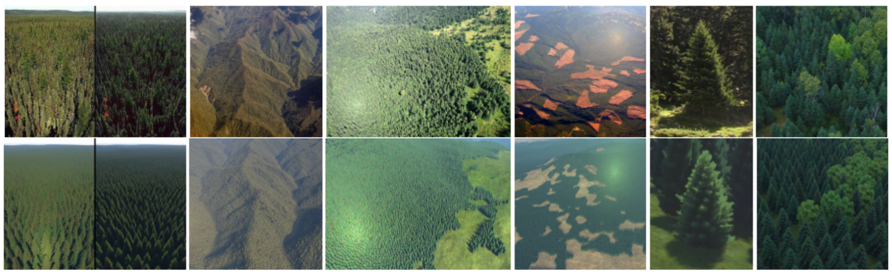


Figure 2.34: Bruneton and Neyret rendering of forests: comparison between photographs (top) and renderings (bottom). Image from [7].

[2]The silhouettes of backlit trees appear brighter than the rest of the tree because they are optically thinner [7].

2.3.3 Multiscale representations

The structure of a tree plays an important role on how its model can be simplified. Its multiscale hierarchy of components define different levels of abstraction that can be used for simplification: trees are composed of a trunk and main branches, each main branch groups smaller branches, boughs are made of leaves or needles, and leaves, needles, boughs and branches resemble each other [4].

Finding repetitive patterns leads to simplifications based on plant structure. More generic methods regroup or merge primitives based on proximity.

Based on plant structure

Tree structures are hierarchical, and thus many techniques base their levels of details on this nature. The disadvantage is the simplification techniques are often highly coupled with the generation process, since specific knowledge about the tree structure is needed to build a multiscale representation. For example, the procedural modeling method could directly include the generation of geometry based on the level of detail. Furthermore, if the generation were fast enough, storage space can be saved by directly generating the models on the fly.

Like in the global representations, image-based representations are also widely used to build a hierarchy of billboards from a single quad representing the whole tree to hundreds of quads for the branches or even to the leaves level.

Point-based rendering has also proven to be useful. This approach builds on the observation that in increasingly complex scenes triangles become smaller than a single pixel, so triangle-based scan-line rendering wastes time in superfluous computations. Thus, the degree of detail can be adapted by adding or removing points. We have already seen this approach in the early Reeves and Blau particle system for trees [46] in section 2.1, and it was later extended by Weber and Penn [53] and [13]. To render a large number of trees, in these approaches the geometry of the branches is progressively reinterpreted: branches become lines and leaves become points.

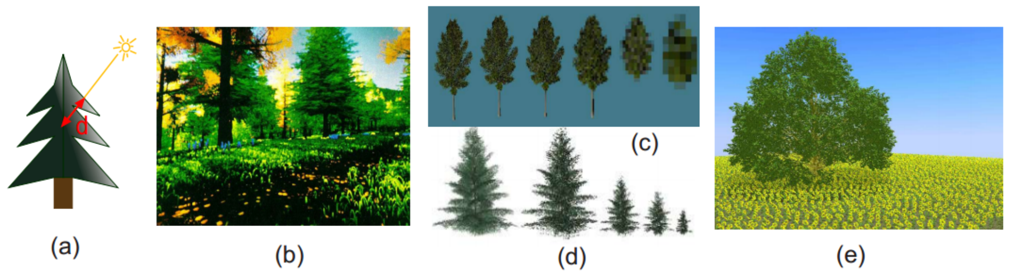


Figure 2.35: Point-based representations of trees: (a)(b) stochastic shadowing model for particle systems proposed by Reeves and Blau [46], (c) results by Weber and Penn [53], (d)(e) results from Deussen et al. [13]. Image from [4].

Finally, there are also proposed solutions that replace the indistinguishable distant data by a simple primitive combined with an illumination model that reproduces the same photometric behavior.

Based on spatial proximity and visibility

This set of techniques use arbitrary rules disconnected from the modeling process, allowing the construction of hierarchical structures from unstructured inputs like polygon soups. The hierarchy is usually built into an spatial structure such as octrees or BSP trees. In some approaches, rendering is performed by raycasting or slicing a volumetric texture. In others, the hierarchy stores points or polygons to draw.

Decaudin et al. [12] propose a volumetric texture rendering technique using a sliced triangular prism shape, which they call texcell. They use two different kinds of texcells: a simple one in which the slices are parallel to the ground, which serves for most views except grazing angles, and a more complex one sliced parallel to the screen, used near the landscape silhouette.

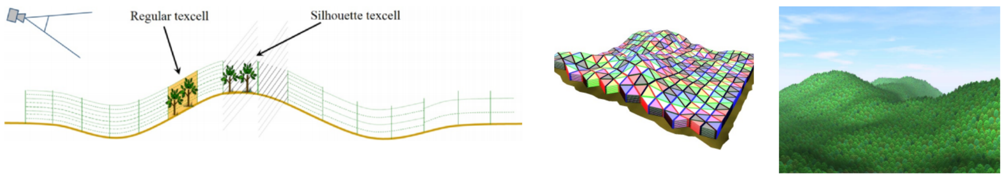


Figure 2.36: Decauding et al. rendering from slices. Left, slicing scheme with level of detail. Center, aperiodic tiling. Right, forest with 30000 trees rendered in real time. Images from [12].

Finally, other authors propose simplification methods based on visibility rules or based on distance by collapsing leaves to build a multiresolution model.

Chapter 3

Overview

We propose a semi-procedural technique for tree generation. In particular, we focused on the reconstruction of trees and shrubs models with a dense crown from a single photograph. A dense crown is that in which the foliage does not allow us to see the branches nor through the tree. The main goal is to obtain a model close enough to the pictured tree and which allows the visualization of thousands of instances efficiently.

In contrast with the approaches presented in 2.2.1, which reconstruct the underlying branching structure and then add the leaves, we will only approximate the shape of the crown. This is sufficient for representing the trees at medium and far distances. For close views, we will switch to a representation using textured leaves. Since we assume the crown is dense enough, we do not need to construct a branching structure. However, should we want to build it, we could use the Space Colonization algorithm of Runions et al. [48] and set our reconstructed crown volume as input.

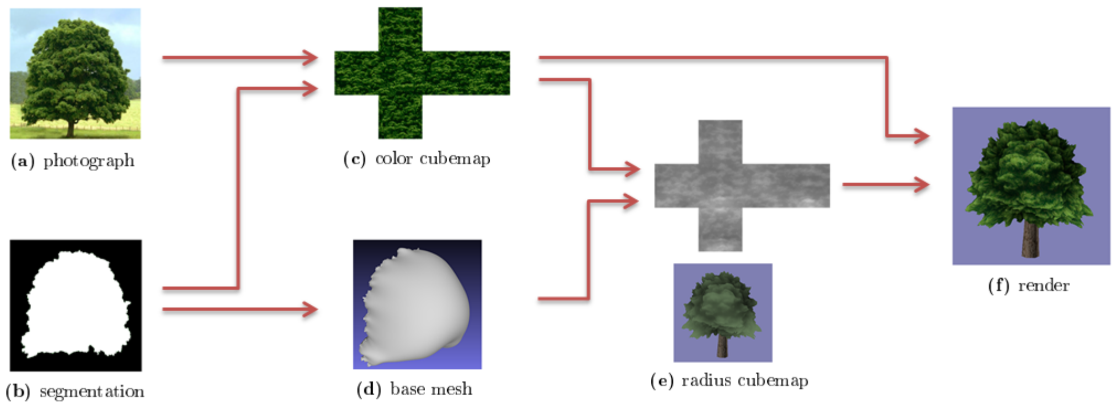


Figure 3.1: Overview of the crown reconstruction process

Our method starts with a picture of a tree or shrub crown, together with a segmentation of the crown foliage area. From this picture, we obtain two outputs: a volume mesh representing the general shape of the crown (base mesh), and a color cubemap representing the appearance of the foliage. This cubemap is used to compute an approximation of the crown relief, and the base mesh is perturbed accordingly.

The reconstructed crown mesh is rendered into a depth cubemap storing the maximum distance from its center to the crown. In other words, for each direction we store the radius from the center of the crown to the silhouette. This sort of radial representation allows us to efficiently render trees using a shader inspired by relief mapping.

Chapter 4 explains in detail how we reconstruct the crown from the input photograph, as well as a previous attempt method we finally discarded. The rendering algorithm is described in chapter 5. The integration of our trees in a real terrain navigation is shown in chapter 6. Finally, chapter 7 provides more tests and results.

Chapter 4

Crown reconstruction

This chapter describes the method we use to obtain a crown model from a single photograph. First, we will specify the required input and how it is treated. Then, we will explain an early approach we implemented and to which we refer to as *radial silhouette approach*. Then, the actual used approach based on *inflation* is presented. Finally, we will show how we obtain the fine relief details from the picture.

4.1 Input

Our method starts from a single picture of a tree or shrub and a segmentation mask for its crown foliage area. Early reconstruction methods presented in section 2.2.1 also asked the user to provide a segmentation of the tree. Newer methods integrate automatic or used-aided matting algorithms. Analyzing the various matting methods and implementing one of them was out of the scope of this project, but in the future we would like to research them and thus provide a fully automatic method that only needs the input photograph.

4.1.1 Photograph requirements

Currently, we need to ensure the following conditions about the input photograph in order to produce correct results:

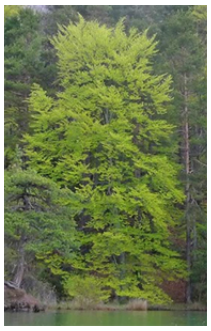
- The crown silhouette must be completely visible. This condition could be relaxed in the case of user-provided segmentations if the user specifies the shape of the crown as well as the occluded areas (otherwise, texture from the occluders would be used).
- The foliage should be dense and uniform, the underlying branching structure should be hidden or hardly visible.
- The shading seen in the crown must be caused by self-occlusions in lighting conditions similar to ambient light. Our method will produce an inadequate relief if there is some gradient due to the direction of the lighting.
- The crown can be approximated as a single volume. For now, we consider only simple crowns. In the future we might extend the method to deal with trees with sparse foliage nuclei.

Although these conditions may seem too restrictive, we will see now that in fact they are not difficult to satisfy.

Figure 4.1 shows some of the most typical trees found in Catalan forests[1]. These species, when they are in the wild, usually grow dense crowns with a volumetric appearance. Some exceptions are *fagus sylvatica*, which grows a dense crown but its branches are very separated and with a sparse foliage, and many individuals of *pinus sylvestris* and *pinus halepensis*, whose crown is usually made up of smaller dense crowns for the major branches, thus revealing the underlying branching structure.

(a) Alzina (*quercus ilex*)

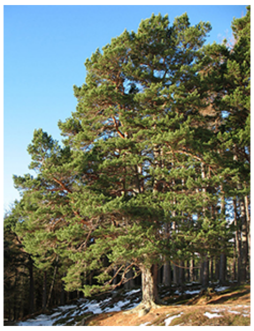
(b) Roure (*quercus humilis*)


(c) Faig (*fagus sylvatica*)

(d) Castanyer (*castanea sativa*)

(e) Pi blanc (*pinus halepensis*)

(f) Pi pinyer (*pinus pinea*)


(g) Pi roig (*pinus sylvestris*)

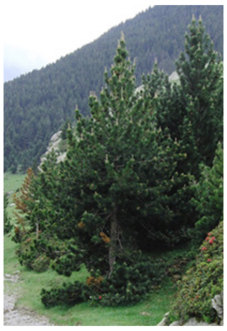
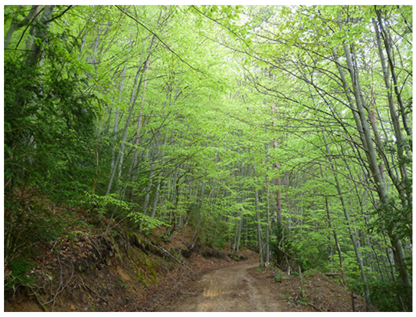

(h) Pi negre (*pinus uncinata*)

Figure 4.1: Common tree species found in Catalunya. Images (a)(c)(d)(g) from Wikimedia Commons. Images (f)(h) from www.floracatalana.net.

Even for the above mentioned bad cases, their global appearance as seen from afar may hide the branches. Therefore, even for the cases in which our method is not applicable, we can still obtain a plausible reconstruction to be used in far views. A clear example is shown in figure 4.2.

[1]See http://www.creaf.uab.es/iefc/pub/Catalunya/MenuEspecies.htm for a detailed list and descriptions, and http://www.creaf.uab.es/iefc/pub/Introduccio/Especies for coverage maps.

Finally, the images in figure 4.3 show different forest landscapes at medium to far views. Notice the density and regular shape of the crowns. Most of the species in picture 4.1 usually show a dense crown at medium distances (10m and beyond).


(a) Inside a *fagus sylvatica* forest, although the forest is dense, individual trees show many branches with a sparse crown.

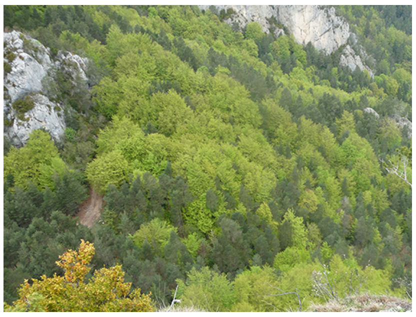
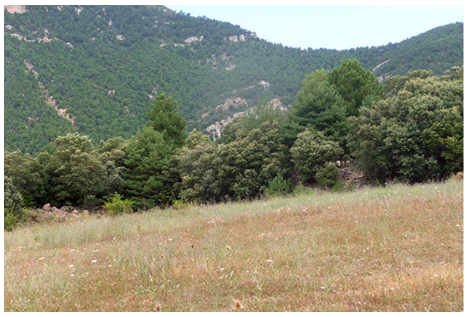

(b) The same forest seen from above.

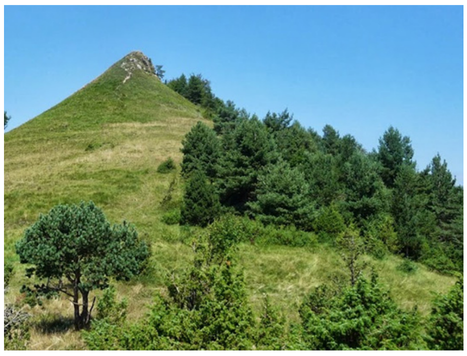
Figure 4.2: *Fagus sylvatica* forest seen at different scales.


(a) Serra de Turp, Alt Urgell

(b) Serra del Verd, Berguedà/Solsonès

(c) Cingles de Vilanova, Osona

(d) Serralada litoral, Maresme


(e) Puig de les Bruixes, Garrotxa

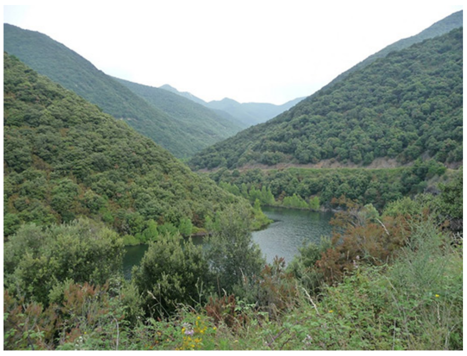

(f) Montseny, Vallès Oriental

Figure 4.3: Forest scenarios pictured in Catalunya like the landscapes we aim to reconstruct.

4.2 Radial silhouette approach

This is the first, naive method we came up with to reconstruct the crown shape. The basic idea is to generate a crown volume from a silhouette, like in one of the steps of the sketch-based approach proposed by Okabe et al. [33].

4.2.1 Radial silhouette

Given the crown segmentation, the first thing we do is to extract the *radial silhouette* of the crown. We call radial silhouette the approximation of the input silhouette defined by a function $r(\varphi) = \text{dist}(s - c)$, where φ is an angle, s a point on the crown silhouette (which may not be unique) and c the centroid, a point inside the crown segmentation area.

The result is the silhouette of a star-shaped polygon in which every point inside it is visible from c. However, this polygon may include regions outside the original segmented crown and exclude regions inside it, depending on the chosen position of c. At first, we set c to be the centroid of the segmented crown area, computed as the sum of valid pixel positions divided by the number of them. From now on, we may refer to the polygon obtained from the radial silhouette as *radial crown*.

