Wood Texture by Growth Simulation

M1 Internship Report

Jérémie DUMAS
École Normale Supérieure de Lyon
Supervisor: Pierre POULIN

Université de Montréal

May-July 2011
Laboratoire d'Informatique Graphique de l'Université de Montréal

Biology and computer science are often closely entwined, as it was the case with this research internship: tree cross-section textures modeling designed for computer graphic applications. From the introduction of solid textures by Peachey [Pea85] or Perlin [Per85] in 1985, to the still undergoing work of Prusinkiewicz [PL96], faithful modelling of flora had been of interest for biologists as well as for 3D artists. In the time spent in the LIGUM, we tried to elaborate procedural textures representing the inner structure of a tree, based on biological and geometrical considerations as mush as possible. We were also interested in efficient computations and representations for computer graphics applications.

Contents

1 Generalities

1.1 Introduction

1.1.1 Solid Textures

One topic that arises frequently in the domain of computer graphics is the question of generating high-quality textures for realistic rendering of objects, often defined by a polygonal mesh. Because it is too expensive in memory and processing to increase the number of polygons defining its mesh, textures are used to simulate the visual effects of details on the surface of a given object.

Those 2D textures have multiple uses: *color maps* are the most common form of texturing, *bump maps* are used to change normals on a given surface (thus altering the way a surface element reflects light), *displacement maps* change the geometry of the given surface only when necessary, and so on.

The main issue of such methods is that finding a convenient mapping from a rectangular 2D texture to the surface of an arbitrarily shaped object is not an easy task. Thus *3D* or *solid texture* approaches have since been designed (see [Pea85] and [Per85]), though not as popular as their 2D counterpart due to numerous drawbacks.

The main idea is to define a function $\rho(x, y, z)$ that maps a 3D point to its associated color (for color mapping). Thus all we have to do to render an object is to evaluate ρ at each point of its surface that appears on the screen: it is as if the object has been *carved* from a solid bloc of material. The method is viable as long as ρ can be defined as a simple function of (x, y, z). However if we want a more complex solid texture, we have to discretize ρ, store its values in a 3D grid for example, and interpolate the color of points with non-integer coordinates. Therefore, the more details we want for function ρ, the more memory it will require to store the 3D grid, precluding the possibility of making high-quality solid textures.

During this research internship, we focused on the particular problem of wood texture generation, with the aim of simulating high-quality wood solid textures. Wood textures may seem to be a subject already revisited, but its omnipresence as material in human design and artifacts makes it even more challenging to generate something either correct from a botanical point of view, satisfying for an artist, or manageable for a rendering system.

Thus our main goals were to encompass a realistic model for the interior of a tree, with an efficient and controllable simulation of the growth process. Although at first botanical concerns were not paramount, we tried to take them into account during our research, with the hope of eventually modeling consistent and rich internal structures. The idea is that eventually such a structure can inspire us and even instruct us about mechanical forces or tensions that come to play inside a tree, as well as some other natural and emerging phenomena we would like to reproduce.

1.1.2 Botanical Considerations

The first question to ponder when modeling natural phenomena is which characteristics of a process we would like to encompass exactly. There is always a trade-off between the biological plausibility of a simulation, and the expected "quality" of a computer graphics wood texture: beautiful-looking pictures of wood may not be biologically realistic, and vice-versa. Another key element to consider is at which scale we want to simulate the phenomena. We will now explain some simple aspects of wood growth we tried to reproduce in the next stages of our development.

Annual ring patterns One of the most recognisable feature in a cross-section of a tree is the annual ring patterns characteristic to the seasonal growth cycle of wood cells. Each year the cells in the vascular cambium (the part between the bark and the inner wood) divide and grow to form a new ring. In a ring we usually distinguish two parts: the *earlywood*, or *springwood*, which is of lighter color because the cells grew rapidly due to availability of water or a more favourable weather ; the *latewood*, or *summerwood*, which is darker because the cells grew more tightly, due to scant water and other factors. This of course varies between species, but that is the general idea.

The transition from earlywood to latewood is quite smooth, although non linear, one ring being composed mainly of earlywood (whose cells are wider), whereas the transition from latewood to earlywood is usually more abrupt: the tree growth often stops during winter, only to resume in the next spring.

Wood is usually classified as either softwood (such as pine), or hardwood (such as oak), the latter having a more complex internal structure (presence of vessels of various sizes and shapes notably).

Knots Another fairly visible feature of wood textures is the presence of knots. This imperfection is due to prematurely dead branches or buds in early stages of its development, around which the remaining cells grow steadily. Hence a tree cross-section is usually more knotty in the center. A knot is also conical in shape, so it appears in cross-sections as an ellipsoid. It varies in sizes and forms, depending on the conditions that lead the original branch to die. Knots do not usually influence the stiffness of wood.

Heartwood and sapwood As the tree grows, a natural chemical process takes place in the innermost regions, and leads to the creation of dead heartwood, whereas the surrounding sapwood remains alive. Heartwood has usually a darker reddish color, and its formation is not correlated with the annual ring pattern. It is not to be confused with decay, which may also lead to the creation of a darker inner region in the wood. See Figure 1 for an illustration.