To construct the function r, we discretize the set of angles and find the intersection of the ray shoot from c and the input crown silhouette. Therefore, we have to set a rule for the cases in which the ray intersects the silhouette multiple times, i.e. whether we want to overestimate or underestimate the input crown. We decided to keep the last intersection, since the resulting crown looks more faithful to the original shape. However, for approximately star-shaped crowns the result is almost perfect.

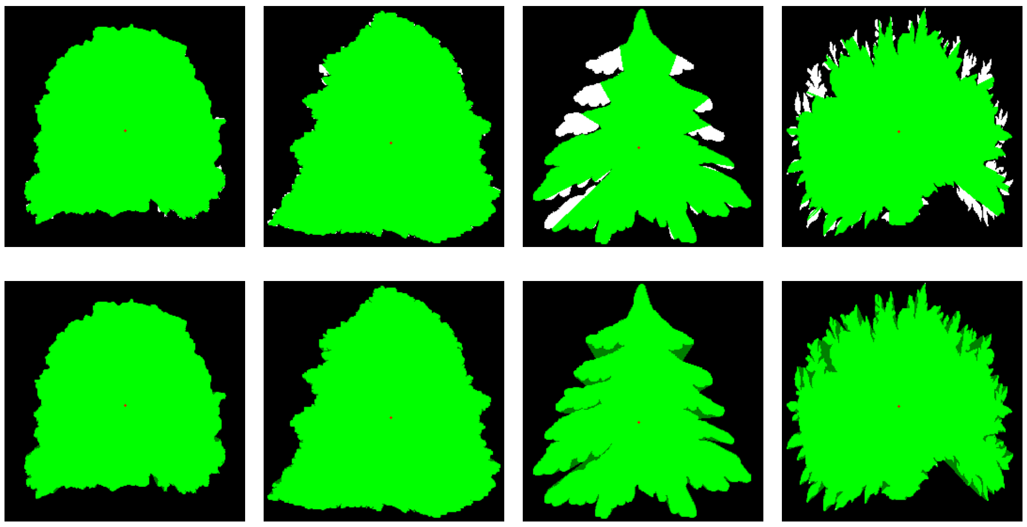


Figure 4.4: Radial silhouettes for first (top row) and last (bottom row) intersection. White pixels represent the original crown segmentation, onto which green pixels showing the obtained polygon or radial crown are overlaid. The centroid is shown in red.

The next thing we can do is to check if there is a point c for which the radial representation is more faithful than using the centroid. Although efficient algorithms exists to test whether a given polygon (in this case the input segmentation of the crown) is star-shaped and what its kernel is, most of our crown examples do not have a kernel. Running a brute-force search for the optimal c, i.e. the point from which its radial crown area outside the original segmentation is minimum, would take a huge amount of time. Therefore, we implemented a gradient descent-like algorithm starting from the centroid and moving to the best of its neighbors. When we ran the tests for this version, we observed the distance between the segmented crown centroid and the optimum found by gradient descent was very small.

Then, we did a brute-force test computing the number of pixels that would be added for each position inside the segmented crown if we took that position as center of the radial silhouette. Results showed that the domain has a large number of local minima, so gradient descent approaches are going to fall into one of them. But another observation was that the relative difference in amount of added pixels between centroid and optimum is really small (see table 4.1). Moreover, the final shape of the radial silhouette was very similar.

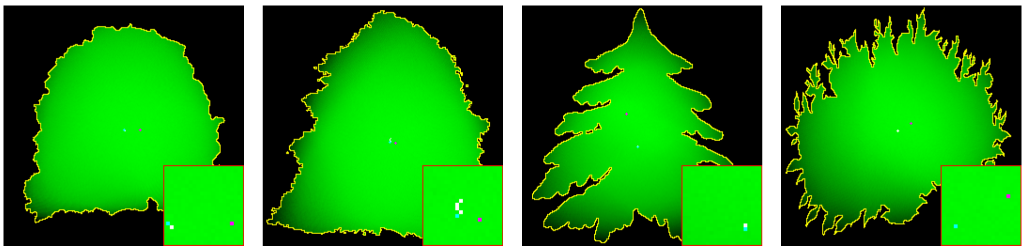


Figure 4.5: Optimal center for different crowns. The green color represents the optimality of choosing each point as center for the radial silhouette, the brighter the better. Magenta point is the global optimum, cyan point marks the centroid, and white points represent the path followed by gradient descend.

segmented crown pixels	c = centroid	c = local optimum	c = global optimum
33753	+373	+361	+341
39438	+580	+560	+546
29695	+1742	+1717	+1574
36836	+2702	+2702	+2623

Table 4.1: Amount of added pixels for each of the test crowns shown in figure 4.5, in order. This test was run on 256×256 images.

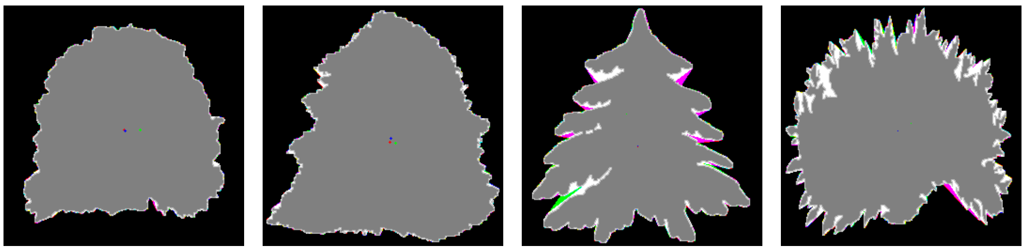


Figure 4.6: Radial crowns obtained from different centers. Three different radial crowns have been overlaid to the original segmentation drawn in gray: the one using centroid in red, using local optimum from gradient descend in blue, using the global optimum in green. Notice that most of the shape is white/gray, meaning all three overlays intersect. Other colors represent either a single crown or the intersection of two of them.

As a consequence, we decided to keep the centroid as the center of our radial silhouette, since it is the easiest and fastest solution and the obtained radial crown is good enough compared to the best we could do. Now, all we have to do is extrapolate a 3D function from r to define the 3D crown.

4.2.2 From 2D silhouettes to 3D volumes

The radial silhouette is defined using a function $r(\varphi)$. What we really want is to represent a radius in any 3D direction, so we need to extend this function with another angle $R(\varphi, \phi)$ and then use this function to build a cubemap of the crown radius.

Given a direction $\mathbf{d} = (x, y, z)$ and using the world up direction $(0, 0, 1)$, we project $\mathbf{d}$ onto the XZ plane and compute the angle φ, and ϕ is computed from the projection onto the XY plane. Then, the two radii $r_1 = r(\varphi)$ and $r_2 = r(-\varphi)$ are linearly interpolated using $\phi \in [-\pi, \pi]$:

$$R(\varphi, \phi) = \frac{|\phi|}{\pi} r_1 + (1 - \frac{|\phi|}{\pi}) r_2 = \frac{|\phi|}{\pi} r(\varphi) + (1 - \frac{|\phi|}{\pi}) r(-\varphi)$$

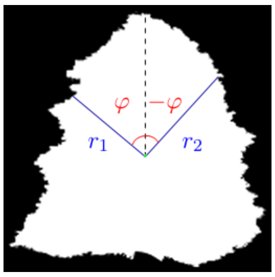


Figure 4.7: The two radii being interpolated to extrapolate the 3D silhouette.

The figure below shows the 3D crown obtained with this method. The most noticeable artifact at first sight is the circular bands caused by the interpolation using the angle ϕ of the direction vector projected onto the horizontal plane. This bands, however, are removed when we add fractal noise or a relief (see section 4.4) to the reconstructed volume.

Then, if we take a second look and compare the resulting volume with its 2D radial crown, we will realize it is much more convex than the radial silhouette computed for it. In fact, if we project the 3D crown onto the XZ plane, the fourth volume is very different from its radial crown. The reason is our extrapolation scheme is covering the concavities of the radial crown when it "revolves" the silhouette along ϕ.

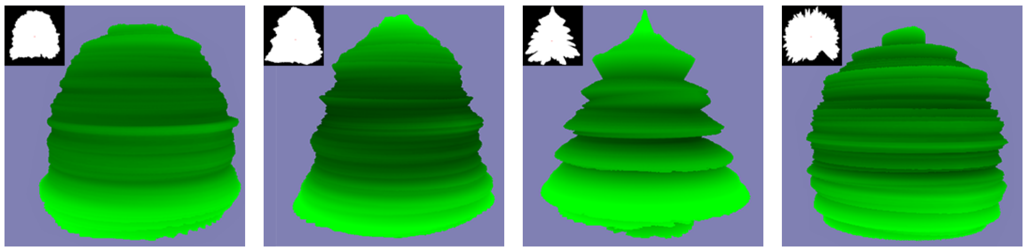


Figure 4.8: 3D crowns reconstructed from their radial silhouettes (shown in the upper-left corner). Color intensity indicates the magnitude of the radius at each point.

The approach presented in this section is very simple but it has some limitations, specially for very concave shapes. Therefore, we propose an alternative more complex method to reconstruct a crown from its silhouette in the next section that gives better results (see figure 4.15 on page 47, which shows the same crowns except the first one). The difference between both methods is specially noticeable in the last example crown.

4.3 Inflation approach

The artifacts in the previous approach were partly produced by using an overly simplified representation for the crown silhouette. In fact, even before extrapolating the 3D model we were already introducing distortions to the silhouette with our radial function. Thus, we want a new method that fully leverages the silhouette as segmented by the user.

Although the input to our algorithm is a photograph, for now all the information we needed was the segmentation, more precisely the segmentation silhouette. Therefore, we looked at the sketch-based approaches of tree reconstruction like [33, 57] and both of them mention that users tend to draw branches extending lateral to the trunk. Moreover, Tan et al. in [51] also cite this observation from [33] when they extract the crown foliage area shape. Therefore, we are going to make the same assumption: the silhouette lies on a plane (for example, $z = 0$) and the inner pixels of the segmented crown extend toward the viewer (z $>$ 0). All we need to do is to *inflate* these inner pixels, like if we were blowing them from behind but keeping the silhouette fixed.

Our first idea was to use the distance field $d(i, j)$, defined for each inner pixel (i, j) as distance to the silhouette, as altitude for the inner pixels. This produces ridges along the maximal values of the distance field that are visible even if we try to make the distance-altitude relationship more spherical, for example by applying a sinus function.

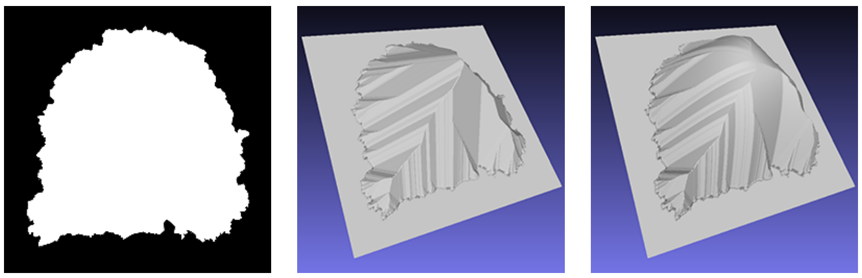


Figure 4.9: 3D crowns reconstructed from the distance field. Left, segmented crown. Center, $z(i, j) = d(i, j)$. Right, $z(i, j) = D \sin(\frac{\pi}{2} \frac{d(i,j)}{D})$, where D is the maximum distance in the field.

What we actually want is to minimize the thin-plate energy defined by a biharmonic equation inside the crown C, and restrict the silhouette S to have $z = 0$:

$$\min_z \int_C \nabla^4 z, \qquad \text{such that } z \geq 0 \text{ and } z(p) = 0 \text{ for } p \in S \tag{4.1}$$

Obviously, the equation above has a trivial solution $z(p) = 0$ for $p \in C$. We still need to constraint the tangent direction for points on the silhouette. Since we want the crown to be inflated upwards in the z direction, we will set tangents of the form $t = (0, 0, k)$ for those points on the silhouette.

4.3.1 Iterative bilaplacian

In a first implementation, we solved the biharmonic equation using an iterative bilaplacian method. We started with the planar mesh obtained by creating a vertex v at each inner pixel in the crown segmentation with $z = 0$ and replicating the silhouette vertices with a negative displacement of k units in the z direction. Then, $2N$ iterations of the laplacian smoothing are applied alternating the sign of λ:

$$L(v^{(i)}) = \frac{1}{|\text{Neigh}(v^{(i)})|} \sum_{n^{(i)} \in \text{Neigh}(v^{(i)})} n^{(i)} - v^{(i)}$$

$$v^{(i+1)} = v^{(i)} + \lambda L(v^{(i)})$$

This method is not only very slow but also reaches and gets stuck into a local optimum of the thin plate energy function that is not what we are looking for:

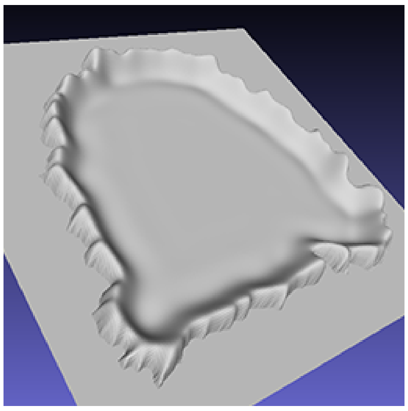


Figure 4.10: 3D crowns reconstructed using the iterative bilaplacian with $2N = 2000$ iterations, $\lambda = 0.5$ and $k = 5$. Increasing k only increments the height of the extruded silhouette shown in the picture. We tried increasing the iterations up to millions (which took several hours to complete) but the results were still very similar.

4.3.2 Linear system

Then, we switched to a more direct solver for the biharmonic equation, this time based on a linear system for z. We need to discretize the domain C, so we use our input segmentation as a 2D grid discretization. But now we need a different strategy to specify the tangents, since each cell (pixel) has a unique height z. We define an outer silhouette S_o as the set of pixels neighboring a silhouette pixel, and apply the negative displacement in z to it. For each inner pixel (i, j):

$$\begin{array}{ll} z(i,j) = x & \text{if } (i,j) \in C \backslash S \\ z(i,j) = 0 & \text{if } (i,j) \in S \\ z(i,j) = -k & \text{if } (i,j) \in S_o \end{array}$$

where x represents an unknown of the linear system, and k is a user-defined value.

Then, for each unknown x we create the equation obtained from the convolution of this point with its neighbors, representing the thin-plate energy computation:

$$\sum_{\substack{-2 \le d_i \le 2 \\ -2 \le d_j \le 2}} M_{d_i+2, d_j+2} \cdot z(i + d_i, j + d_j) = 0 \tag{4.2}$$

$$M = \frac{1}{16} \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 2 & -8 & 2 & 0 \\ 1 & -8 & 16 & -8 & 1 \\ 0 & 2 & -8 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

We solve this system using the `SimplicialLDLT` solver included in the `Eigen` library [19], which implements an $LDLT$ Cholesky factorization. This is the result obtained with this system:

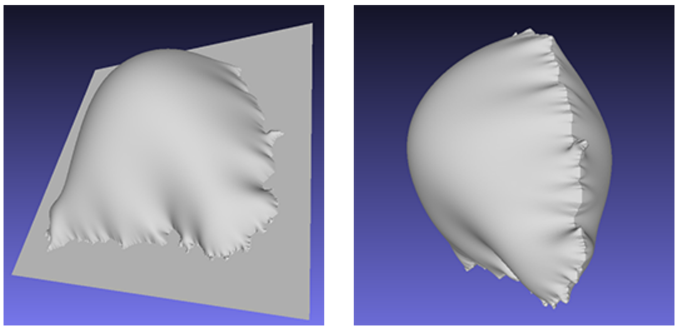


Figure 4.11: 3D crowns from the bilaplacian system and $k = 3$. Left, the direct heightfield obtained as a solution to the system. Right, mesh after cropping the crown area and merging it with its mirror mesh.

This result looks promising, but there are still some issues. Due to the way we are now defining the tangents using the outer silhouette on the grid, tangents can not be defined as we wanted: as vectors pointing in the z direction, because there is always a displacement in x and y between the outer silhouette and the real silhouette. Therefore, the crown does not look as spherical as we expected. Increasing the value of k increases the protuberance height, but tangents never match with the mirrored ones at the silhouette and the overall shape looks like a large ellipsoid. Another possible cause may be that each inner point has its x and y fixed, so we may be limiting the degrees of freedom needed to produce almost vertical tangents at the silhouette.

This is why we decided to extend the previous linear system to 3D with unknowns for x and y as well. Now, inner vertices will no longer be restricted to be on a grid arrangement:

$$\begin{array}{ll} z(i,j) = (x_x, x_y, x_z) & \text{if } (i,j) \in C \backslash S \\ z(i,j) = (i, j, 0) & \text{if } (i,j) \in S \\ z(i,j) = (i - 0.5d_i, j - 0.5d_j, -k) & \text{if } (i,j) \in S_o \end{array}$$

where x_x, x_y and x_z are the unknowns, and we use equation 4.2 independently on each of them.

The values d_i and d_j that appear on the position of pixels in S_o are the same as the ones in equation 4.2. Note that for each inner pixel we define a convolution that has a radius of 2 pixels, therefore some of them will convolve with the outer silhouette pixels and the resulting value will be added to the independent term. What we are doing is setting the position of these outer pixels halfway through their actual position and the position of the inner pixel that is consulting them, thus simulating they are located on the real silhouette. We saw empirically that this trick improves a lot the tangents obtained from the solver at the silhouette. See figure 4.12.

Finally, we found the connectivity of the silhouette pixels plays a big role in the reconstruction process. Figure 4.13 shows that we were not able to reconstruct a sphere from a circle image using 4-connected silhouettes. For small values of k we obtained very flat shapes, but increasing these values produced excessively elongated volumes and made a star-shaped pattern appear. Using 8-connectivity in the silhouette pixels, however, solved all these artifacts.

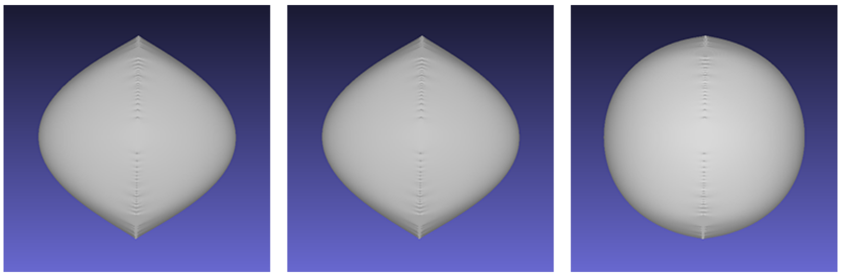


Figure 4.12: Difference between the 1D and 3D system reconstructions, using a circle as input image. Left, the result of the 1D system. Center, the result of the 3D system without displacing the pixels in S_o. This shows that our hypothesis that using more dimensions would allow vertices to move was wrong. Right, the result of the 3D system with the displacement trick for pixels in S_o, now the tangents look good and the reconstructed mesh is spherical.

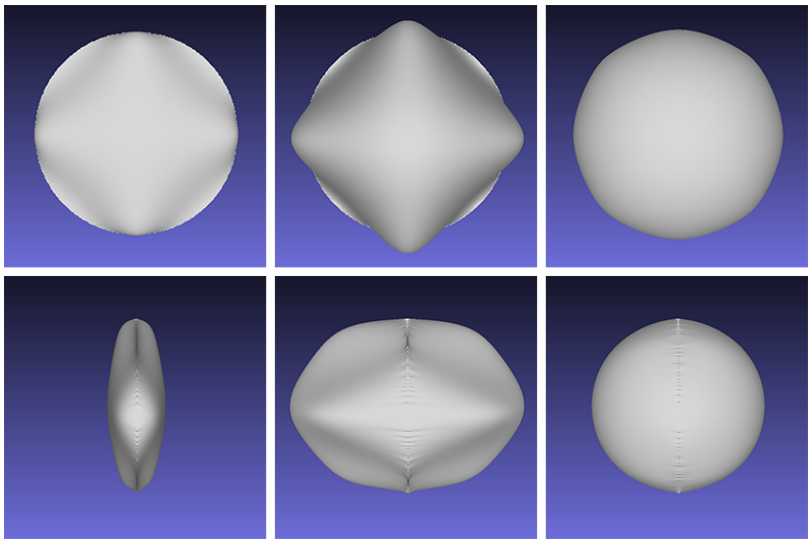
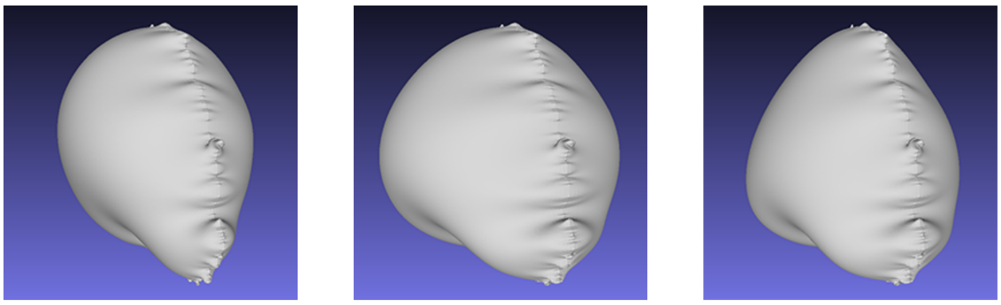


Figure 4.13: Effect of silhouette connectivity in the 3D bilaplacian system. From left to right: 4-connected and $k = 2$, 4-connected and $k = 8$, 8-connected and $k = 2$.

One of the advantages of using the inflation method is that we can play with the tangents at the silhouette and have a better control over the final reconstructed shape. In our current implementation, this is done by modifying the value of k, either by setting a constant value or defining a hard-coded function $k(i, j)$. An interesting feature for a final software would be allowing the user to define these functions.



(a) $k(i, j) = 3$ **(b)** $k(i, j) = 3 * (1 + j^2)$ **(c)** $k(i, j) = 3 * (0.5 + \frac{4^{2j}}{8})$

Figure 4.14: Effect of modifying the silhouette tangents. Origin is located in upper-left corner, and $i, j \in [0, 1]$.

These are the rest of the test crowns reconstructed using this method, shown below. When the mesh is viewed laterally it is clearly evident that it is composed of two mirrored parts. However, this symmetry is removed once we add the relief (see next section).

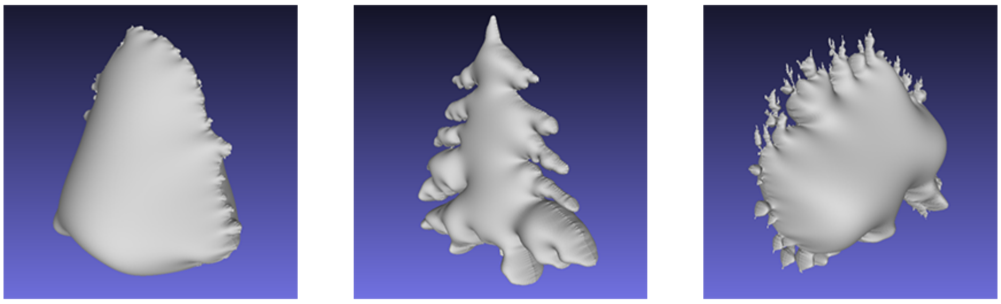


Figure 4.15: Crowns reconstructed using the inflation method.

Also, as one can see, silhouettes with lots of concavities produce a final mesh that looks flat, like a helium balloon. We still need to improve our method for these cases. Right now, what we do for these silhouettes is similar to how Tan et al. [51] extrapolated their 3D attractors: we create rotated copies of the base mesh and merge them to cover more viewing angles while respecting the similarity to the input silhouette. For example, the following image shows how we managed to reconstruct a plausible model from the center mesh of the figure above.

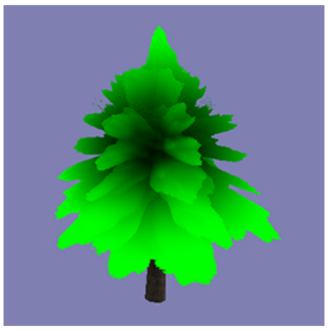


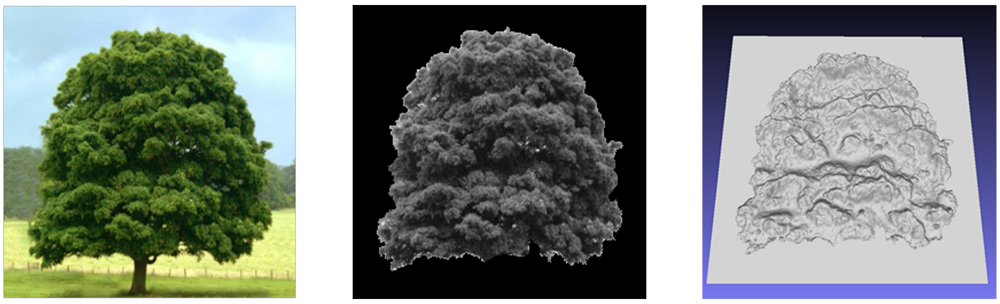
Figure 4.16: Tree created from three rotated copies of the base mesh and relief perturbation.

4.4 Extracting relief

When we look at a picture of a tree, we are not only able to distinguish its silhouette but also we perceive its volume and relief. Even with a single picture of a tree we have never seen before, we are capable of mentally modeling the visible part of it with relief. The perceived illumination gives us visual cues of the relief and occlusions between the elements in the picture, so maybe we could use it to obtain the crown relief.

Shape-from-shading approaches are an under-constrained problem, lots of different shapes, materials and lights can produce a given shaded picture. However, taking some assumptions, there are previous works that offer solutions to the problem. For example, Glencross et al. [16] proposed a technique they call *Depth Hallucination method*. It is based on the premise that under diffuse lighting, surface luminance depends primarily on a local aperture function defined as the solid angle subtended by the visible sky at each surface point. In other words, if there is no light creating shadows and shading gradients on the picture, the luminance can be seen as ambient occlusion. They estimate depth by modeling the pits as cylinders and the protrusions as hemispheres, and then they deterministically calculate the height of the cylinder/hemisphere that is producing the corresponding shading at each pixel.

Our relief estimation method was inspired by this work, but uses a more simple approach. We directly use the luminance as the estimator for depth. We consider that a tree has a huge number of self-occlusions from the different branches and leaves, but these are not completely opaque and a fraction of incoming light still reaches occluded portions. The deeper we are inside the crown, the darker we expect it to be. We are using the fact that the input crown is required to be dense, otherwise deep areas close to the center of the crown could be receiving direct light and our model would fail.



(a) Input image **(b)** Luminance **(c)** Relief mesh obtained

Figure 4.17: Crown relief extracted from the luminance.

The image above shows that a plausible reconstruction is obtained using our hypotheses and method. Using the luminance directly, the result is very noisy. Instead, we developed a multiresolution approach. A more in-depth description of this procedure follows.

We start by computing the luminance of the input image. Since the input is RGB, we obtain luminance as $L = 0.2126R + 0.7152G + 0.0722B$ and normalize the resulting image to be in the range $[0, 1]$. Then, we compute a blur pyramid of this luminance image, similar to the *Gaussian pyramid* of Burt and Adelson [8]. The difference is that instead of reducing the image at each level, we increase the Gaussian blur radius. Since the 2D Gaussian blur is separable, using large kernel radii does not impact the method performance. If we instead reduce the image, we will need to upsample it to combine all the levels, and upsampling artifacts appear in the final relief.

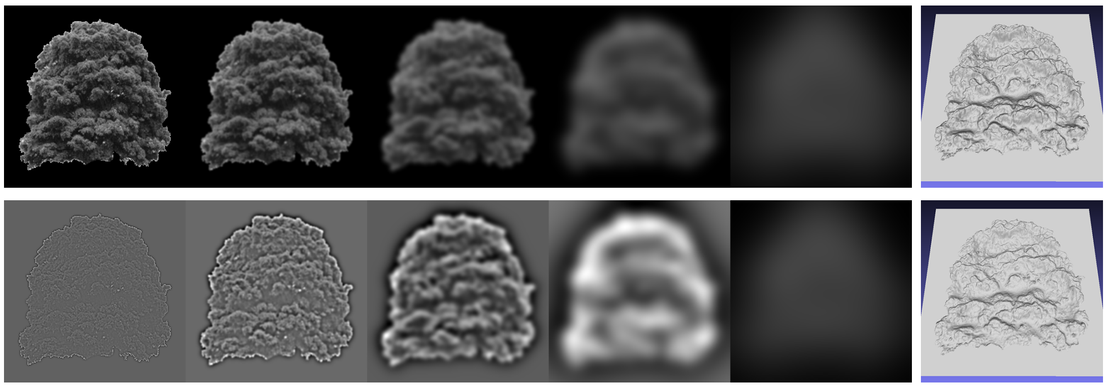


Figure 4.18: Result using Gaussian (top) and Laplacian (bottom) pyramids. The result using Gaussian pyramid is more prominent because lower frequencies are counted multiple times.

Optionally, once we have the set of blurred images, we can subtract each level with the next (except the last one) to obtain a *Laplacian pyramid*, in which each level can be seen as the result of a band-pass filter instead of a low-pass filter. In practice, both filtering methods produced almost identical reliefs.

Once we have N levels of the pyramid (we used $N = 4$ and $N = 5$), we combine them using a weighted sum of the luminances at each level. Lower levels (more blurred) will have more impact because they represent larger structures, while higher levels are usually more prone to noise and just add small fine details to the relief.

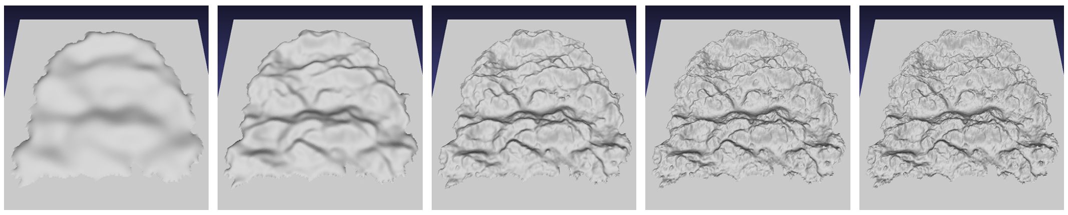


Figure 4.19: Multiscale relief generation: each heightfield is the result of adding its corresponding Gaussian pyramid level to the previous one, and reducing the weight at each step.

Finally, we are interested in forcing the silhouette to lie on the plane, i.e. to have a value of 0 in this heightfield. To achieve this, we followed the ideas in the *Poisson Image Editing* method of Pérez et al. [37]: we "paste" the obtained heightfield to a plane with height 0 and force the values to coincide at the silhouette of the crown. This is done in gradient domain by solving a Poisson equation with Dirichlet boundary conditions whose corresponding discrete version solution satisfies the following linear system:

$$\text{for all } p \in \Omega, \qquad |N_p| f_p - \sum_{q \in N_p \cap \Omega} f_q = \sum_{q \in N_p \cap \delta\Omega} f_q^* + \sum_{q \in N_p} v_{pq}$$

where f_p is the source value of p, f_p^* the destination value of p, v_{pq} the gradient in $\vec{pq}$ direction, N_p the set of neighbors of p, Ω the domain in which the source is defined, and $\delta\Omega$ the boundary of it. If we apply this to our particular case, the following system results:

$$\text{for all } p = (i,j) \in C, \qquad 4x_p - \sum_{q \in N_p \cap C} x_q = 4H_{i,j} - H_{i+1,j} - H_{i-1,j} - H_{i,j+1} - H_{i,j-1}$$

where C represents the set of pixels segmented as crown, x_p the unknown for p, and $H_{i,j}$ means the heightfield value at (i,j). Note that our destination image is a flat plane, so $f_p^* = 0$ for all p, and the neighborhood of p is defined by its 4-connected neighboring pixels.

This heightfield we have obtained will be used to perturb the base crown mesh we created before. But first, we need to deal with the non visible rear part of the crown.

4.5 Texture generation

A single picture of a tree contains a representation of about less than half the tree surface. However, it is logical to assume that the parts which are not visible will have a similar appearance to those we see. What we need to do is to generate these remaining parts, synthesize them to look like the provided image.

This problem is called *example-based texture synthesis* and has been well studied by many authors; a survey is available in [54]. We implemented the texture synthesis method of Wei and Levoy [55].

Their technique initializes the output texture with random pixels taken from the input image. Then, pixels in the output image are replaced in raster scan ordering. To determine the color of a pixel p, its local neighborhood $N(p)$ is compared against all neighborhoods $N(p_i)$ of input image pixels p_i. The value of p_i corresponding to the most similar neighborhood will be assigned to p. Causal neighborhoods ensure that only already synthesized and valid pixels - except for the first rows which use the random values - determine the value of the current pixel. Borders are treated toroidally.

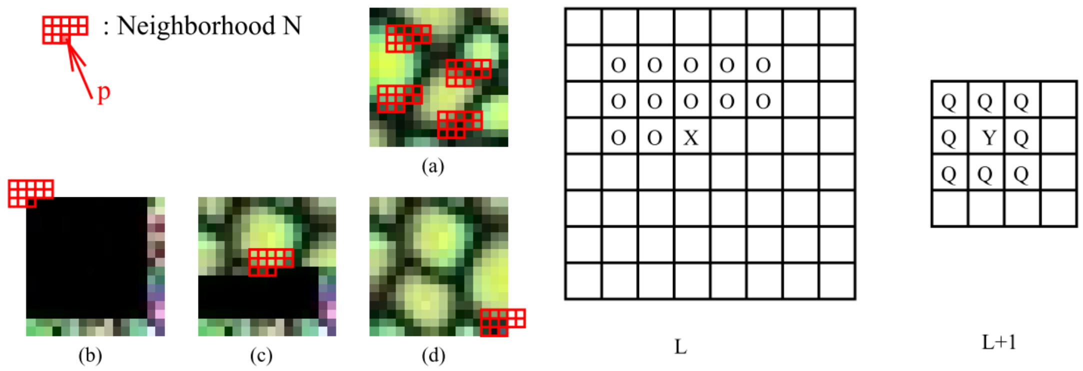
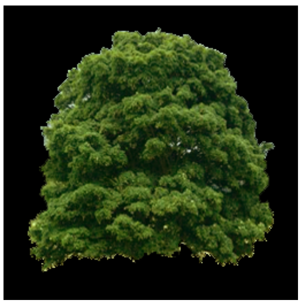


Figure 4.20: Wei and Levoy texture synthesis algorithm. Left, the single resolution texture synthesis. The neighborhood $N(p)$ is compared against all neighborhoods in the input image (a), and the output image is built in scan order (b)(c)(d). Right, the definition of a neighborhood for multiresolution synthesis. $N(x)$ includes all the pixels marked with O, Q, and Y.

Wei and Levoy's method allows for multiresolution texture synthesis. They build a Gaussian pyramid for both input and output image. The top (smaller) level is synthesized as before. And then each successive (bigger) level uses a neighborhood containing both the causal neighborhood in the current level and the corresponding non-causal neighborhood of the previous one. See figure 4.20.


(a) Input photograph

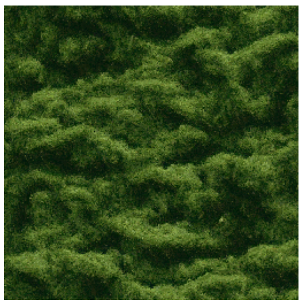

(b) Synthesized texture

Figure 4.21: Foliage synthesis using Wei and Levoy's algorithm. For this example, we used a 7×7 neighborhood and 3 pyramid levels.

The figure above shows the result we obtained from one of the test crowns foliage. Although texture synthesis allows us to produce arbitrarily large images, the output image must be about the same size as the input crown because we want to capture the same structures size. If we built a texture twice as big and mapped it to the tree, the apparent size of feature would be half the size of the features seen in the original picture.

After having implemented this method like Wei and Levoy described, we made some changes to improve the results. We were not satisfied with the produced outputs, still some of the structures seen in the original photograph were not being represented. For example, in the synthesis shown in 4.21 it failed to capture the rounded structures from the input image. Increasing the neighborhood size or the pyramid levels did not improve the output.

4.5.1 Improving the synthesis algorithm

Other approaches that reproduce more faithfully the original picture are the ones that directly take patches from it and stitch them, instead of synthesizing one pixel at a time. Two examples are the Image Quilting method of Efros and Freeman [15] and the Graphcut Textures by Kwatra et al. [23]. The problem with approaches like those is that if a very characteristic spot or salient feature is present in the input image, it is easy to find repetitions of it in the output.

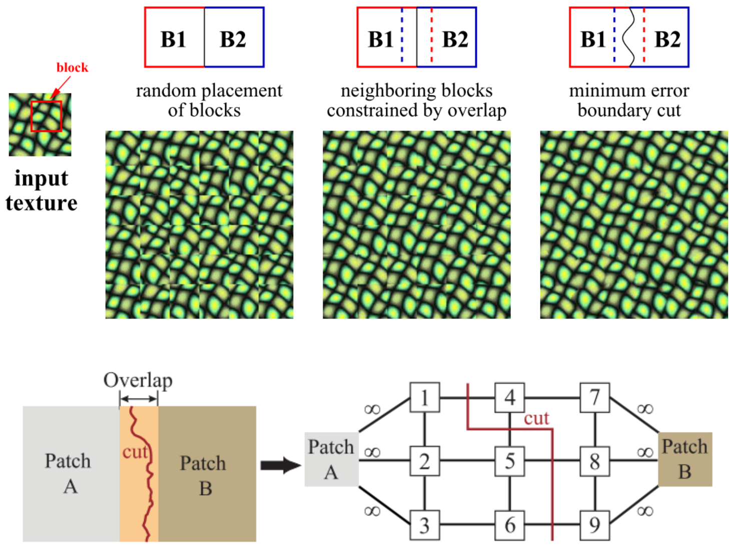

B1
B2
block
random placement of blocks
neighboring blocks constrained by overlap
minimum error boundary cut
input texture
Overlap
Patch A
cut
Patch B
∞
1
2
3
4
5
6
7
8
9
Patch A
Patch B

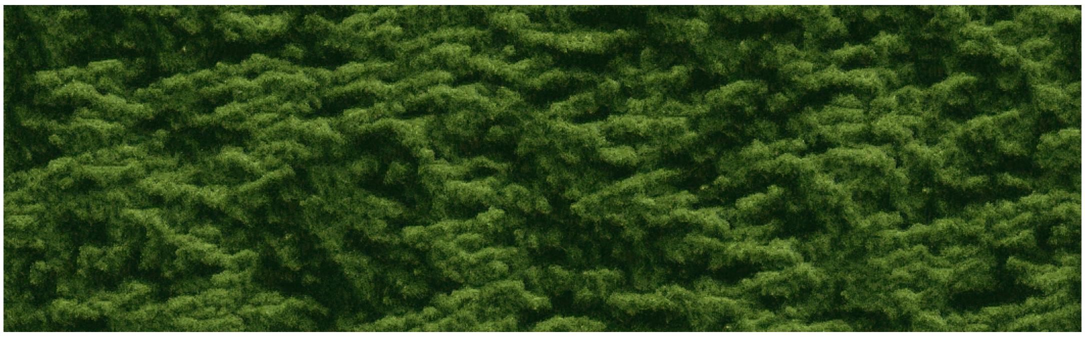
Figure 4.22: Patch-based synthesis algorithms. Top, texture quilting from [15]. Bottom, graph-cuts [23] find the best seam between blocks by finding a min-cut on a cost graph.

Therefore, we implemented an approach combining the best of pixel-based and patch-based synthesis. The first synthesized level of the Gaussian pyramid used by Wei and Levoy starts with noise. Instead, we create this level using square patches taken directly from the input, like in Image Quilting, and find the best seam between two overlapping blocks using a graphcut. We implemented the graphcut by Kwatra et al. instead of the dynamic programming cut that Efros and Freeman propose because it not only has more freedom to find the best seam but we will also need this method later (see section 4.5.2).

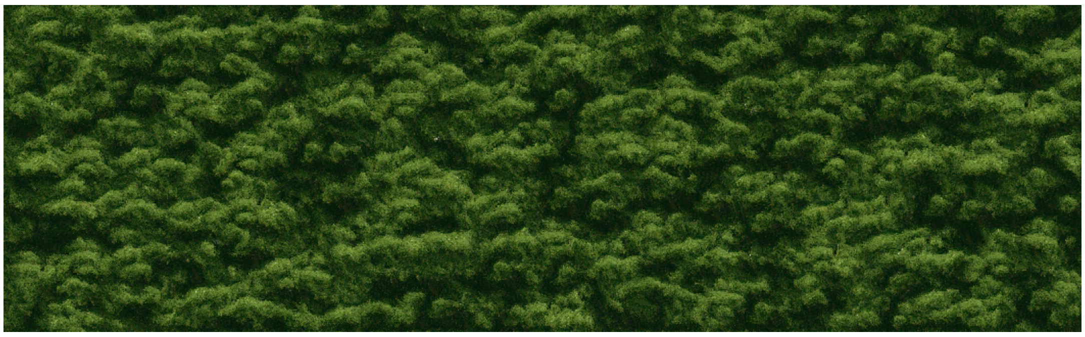
Once we have the first level synthesized, we proceed like before, using the pixel-based multiresolution algorithm of Wei and Levoy. This change in the initial conditions provides much better results (see figure 4.23).

4.5.2 Synthesizing a cubemap texture

So far we have implemented a texture synthesis algorithm that produces 2D textures. Since we want to represent the crown foliage in any direction, what we actually need is to synthesize a cubemap texture. This introduces new challenges such as how to define neighborhoods across faces of the cube.



(a) Synthesis using Wei and Levoy multiresolution algorithm



(b) Synthesis using our modified version with patch-based synthesis on the first pyramid level

Figure 4.23: Comparison between synthesis algorithms. Note that the new version captures better the rounded structures visible in figure 4.21a. Furthermore, the raster scan traversal is visible in the original version as diagonal patterns from top-left to bottom-right. This is not visible in the new algorithm.

Instead of trying to define an overly complicated neighborhood at corners and edges of the cube, we decided to reuse the tools we had already implemented: a 2D texture synthesizer and the graphcut algorithm to find the best seam between two overlapping images. Suppose we want a cubemap with N pixels on each side, our synthesis proceeds as follows:

1. Synthesize a $(4N + B) \times (N + 2B)$ image I_L. This image will become the lateral faces of the cube.
2. Synthesize two $N \times N$ images, corresponding to the top I_T and bottom I_B.
3. Find the best seam between the leftmost B pixels of I_L and its rightmost B pixels, obtaining a new image I'_L of size $4N \times (N + 2B)$ that is horizontally tileable. Therefore, the cubemap lateral is continuous.
4. Now, use the top B rows of I'_L to find the best seam with a border of size B in I_T, and produce a new image I'_T that is continuous with the lateral faces after discarding the top B rows of I'_L.
5. Do the same with the bottom B rows of I'_L to merge it with I_B.
6. Separate each lateral face from I'_L, and build the cubemap from the six textures produced.

B is the size of the bands we will use to find the best cut and join the textures. In our tests, we usually set it to $N/8$.

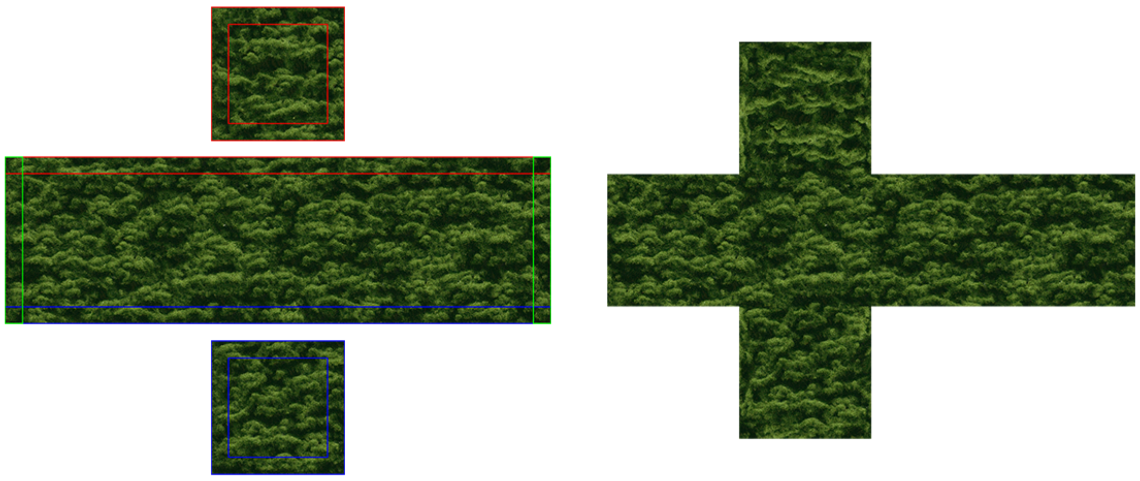


Figure 4.24: Cubemap texture synthesis by merging 2D textures. On the left, I_T, I_L and I_B, respectively, with the merging bands color-coded. The final cubemap is shown on the right.

4.6 Crown model

In the previous sections, we have seen all the tools we need to build the crown model. This is a summary of the crown reconstruction steps:

1. The *base mesh* is built from the silhouette using the inflation method (section 4.3).
2. We synthesize a *color cubemap* from the segmented crown foliage (section 4.5.2).
3. The *color cubemap* is used to create a *height cubemap*, using the relief estimation algorithm (section 4.4).
4. Combine the *crown base* with the *height cubemap*, and store the result as a radial representation from the crown center into a cubemap we call the *radius cubemap*.

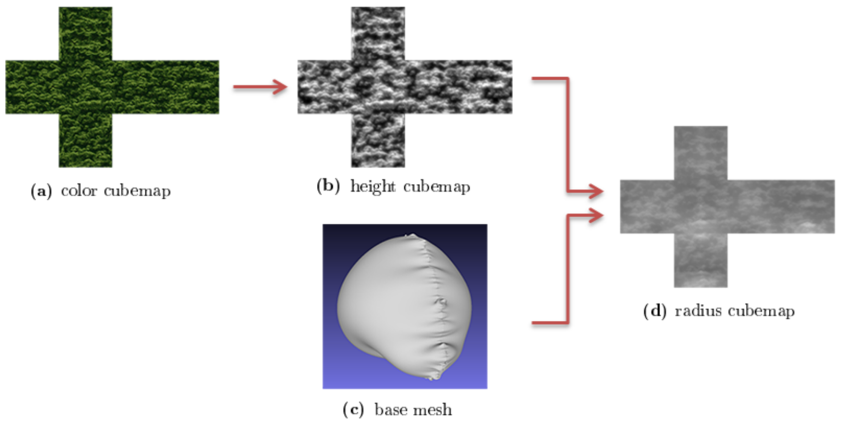


Figure 4.25: Summary of the crown reconstruction process

The only step we had not seen yet was the last one. To combine the base mesh with the relief cubemap, we just render the mesh and use the fragment shader to compute the radius as follows:

$$||p - c|| \cdot (1 + \rho \cdot (h(p - c) - 0.5))$$

where p is a point on the base mesh surface, c the center of the crown, $h(p - c)$ the relief cubemap value in the direction $p - c$ and ρ a parameter to control the weight of the relief map.

This result is rendered to the *radius cubemap*. Also, if we wanted to use various rotated copies of the base mesh (see figure 4.16), we do this during this stage by rendering it multiple times and keeping the maximum radius value in the output cubemap.

Chapter 5

Crown rendering

The previous chapter discussed how we build a crown model from a photograph. Recall that the result is two cubemaps, one for the texture (colors) of the crown and another one with a radial representation of it. We propose two different methods to render this model.

5.1 Relief-mapping crown approach

The first method implements a direct visualization of the radius cubemap, similar to a *relief mapping* algorithm [38]. Basically, we draw a volume that encloses the crown and raycast from the surface of this volume until we either exit and discard the fragment or intersect the crown and then we texture and shade it accordingly.

Like in relief mapping, we first do a linear search until we detect we are inside the volume, and then a binary search to refine the intersection point. The main difference is that we use world space positions and traverse a ray in the viewing direction from the bounding volume position of the fragment.

The number of linear steps, their length, and the number of binary steps are computed dynamically in the vertex shader and linearly depend on the radius of the bounding sphere projected on the screen.

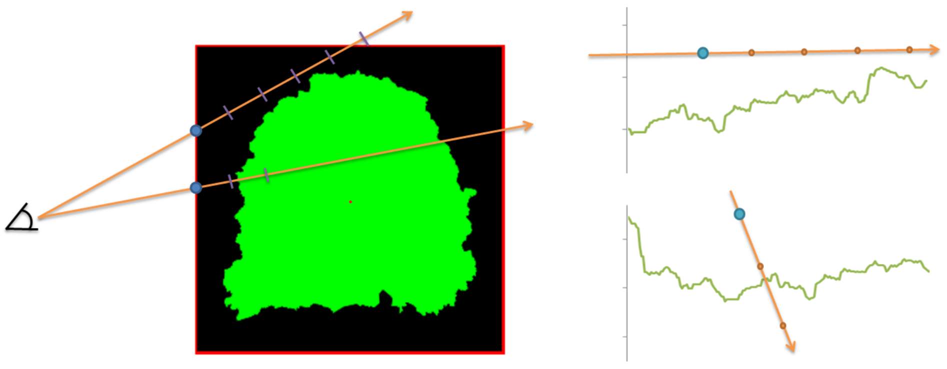
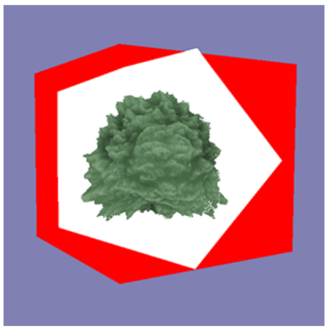
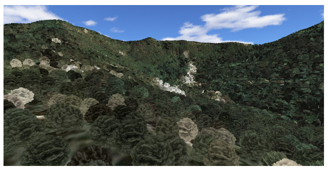


Figure 5.1: Relief crown rendered using raycasting. Starting on the surface of the enclosing volume, we do linear steps until we are outside the volume again or inside the crown. The plots on the right side show the crown distance to the center (green) and the ray distance samples.

The choice of the bounding volume is relevant: we want it to be as tight as possible to the crown shape so fewer fragments end up being discarded. As we are using a radial representation inside the cubemap, it is reasonable to assume that a sphere would give the best results on average (some trees are more elliptical than spherical). However, rendering a detailed sphere would be very costly, so we approximate it with a scaled icosahedron. In order to guarantee a proper rendering, the bounding volume must be big enough to inscribe a sphere with a radius equal to the maximum value of the radius cubemap.



(a) Bounding cube (red) and icosahedron (blue) drawn around the crown model.



volume	normal	optimized
cube	15.4 ms	12.8 ms
icosahedron	12.9 ms	12.5 ms

(b) Measured rendering times for a forest scene with about 22000 trees.

Figure 5.2: Comparison between bounding cube and icosahedron.

The table in figure 5.2b shows that drawing a tighter volume improves performance ("normal" column). Then, we did a further optimization to the fragment shader. Since we know the minimum and maximum value in a radius cubemap, R_{min} and R_{max}, if the distance between the crown center and the ray is bigger than R_{max}, we can directly discard this fragment without performing the relief mapping algorithm. While this change is noticeable for the case of using a cube bounding volume, the icosahedron does not improve much because it is already close to being a sphere of radius R_{max} (see "optimized" column in figure 5.2b).

Similarly, if the distance between the ray and the crown center is smaller than R_{min}, we know that this ray will hit the crown. In this case, we compute the intersection point of the ray with this sphere of radius R_{max} and skip the linear search, we only do the binary steps. This optimization, however, did not show a significant effect on the performance. A plausible explanation of this behavior is the performance penalty of branch divergence in current GPU architectures.

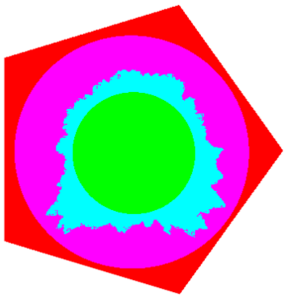


Figure 5.3: Optimizations for relief crowns. The rays for the red fragments do not intersect the circumsphere of the crown, so are directly discarded. Similarly, the green fragments only perform binary search since we know they intersect the crown. The rest of fragments use the full algorithm, some of them end up being discarded (magenta) and others are part of the crown (cyan).

5.2 Detailed crown with leaves

The relief representation of the crowns looks unnatural when we get too close. One of the main reasons is that it lacks leaves and high frequency details. Therefore, we propose another representation for close views.

For each crown, we will place a number of billboards textured with leaves and small branches inside a unit sphere scaled to the maximum radius of the crown. Since this approach by itself would produce rounded spherical crowns, in the fragment shader we discard all the fragments of the billboards that are outside the crown volume. We can do this efficiently with a single query to the crown radius cubemap.

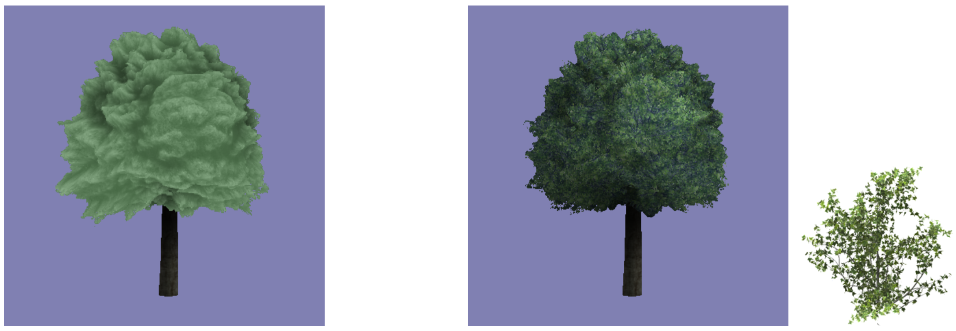


Figure 5.4: Relief crown (left) and detailed crown (right). The texture used for the leaves is shown next to it.

The points are randomly placed following a blue noise distribution. One method to generate blue noise is dart throwing. Instead, we randomly perturbed the voxel centers of the first three levels of an octree. The magnitude of the perturbation is set to be in the range $[-d_L, d_L]$, where d_L is the length of the octree cells at level L. If a point is outside the unit sphere, we discard it. We obtained 320 points shown in the figure below:

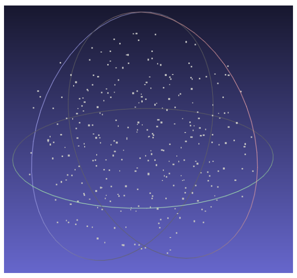


Figure 5.5: Blue noise distribution of points in a sphere.

Each point will potentially become the center of a billboard facing the camera. The geometry shader, depending on the crown projection size on the screen, will decide how many of the points to render, it will create the quad for each one of them, and discard the others. The last few leaves selected to be drawn fade in progressively to minimize popping effects. To ensure a good coverage of all the crown with few billboards in far views, we have previously sorted the points such that for the point p_i in the i-th position:

$$\sum_{j=M}^{i-1} \text{dist}(p_i, p_j) > \sum_{j=M}^{i-1} \text{dist}(p_k, p_j) \quad \forall k > i$$

where M is the number of fixed positions that will always be drawn.

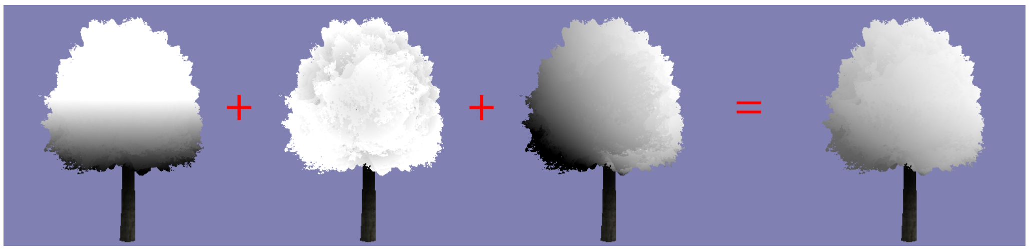
5.3 Shading

Visual realism increases if we add lighting to the scene. Since a full global illumination approach is too computationally expensive to be done in real time, we focused only on ambient occlusion, which will add more detail to the shapes and reliefs of the crowns.

We could bake the shading and store it in another cubemap like the relief, but then those billboards that are placed parallel to the radial direction of the cubemap would look bad. Screen space ambient occlusion techniques are not applicable for the detailed representation using billboards because the depth map would look as a set of squares put one above the other. Also, we do not want to change the depth of individual leaves from a billboard in the fragment shader, as this would prevent the GPU from using early-z culling, which has a high impact on performance.

Therefore, we use an approximation of the ambient occlusion based on the data we already have. Our shading uses three components:

- Height with respect to the crown center: lower branches receive more occlusions from the other branches above them.
- Radial distance to the center of the crown: the outer leaves will receive more light than those deep inside the crown.
- Lambertian shading: classic local illumination model depending on the position of the light source.



(a) Height term **(b)** Radial term **(c)** Diffuse term **(d)** Combination

Figure 5.6: Shading components of a crown.

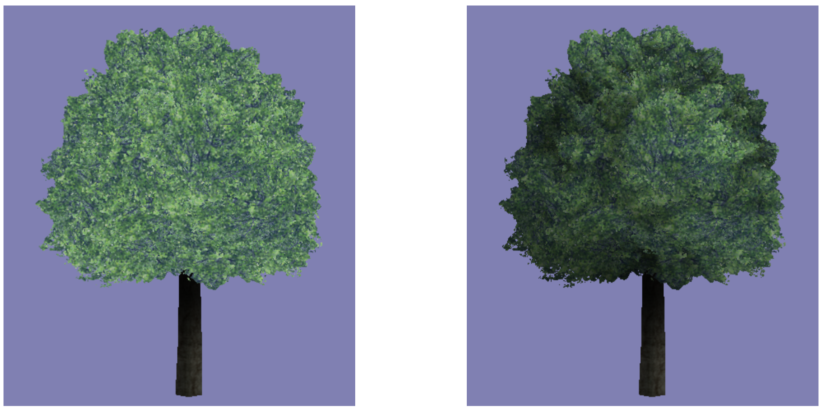


Figure 5.7: Unshaded (left) and shaded (right) tree comparison.

Chapter 6

Terrains with vegetation

In this chapter, we discuss how we integrate our results in a terrain viewer application to display forest areas with thousands of trees.

6.1 Terrain description

Our terrain models are built from the information publicly available at Institut Cartogràfic i Geològic de Catalunya (ICGC) [21]. Specifically, we use their webservices to obtain the orthographic pictures and digital terrain model (DTM) for a desired region. The terrain mesh is built from the DTM image, and then we use a segmentation procedure [10] to classify terrain areas.

The input to our viewer application is:

- A digital terrain model provided as an Wavefront/obj file. We then render this model to a depth map to compute the DTM image automatically, since we need it to know the elevation at each point when placing trees in the scene.
- The orthographic picture for its region.
- The segmentation mask. This mask classifies each region of the terrain as: trees, shrubs, grass or no vegetation. Furthermore, it contains the location of tree trunks.

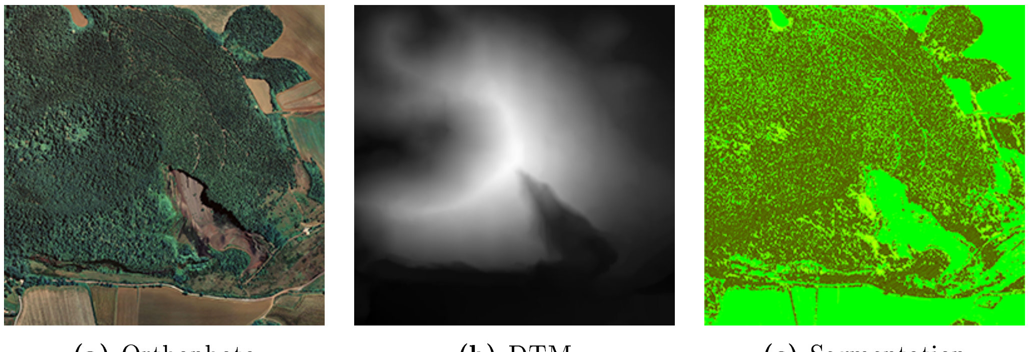

(a) Orthophoto **(b)** DTM **(c)** Segmentation

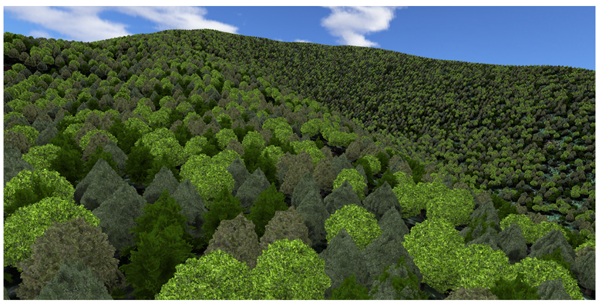
Figure 6.1: Inputs of the viewer application, Volcà del Croscat (Garrotxa) terrain model.

6.2 Vegetation description

There is also another input file describing the vegetation distribution and appearance in the scene. This file contains:

- The list of leaves textures to use.
- The list of crowns to use. Recall that a crown is made up of two cubemaps: the color and the radius cubemap. Also, for each crown we specify in this file the range of scaling and elevation we want to uniformly sample from to obtain the size and position of each individual instance of this crown.
- A set of tuples that will define the actual "species" that can be found during navigation. Each of them is made of:
  - Type of vegetation (tree, bush or grass)
  - Crown model
  - Leaves texture
  - Probability of appearing

During the application startup, the individual plants are created by sampling the distributions specified in this file. The position of the individual trees is given by the segmentation mask, since they will try to match the position of apparent trees seen in the orthophoto. For bushes we sample a position inside a cell around the pixel segmented as shrub. For each tree, bush or grass individual being created, we select one of the crown-leaves pairs defined in the tuples according to their appearance probability by sampling a uniform distribution.


(a) 4 species with equal probability

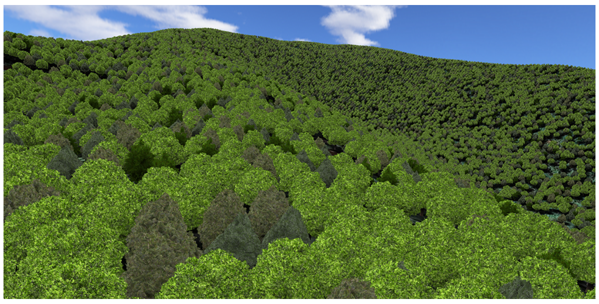

(b) Probabilities: 0.6, 0.2, 0.1, 0.1

Figure 6.2: Specifying different vegetation distributions.

This is a very simple and unnatural model for plant distribution. In the future, we want to explore how to place them more realistically.

6.3 Rendering large vegetation areas

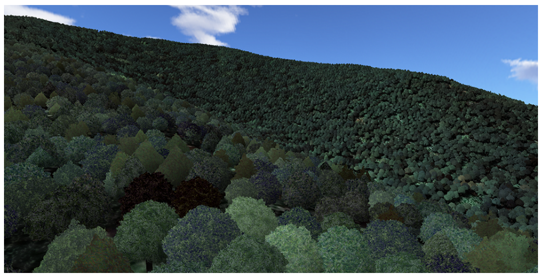
One of our goals is to allow the rendering of large forest scenes with thousands of individuals. We take advantage of modern GPU features to efficiently render the plants. This is the outline of the rendering pipeline:

1. Render the terrain. This will allow the GPU to discard occluded trees using early depth tests.
2. On the CPU, for each plant position perform a frustum culling test.
3. Classify the visible plants either as close or far, based on their distance to the camera.
4. Render close plants using the detailed crown model.
5. Render far plants using the relief crown model.

Each individual plant is created from an instance of its base model. For the case of detailed plants we instance a model made of 320 points (see chapter 5.2), and for the relief crowns we create an instance of an icosahedron.

The input attributes for each instance are: crown translation (position in the world), crown rotation, crown scaling, height of crown center from the ground, average color in the plant position (see below), and plant type. The plant type is an integer used to index the attribute arrays in the shaders that store data that varies only between species like crown cubemaps or leaves textures.

The color each plant has associated corresponds to the value of the orthophoto in the plant position. Using it, we perturbate the resulting plant color and make it more similar to what can be seen in the orthophoto as trees get farther. In addition, this adds more diversity to the plant instances.


(a) Mixing tree models with orthophoto color

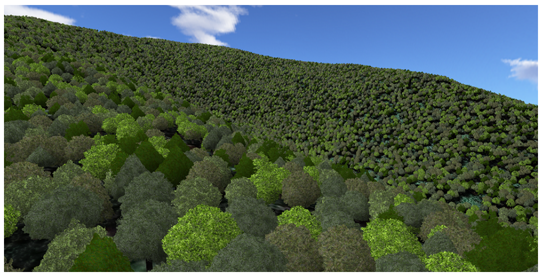

(b) Tree models with their original texture color

Figure 6.3: Comparison between mixing with orthophoto color and using model texture.

6.3.1 Level of detail jittering

Although both detailed and relief representations are based on the same crown volume given by the radius cubemap, due to the different way they are rendered they do not produce the same exact result for a tree. As a result, for some plants the LOD change between the detailed and relief models can be seen.

If we change LOD just on a threshold distance, the effect we see during navigation is that of a wavefront moving away from the camera, corresponding to the switch distance. This regularity makes the LOD switch more apparent than it should be.

To reduce this effect, we introduce jitter to the LOD switch: we define a range of distances $[d_{\min}, d_{\max}]$ in which the plants can change LOD. For each individual, during initialization we randomly sample a variable $t \in [0, 1]$ such that this plant will switch between models at a distance $d = d_{\min} + t \cdot (d_{\max} - d_{\min})$.

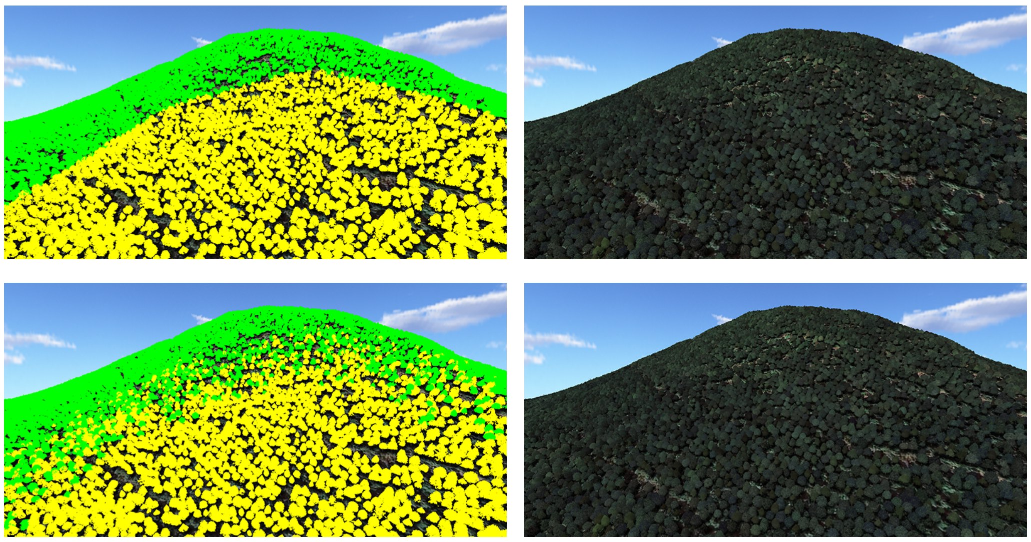


Figure 6.4: Top, LOD switch at fixed $d = 160$. Bottom, LOD switch in range $[120, 200]$. Although both images look almost identical, the visual effect of LOD change during navigation is considerably reduced using jittering.

6.4 Other scene elements

Apart from the foliage, which has been the focus of this project, our viewer needs to render other terrain elements to produce a complete scene. In the following subsections there is a brief description of how we render such elements. Since we used very simple representations for them, we will also mention some improvements or extensions that could be done as a future work.

6.4.1 Trunks

This part is just an adaptation from the previous viewer (see [1]) into the new one, so a brief description is provided for completeness. The trunks are drawn entirely using the geometry shader. A Vertex Buffer Object is built once during initialization, and it contains all the trunk positions in the scene as a point for each one. Then, the geometry shader creates an hexagonal prism with the base located at this point. This prism is composed of various sections along its main axis, so we can make curved trunks. To add variety, the shader randomly samples parameters such as the height, rotation, radius, curvature of the trunk, etc. The fragment shader randomly selects two bark textures, one as a base color and the other one to perturb it.

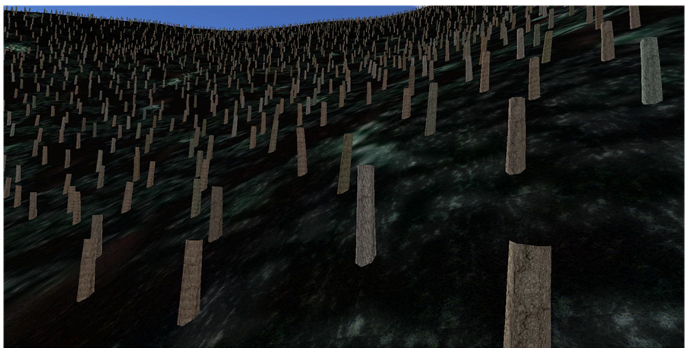


Figure 6.5: Trunks diversity

As one can see in the previous picture, the interface between the trunk and the ground is given by the intersection. One future extension could be modeling the group of roots that extend on the ground. Also, since we modeled dense crowns, our trunks only have the main trunk and no branching.

6.4.2 Grass

Currently, we just apply a grass texture to the ground on those terrain areas near the camera that have been segmented as vegetation of any kind. We assume that there is grass under the trees and shrubs, not only on the areas specifically segmented as grass. If the grass receives shadow from the trees, we modulate the texture with another one that represents the pattern that could produce tree leaves. See the following picture:

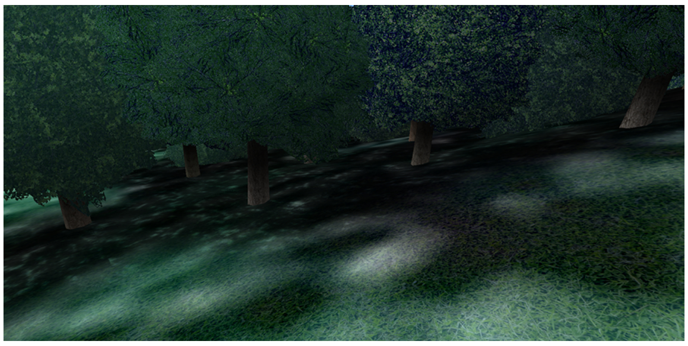
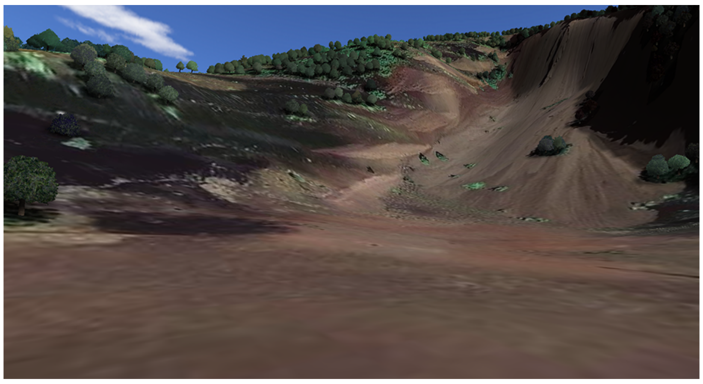


Figure 6.6: Grass and shadows under the trees

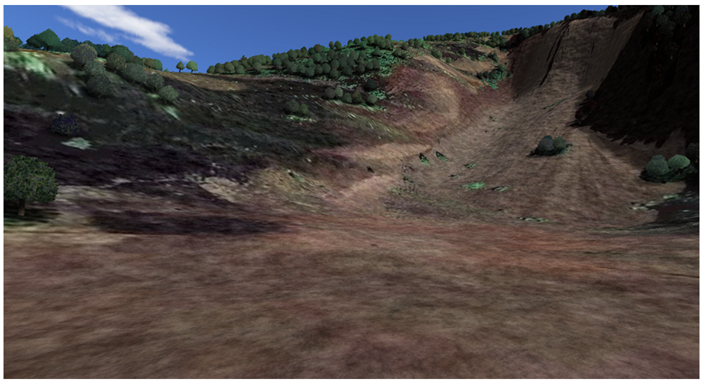
A future extension might be drawing billboards on the ground with a texture of grass blades, and even animating it.

6.4.3 Terrain

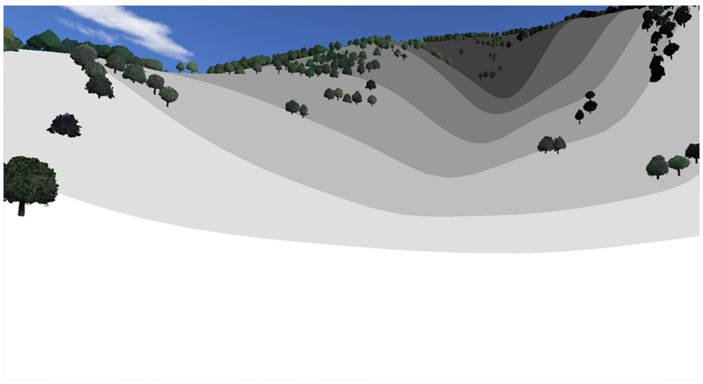
Using only the orthophoto to texture the terrain ground does not provide enough level of detail for close views, since the resolution is about 25 cm/pixel. To add more detail we compute a Fractional Brownian Motion (fBm) perturbation of both the color and the normal by summing successive octaves of a 3D noise function. Since we do this directly on the GPU, it is one of the most expensive steps in the rendering of the scene. Although terrain noise was already implemented in the previous version of the viewer, now the number of octaves to add is determined by the distance of the terrain fragment, which improves performance significantly.



(a) Orthophoto, no perturbation



(b) Orthophoto, 8 to 2 octaves of fBm



(c) Level of detail of the number of octaves

Figure 6.7: Terrain ground rendering with added noise.

There is still many improvements to be done in the terrain rendering. Using a more efficient noise algorithm (maybe moving it to the CPU side) is one of the pending tasks. Moreover, tessellating the terrain and adding a simulated displacement map with rocks and bumps could yield more realistic results than the noise we are currently using. Similarly, it could also be interesting to have a variety of grounds, for example trails with a more uniform and sandy appearance, or screes with both small and large rocks.

6.4.4 Shadows

Shadows add more realism to the scene, since one expects areas under the trees to appear darker. The previous viewer used the vegetation density provided by the segmentation input to estimate the occlusions. Now we render a shadow map using an orthographic camera from the light direction, with a resolution of 4096 × 4096. To provide smooth shadows on the ground, we use Percentage Close Filtering and take 16 samples around the projection using a 16-tap Poisson disk. For the trunks we just use 4 samples.

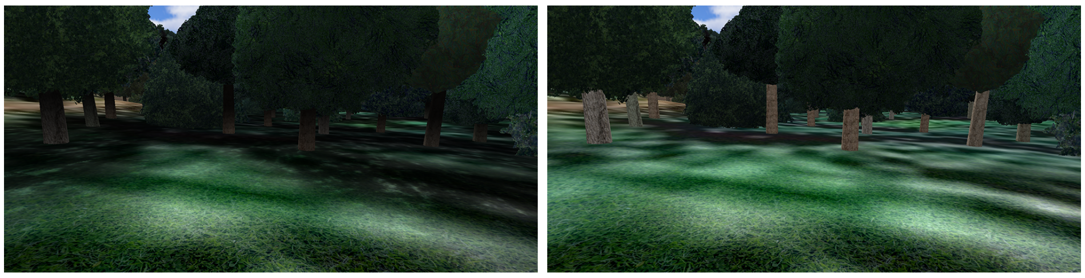


Figure 6.8: Comparison between shadowed and unshadowed terrain.

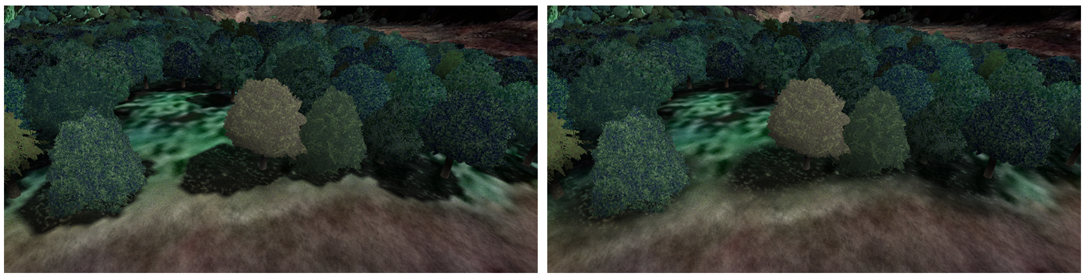


Figure 6.9: Shadow mapping from 1 sample (left) and PCF shadow mapping using a 16-tap Poisson disk.

6.4.5 Sky

The sky simply uses a textured spherical dome, see the images included in this chapter for some examples. In the future, we would like to simulate and add an atmospheric scattering model like in [6].

Chapter 7

Results

In this chapter we will show the visual results obtained with our reconstruction algorithm, the cost of the preprocessing and the performance of our algorithm during rendering.

Unless otherwise specified, the tests have been executed on a laptop with an NVIDIA GeForce GTX 860M graphics card and an Intel Core i7 4712MQ processor running at 2.3-3.3 GHz. The screen resolution is set to Full HD (1920×1080).

7.1 Crown reconstruction

The crown reconstruction is done as a preprocess separated from the rendering program. The following table shows the reconstruction times for a cubemap with a side of 256 pixels. The intervals show the measured times for the crowns in figure 7.1.

step	**time**
base mesh generation	2-3 s
texture synthesis	40-45 min
cubemap assembly	20-25 s
relief extraction	< 1 s
final model composition	< 1 s

Table 7.1: Crown reconstruction times

Base mesh generation is the time needed to solve the bilaplacian linear system. Texture synthesis counts the time needed to produce the three individual textures that will make up the color cubemap: laterals, top and bottom. In this test, the sizes including the merging band are, respectively: 1056×320, 256×256 and 256×256. Cubemap assembly accounts for the time needed to merge the previous images into a continuous cubemap. Relief extraction is the algorithm that computes a heightfield for this cubemap, and the composition is the application of the relief upon the base mesh and storing it into the final radius cubemap.

As we can see, all the generation process would take less than a minute except for the texture synthesis. The algorithm we have implemented is costly due to its per pixel based execution, and it is not easy to run in parallel since each new pixel to synthesize needs the previous ones to be already synthesized. Still, some parallel algorithms have been developed (see [56, 25, 24]) and we would like to improve our synthesis in the future.

In the following figure we can see five reconstruction examples. Note how the added relief avoids a symmetric side silhouette and the seam between the two mirrored meshes is not visible.

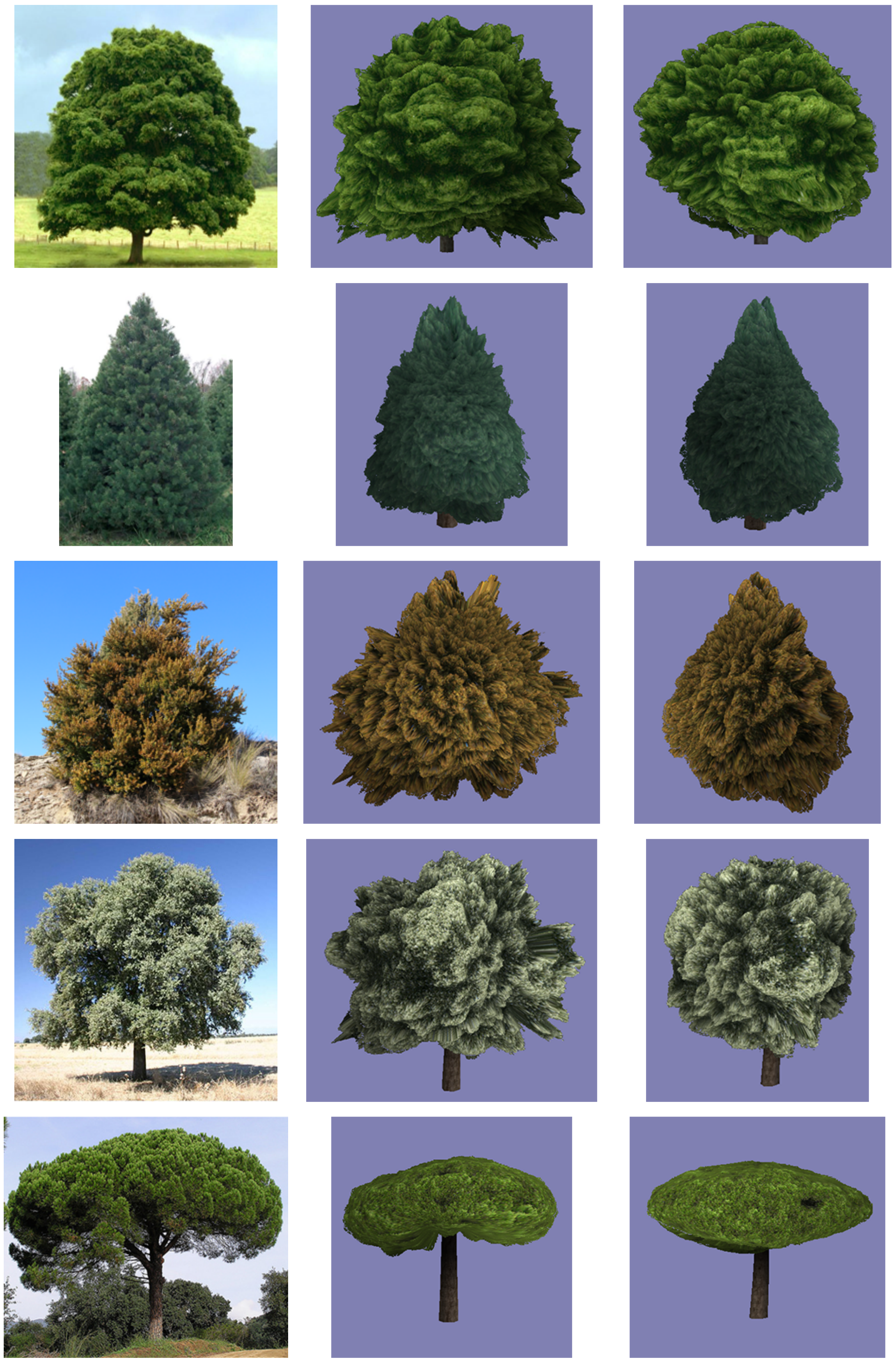


Figure 7.1: Reconstructed relief crowns: input photograph, front view and side view.

7.2 Detailed crowns

The detailed crowns combine a volume like the ones we have seen in the previous section with a leaves texture. This allows us to multiply the variety of different trees we can have, as shown in the next images:

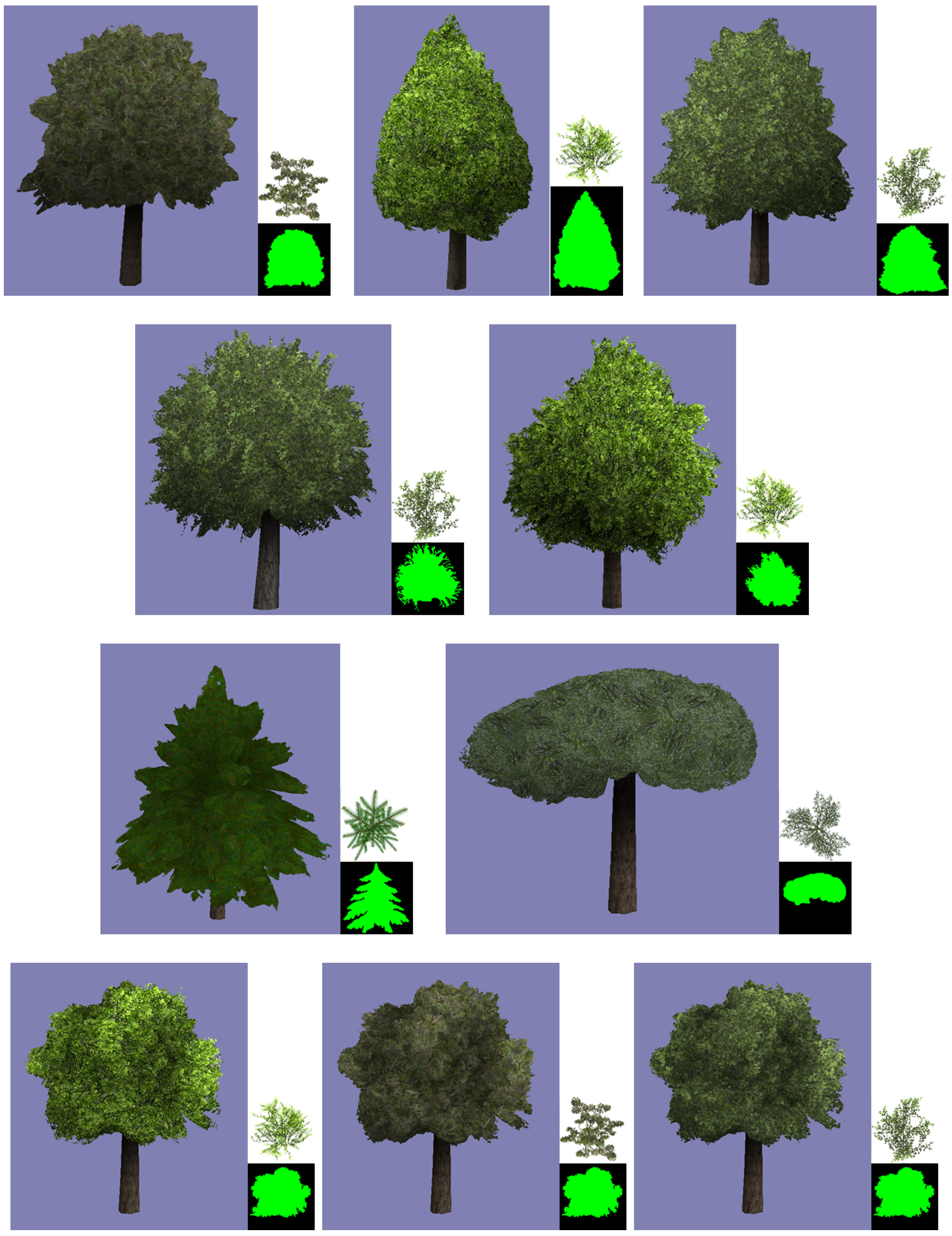
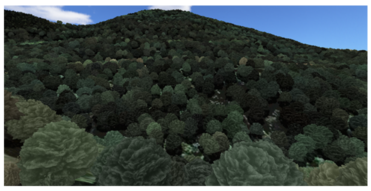
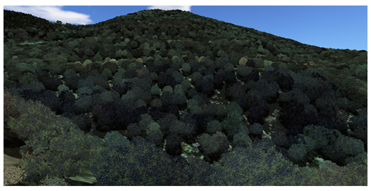


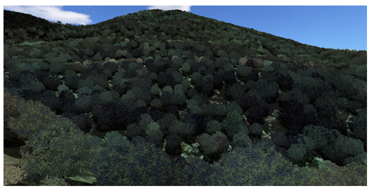
Figure 7.2: Detailed crowns with various leaves. The last row shows how we can add more variety combining different textures with the same crown model. For clarity, the mask that generated the base crown is included on the bottom right, as well as the leaves texture used on the billboards.

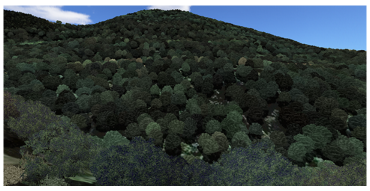
7.3 Level of detail test

This test evaluates the performance of combining both representations (detailed and relief) in a real scene and the effect of modifying the switch distance. The chosen scene is a forest landscape with views on a mountain side, which offers a wide range of tree distances in sight.


(a) $d = 0$


(b) $d = 100$


(c) $d = \infty$


(d) $d = 50$

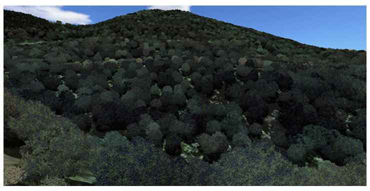

(e) $d = 200$

Figure 7.3: Level of detail visual results at various distances d.

distance	N_{detail}	N_{relief}	T_{detail}	T_{relief}	T_{total}
0	0	27424	0.0	21.0	21.0
50	97	27327	3.8	16.9	20.7
100	678	26746	12.7	12.5	25.2
150	1347	26077	16.8	9.9	26.7
200	2318	25106	19.5	8.3	27.8
250	3477	23947	21.6	6.8	28.4
∞	27424	0	46.9	0.0	46.9

Table 7.2: Level of detail distance threshold test: number N of trees for each representation and time T in milliseconds to render them.

We can see that the relief representation is obviously more efficient, but also provides worse visual results for close views due to its lack of high frequencies and leaves. Also, there is a case not contemplated in this test in which the relief representation becomes really expensive: when the tree is viewed very close and the number of pixels executing the relief mapping is very high.

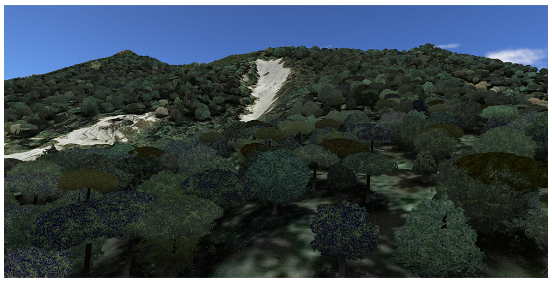
Regarding the detailed representation, the first and closer trees are the most expensive because we compute the number of leaves depending on the projection of the tree on the screen. Typical ranges are between 128 and 20 leaves.

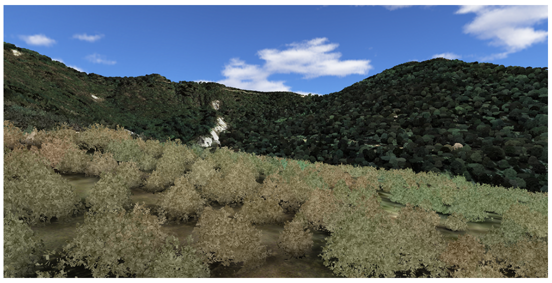
As can be seen in the test results, increasing the distance at which we switch representations also increases the rendering time. However, the visual quality of relief trees is poor for close views. Therefore, we decided to set the LOD threshold in the range 100-200. Recall that trees do not change representation at a fixed distance - unlike in this test - but we add a jittering to minimize the LOD wavefront effect.

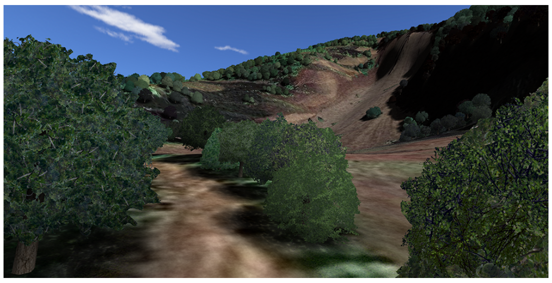
7.4 Test scenes

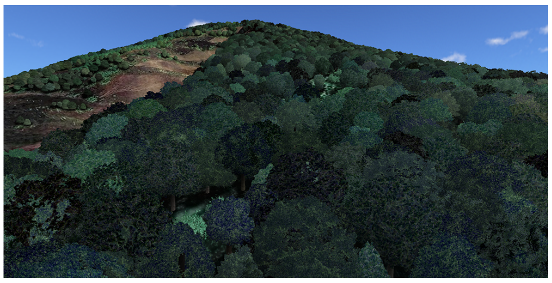
Now we will analyze the performance of our rendering algorithm in test cases extracted from real segmented terrains.

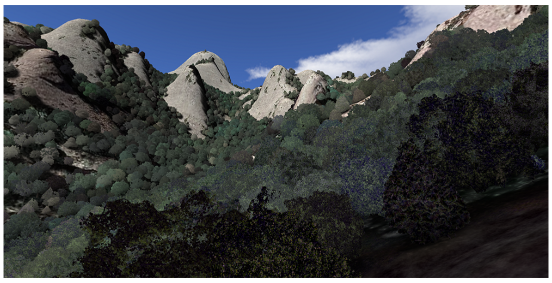
Below are the different scenes and chosen viewpoints. They show different kinds of landscapes, with varying ratios of trees, shrubs, and bare terrain, and different kinds of vegetation crowns all reconstructed from input photographs.

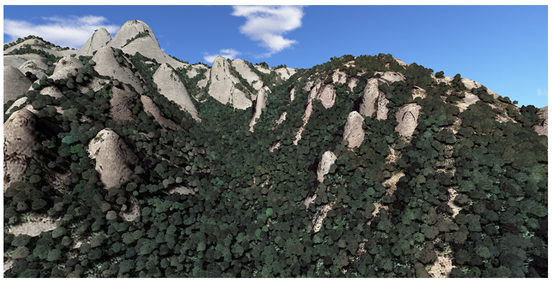

(a) Creu de Gurb 1

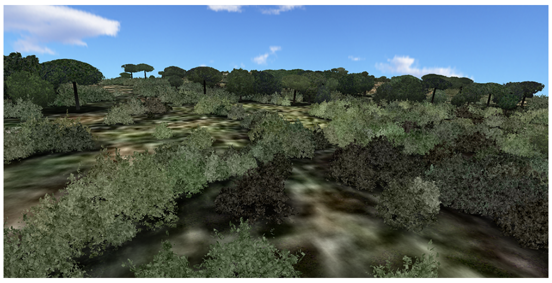

(b) Creu de Gurb 2


(c) Croscat 1


(d) Croscat 2


(e) Montserrat 1


(f) Montserrat 2


(g) Garraf 1

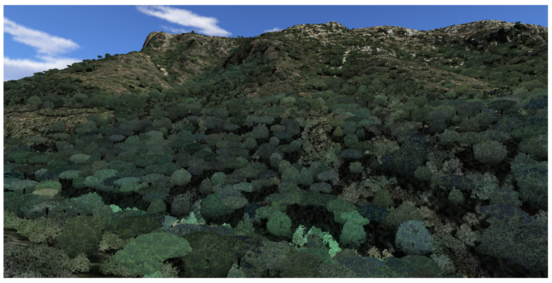

(h) Garraf 2

Figure 7.4: Test scenes.

The following tables show the number of plants drawn for each representation as well as the number of trunks drawn (only for trees), and the rendering times for each scene element.

scene	N_{detail}	N_{relief}	N_{trunks}	T_{detail}	T_{relief}	T_{trunks}	T_{terrain}
Creu de Gurb 1	887	75736	39162	15.2	13.3	17.7	1.2
Creu de Gurb 2	3527	55172	39162	11.8	13.4	14.5	0.6
Croscat 1	718	43691	62333	13.3	10.4	27.5	4.2
Croscat 2	1753	45425	62333	42.1	9.7	27.6	0.6
Montserrat 1	1847	28496	13305	14.8	5.9	6.1	2.3
Montserrat 2	768	29243	13305	2.5	10.4	6.0	2.2
Garraf 1	3714	102006	9353	17.5	7.3	6.0	3.6
Garraf 2	9502	92986	9353	28.1	16.7	6.2	1.8

scene	T_{veg}	T_{total}	FPS_{veg}	$\text{FPS}_{\text{total}}$
Creu de Gurb 1	28.5	47.4	35.1	21.1
Creu de Gurb 2	25.2	40.3	39.7	24.8
Croscat 1	23.7	55.4	42.2	18.1
Croscat 2	51.8	80.0	19.3	12.5
Montserrat 1	20.7	29.1	48.3	34.4
Montserrat 2	12.9	21.1	77.5	47.4
Garraf 1	24.8	34.4	40.3	29.1
Garraf 2	44.8	52.8	22.3	18.9

Table 7.3: Test scenes performance test. Times in milliseconds for Full HD rendering. Bottom table shows the total of time for vegetation rendering, and the total time for complete scene rendering, as well as the frames per second.

First, we can say that our vegetation rendering algorithm runs in real time for almost all the test scenarios (see columns T_{veg} and FPS_{veg}). The bottleneck is the rendering of the detailed representation in those views in which we have many close trees, because the number of leaves drawn per tree is very high.

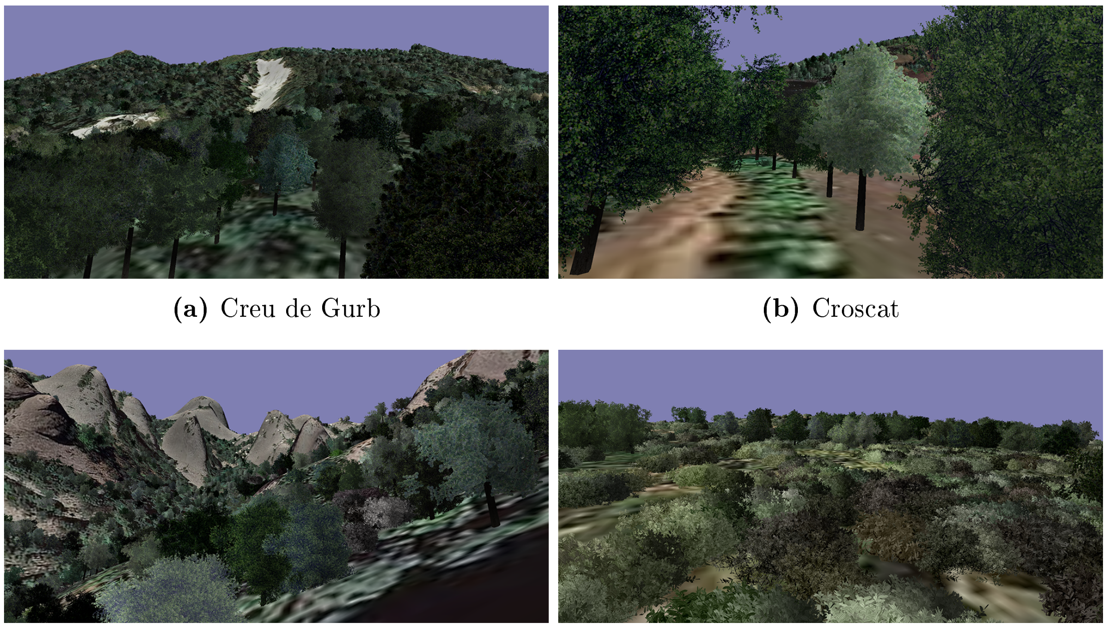
If we now look at the overall results, T_{total} and $\text{FPS}_{\text{total}}$, we see that on average the viewer runs on real time. On scenes with a large number of trees (Creu de Gurb, Croscat) the trunks can be the most expensive element to render, since we have not optimized them at all. Their geometry shader produces always a triangle strip with 42 vertices for each trunk input point. One possible optimization could be creating the geometry of the prism depending on the distance to the camera, using a single billboard for farther trunks and more rounded prisms on close views.

Lastly, we observe that the terrain overhead is usually very small. Only in the cases in which a considerable amount of bare rock or ground is visible, the computation of the noise to perturb the colors and normals becomes perceptible.

7.5 Comparison with previous viewer

We now compare our results with the previous version of the terrain viewer, the one used in [1]. That version used the tessellation shaders to create the leaves billboards around the crown. The shape of the crown was controlled by the exponents of two cosine functions: one for the upper half of the crown and another one for the lower.

The following results were produced using an early version of the viewer developed in this project, one that used the same shaders to render vegetation as [1]. This is why there is no skydome and the terrain shader is just a rendering of the orthophoto. Nevertheless, this still allows us to compare the visual quality and rendering cost of the vegetation.



(a) Creu de Gurb **(b)** Croscat

(c) Montserrat **(d)** Garraf

Figure 7.5: Test scenes rendered using the old method.

Visually, the crowns look a lot more similar between them than in our results because the cosine function used to define the shapes only accounts for ellipsoidal crowns, and thus it is difficult to design a particular forest look (compare the pine trees in Garraf 1). Furthermore, it is difficult to control the desired level of detail and, depending on the settings, trees can disappear in the distance because very few billboards are used to render them. Our raycasted relief representation for far trees does not suffer from this problem while still allowing for an easy level of detail control using the ray step size and number of steps.

These are the measured rendering times for the vegetation in the test scenes. For convenience, we included a copy of the vegetation rendering time shown in table 7.3 as well as the total number of plants drawn. As we can see, for a reasonable quality and level of detail, the previous method only achieved interactive frame rates.

scene	$N^{(\text{old})}_{\text{plants}}$	$T^{(\text{old})}_{\text{veg}}$	$T^{(\text{new})}_{\text{veg}}$	$N^{(\text{new})}_{\text{plants}}$
Creu de Gurb 1	45051	199.3	28.5	76623
Creu de Gurb 2	45051	220.3	25.2	58699
Croscat 1	51246	207.7	23.7	44409
Croscat 2	51246	301.8	51.8	47178
Montserrat 1	31087	119.7	20.7	30343
Montserrat 2	50406	198.2	12.9	30011
Garraf 1	137478	377.5	24.8	105720
Garraf 2	137478	1140.1	44.8	102488

Table 7.4: Comparison of rendering times between old and new method. Times in milliseconds.

Chapter 8

Conclusions and future work

In this project, we have successfully designed and implemented a method for crown reconstruction from photographs and a representation which allows an efficient rendering with good visual quality. The main contributions have been:

- Our method is capable of reconstructing a variety of crowns from a single input photograph, and the obtained model is very similar to the picture.
- We proposed two different representations for crown foliage rendering that allow a level of detail transition with minimum impact.
- We can render real terrain scenarios with tens of thousands of trees in real time.
- The performance with respect to the previous rendering method has been improved by an order of magnitude.

This project has been the starting point of my PhD studies. Therefore, we still have future work to do in tree reconstruction from photographs and related semi-procedural methods. These are some ideas for the crown reconstruction and rendering that we would like to explore next:

- Currently the input requires the segmentation of the crown foliage by the user. Most of the matting algorithms we have seen require some kind of input by the user, like scribbles on both foreground and background, to start the process. We could try to use knowledge about the kind of pictures we want to process to provide at least a first fully automatic segmentation proposal to the user.
- The base mesh is extracted as a single volume from the segmented crown. For the case of more sparse trees with dense subcrowns, or dense trees that have a clear "blobby" structure, we could apply the reconstruction for each segmented subcrown.
- Instead of synthesizing the color cubemap textures from scratch, we could start from the segmented crown foliage on one lateral face and synthesize new texture around it. This way, for a given viewpoint the tree could look almost identical to the photograph.
- A radial representation such as the one we have presented is limited in the amount of crowns we can reproduce. For example, crowns with concavities in the radial

directions cannot be represented, since we only store one distance along this direction. Similarly, we do not allow transparencies through the crown. We could research the use of other structures, for example octrees, to use a volumetric representation instead of a radial one. In this direction, we could leverage techniques from the field of volume rendering to efficiently render the levels of detail.

- During rendering, we are using a very simple and non-physically based approximation for the tree ambient lighting. A more realistic approach would be precomputing it using an appropiate BRDF model for the leaves and crown light propagation and store it in the model representation.
- When the leaves billboards are cropped to match the crown volume shape, we are doing it on a per fragment test and some leaves are partially discarded. We want to extend the leaves textures with information about the center of each individual leaf so a fragment is only discarded if its associated leaf center is outside the crown volume.
- Finally, the plant distribution we used is random and looks very unnatural. Species tend to grow in certain areas according to distribution of light, humidity, soil properties, other species... There exist already published works that simulate the development of plant ecosystems, but we want to see how we could integrate the information of existing datasets to directly extract the distribution of the species: crown heights provided by Digital Surface Models, color of the crowns from the orthophoto, terrain altitude, etc.

Bibliography

[1] C. Andújar, A. Chica, M. A. Vico, S. Moya, and P. Brunet. Inexpensive reconstruction and rendering of realistic roadside landscapes. *Computer Graphics Forum*, pages n/a–n/a, 2014.

[2] M. Aono and T. Kunii. Botanical tree image generation. *IEEE Comput. Graph. Appl.*, 4(5):10–34, May 1984.

[3] J. Bloomenthal. Modeling the mighty maple. *SIGGRAPH Comput. Graph.*, 19(3):305–311, July 1985.

[4] F. Boudon, A. Meyer, and C. Godin. Survey on Computer Representations of Trees for Realistic and Efficient Rendering. Rapport de recherche 2301, LIRIS, Université Claude Bernard Lyon 1, 2006.

[5] D. Bradley, D. Nowrouzezahrai, and P. Beardsley. Image-based Reconstruction and Synthesis of Dense Foliage. *ACM Trans. Graph.*, 32(4):74:1—-74:10, July 2013.

[6] E. Bruneton and F. Neyret. Precomputed atmospheric scattering. *Comput. Graph. Forum*, 27(4):1079–1086, June 2008. Special Issue: Proceedings of the 19th Eurographics Symposium on Rendering 2008.

[7] E. Bruneton and F. Neyret. Real-time Realistic Rendering and Lighting of Forests. *Computer Graphics Forum*, 31(2pt1):373–382, May 2012.

[8] P. J. Burt and E. H. Adelson. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, 31:532–540, 1983.

[9] D. Cohen. Computer simulation of biological pattern generation processes. *Nature*, 216:246–248, 1967.

[10] M. Comino. 3D Reconstruction of vegetation from orthophotos, 2013.

[11] P. de Reffye, C. Edelin, J. Françon, M. Jaeger, and C. Puech. Plant models faithful to botanical structure and development. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 151–158, New York, NY, USA, 1988. ACM.

[12] P. Decaudin and F. Neyret. Rendering forest scenes in real-time. In A. Keller and H. W. Jensen, editors, *Rendering Techniques*, pages 93–102. Eurographics Association, 2004.

[13] O. Deussen, C. Colditz, M. Stamminger, and G. Drettakis. Interactive visualization of complex plant ecosystems. In *Proceedings of the Conference on Visualization '02*, VIS '02, pages 219–226, Washington, DC, USA, 2002. IEEE Computer Society.

[14] O. Deussen and B. Lintermann. *Digital Design of Nature: Computer Generated Plants and Organics*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[15] A. A. Efros and W. T. Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 341–346, New York, NY, USA, 2001. ACM.

[16] M. Glencross, G. J. Ward, F. Melendez, C. Jay, J. Liu, and R. Hubbold. A perceptually validated model for surface depth hallucination. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 59:1–59:8, New York, NY, USA, 2008. ACM.

[17] Google earth. http://earth.google.com.

[18] Google street view. http://www.google.com/maps/views/streetview.

[19] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[20] H. Honda. Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology*, 31(2):331 – 338, 1971.

[21] Institut cartogràfic i geològic de catalunya. http://www.icgc.cat.

[22] A. Jakulin. Interactive vegetation rendering with slicing and blending. In A. d. Sousa and J. C. Torres, editors, *Proc. of Eurographics (Short Presentations)*, Interlaken, Switzerland, 2000.

[23] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick. Graphcut textures: Image and video synthesis using graph cuts. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 277–286, New York, NY, USA, 2003. ACM.

[24] A. Lasram and S. Lefebvre. Parallel patch-based texture synthesis. In *High Performance Graphics conference proceedings*, 2012.

[25] S. Lefebvre and H. Hoppe. Parallel controllable texture synthesis. *ACM Trans. Graph.*, 24(3):777–786, July 2005.

[26] A. Lindenmayer. Mathematical models for cellular interactions in development: Parts i and ii. *Journal of theoretical biology*, 18(3):280–315, 1968.

[27] B. Lintermann and O. Deussen. Interactive modeling of plants. *IEEE Comput. Graph. Appl.*, 19(1):56–65, Jan. 1999.

[28] Y. Livny, S. Pirk, Z. Cheng, F. Yan, O. Deussen, D. Cohen-Or, and B. Chen. Texture-lobes for Tree Modelling. *ACM Trans. Graph.*, 30(4):53:1—-53:10, 2011.

[29] Y. Livny, F. Yan, M. Olson, B. Chen, H. Zhang, and J. El-Sana. Automatic Reconstruction of Tree Skeletal Structures from Point Clouds. *ACM Transactions on ...*, 29(6):151:1—-151:8, 2010.

[30] R. Měch and P. Prusinkiewicz. Visual models of plants interacting with their environment. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 397–410, New York, NY, USA, 1996. ACM.

[31] Nasa world wind. http://worldwind.arc.nasa.gov.

[32] B. Neubert, T. Franken, and O. Deussen. Approximate Image-based Tree-modeling Using Particle Flows. *ACM Trans. Graph.*, 26(3), 2007.

[33] M. Okabe, S. Owada, and T. Igarashi. Interactive Design of Botanical Trees using Free-hand Sketches and Example-based Editing. *Computer Graphics Forum*, 24(3):487–496, 2005.

[34] P. E. Oppenheimer. Real time design and animation of fractal plants and trees. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 55–64, New York, NY, USA, 1986. ACM.

[35] W. Palubicki. *Fuzzy Plant Modeling with OpenGL- Novel Approaches in Simulating Phototropism and Environmental Conditions.* VDM Verlag, Saarbr&#252;cken, Germany, Germany, 2007.

[36] W. Palubicki, K. Horel, S. Longay, A. Runions, B. Lane, R. Měch, and P. Prusinkiewicz. Self-organizing tree models for image synthesis. In *ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, pages 58:1–58:10, New York, NY, USA, 2009. ACM.

[37] P. Pérez, M. Gangnet, and A. Blake. Poisson image editing. *ACM Trans. Graph.*, 22(3):313–318, July 2003.

[38] F. Policarpo, M. M. Oliveira, and J. L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pages 155–162, New York, NY, USA, 2005. ACM.

[39] P. Prusinkiewicz, M. S. Hammel, and E. Mjolsness. Animation of plant development. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 351–360, New York, NY, USA, 1993. ACM.

[40] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[41] P. Prusinkiewicz, L. Mündermann, R. Karwowski, and B. Lane. The use of positional information in the modeling of plants. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 289–300, New York, NY, USA, 2001. ACM.

[42] P. Prusinkiewicz, L. Mündermann, R. Karwowski, and B. Lane. The use of positional information in the modeling of plants. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 289–300, New York, NY, USA, 2001. ACM.

[43] X. Qin, E. Nakamae, K. Tadamura, and Y. Nagai. Fast photo-realistic rendering of trees in daylight. *Computer Graphics Forum*, 22(3):243–252, 2003.

[44] L. Quan, P. Tan, G. Zeng, L. Yuan, J. Wang, and S. B. Kang. Image-based Plant Modeling. *ACM Trans. Graph.*, 25(3):599–604, 2006.

[45] A. Reche, I. Martin, and G. Drettakis. Volumetric Reconstruction and Interactive Rendering of Trees from Photographs. *ACM Transactions on Graphics*, 23(3):720–727, 2004.

[46] W. T. Reeves and R. Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *SIGGRAPH Comput. Graph.*, 19(3):313–322, July 1985.

[47] Y. Rodkaew, P. Chongstitvatana, S. Siripant, and C. Lursinsap. Particle systems for plant modeling. *Plant growth modeling and applications*, pages 210–217, 2003.

[48] A. Runions, B. Lane, and P. Prusinkiewicz. Modeling Trees with a Space Colonization Algorithm. In *Proceedings of the Third Eurographics Conference on Natural Phenomena*, NPH'07, pages 63–70, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[49] I. Shlyakhter, M. Rozenoer, J. Dorsey, and S. Teller. Reconstructing 3D Tree Models from Instrumented Photographs. *IEEE Computer Graphics and Applications*, 21(3):53–61, May 2001.

[50] R. Sun, J. Jia, H. Li, and M. Jaeger. Image-based Lightweight Tree Modeling. In *Proceedings of the 8th International Conference on Virtual Reality Continuum and Its Applications in Industry*, volume 1 of *VRCAI '09*, pages 17–22, New York, NY, USA, 2009. ACM.

[51] P. Tan, T. Fang, J. Xiao, P. Zhao, and L. Quan. Single Image Tree Modeling. *ACM Transactions on Graphics*, 27(5):108:1—-108:7, Dec. 2008.

[52] P. Tan, G. Zeng, J. Wang, S. B. Kang, and L. Quan. Image-based Tree Modeling. *ACM Transactions on Graphics*, 26(3):87, 2007.

[53] J. Weber and J. Penn. Creation and rendering of realistic trees. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 119–128, New York, NY, USA, 1995. ACM.

[54] L.-Y. Wei, S. Lefebvre, V. Kwatra, and G. Turk. State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report, EG-STAR*. Eurographics Association, 2009.

[55] L.-Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 479–488, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[56] L.-Y. Wei and M. Levoy. Order-independent texture synthesis. Technical report, TR 2002, 2002.

[57] J. Wither, F. Boudon, M.-P. Cani, and C. Godin. Structure from silhouettes: a new paradigm for fast sketch-based design of trees. *Computer Graphics Forum*, 28(2):541–550, 2009.

[58] H. Xu, N. Gossett, and B. Chen. Knowledge and heuristic-based modeling of laser-scanned trees. *ACM Trans. Graph.*, 26(4), Oct. 2007.