Other factors The growth process may be influenced by other various environmental factors such as insects and animals feeding on the tree, other diseases caused by microorganisms, weather influences, etc. For example mechanical forces caused by wind, gravity, presence of a fence nearby, or other factors, will greatly influence the final shape

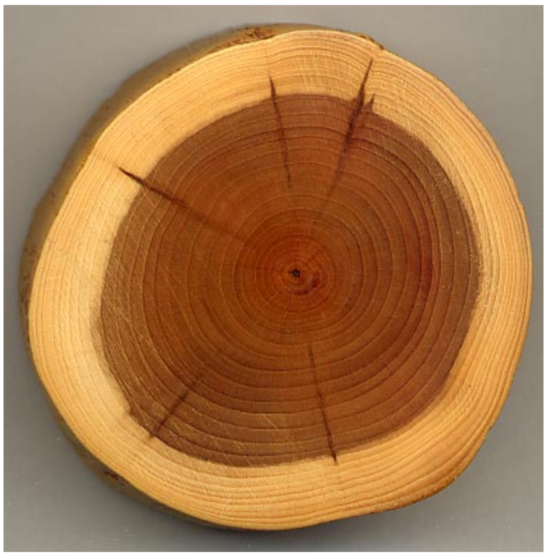

Figure 1: A section of a Yew branch showing 27 annual growth rings, pale sapwood and dark heartwood, and pith (dark central spot). Source: Wikipedia. Author: MPF.

of the tree. Likewise, the availability of light or water during its development will have severe consequences on the resulting tree: one will grow more gnarled because cells died randomly by lack of water, while another will have a rapid growth due to extensive light availability.

Of course we had no intention to reproduce precisely each possible phenomenon, but we tried to develop a flexible model which can be controlled by intuitive tunable parameters, so that some features can be simulated, while others would require more reflection, but could be added on.

1.2 Previous Work

1.2.1 Procedural Textures

As stated in the introduction, 2D textures are convenient structures to increase the level of details on an object without any major overhead, yet computing a desired mapping to a particular surface may be difficult. A contrario, 3D textures are free of most mapping issues, but can be memory-expensive. Bitmap textures use raster graphics images (represented by a 2D or 3D grid of pixels — called texels) and have a limited level of details to offer when rendering an object. In contrast, procedural textures may rely on vector graphics (created with simple primitives, such as lines or curves) as well as fractal noise for the rendering process, and can offer multiple levels of details depending on the sampling resolution and kernel shape which are used. As we render such textures from different distances, we also have to be aware of filtering issues, for jaggies and interference patterns (called *moiré patterns*) may appear, resulting from insufficient sampling of high-frequency functions.

Solid wood textures have been introduced in 1985 by Peachey [Pea85], and while being simplistic, the generative method was nonetheless a fully procedural solid texture. We also took interest in the work of Norell [Nor09] for the creation log end face images, because it showed a procedural 2D method to create a texture of a wood cross-section of arbitrary resolution.

1.2.2 L-systems

An L-system is a variant of a formal grammar introduced in 1968 by Lindenmayer [Lin68] as a way to model the growth processes of plant development. In its most general form, it can be defined as a tuple $G = (V, \omega, P)$, where V is a set of symbols called the *alphabet*, ω is a string of symbols defining the initial state of the system, called *axiom*, and P is a set of rewriting rules. A rewriting rule is merely a tuple $A \rightarrow B$ indicating that A, the string of symbols, is to be replaced by B when applying the rule.

The main difference between L-systems and formal grammars is that at each iteration, every possible rewriting rule is applied in *parallel* to the current state of the system. Thus languages defined by L-systems form a strict subset of the usual formal languages. Like formal grammars, an L-system can be *determinist* and/or *context-free* depending on how the rewriting rules are defined.

Plant-growth modeling with L-systems and their application in computer graphics have been popularised by Prusinkiewicz over the last decades [PL96]. A variant of L-systems has been designed by Terraz et al. [TGM+09] to simulate wood growth as well as solid textures. By repeatedly splitting or adding new volumes to an initial prism, according to a mechanism defined by their grammar, they achieve fair simulations of wood growth processes, defined by its resulting polygonal mesh (see Figure 2).

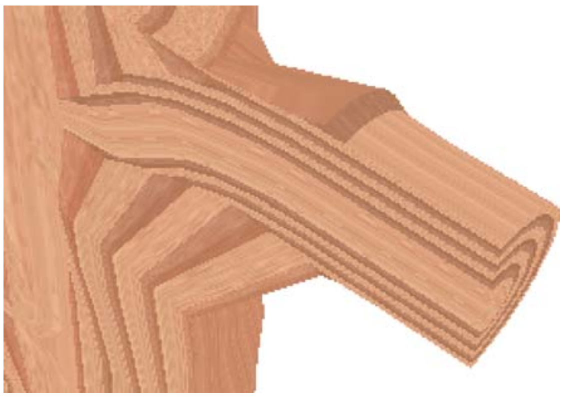

Figure 2: Sample rendered image from Terraz et al. [TGM+09]

However while plants modeled from L-systems are quite faithful with their original botanical parameters, it is not easy for the heathen to develop an appropriate grammar that will lead to expected results, which is why L-systems are still uncommon among computer graphics artists.

1.2.3 Voxel Simulation

An alternative to the procedural texture methods is to use a discrete approach. Closest to the aforementioned L-system growth simulation, a voxel simulation algorithm was used by Buchanan [Buc98] to produce solid wood textures. The main idea of a voxelized method is to split the 3D space in a regular grid, using small volume elements (called *voxels*) to store information.

In the paper of Buchanan, voxels hold wood cells, which grow and populate adjacent cells, with embedded information such as their age or color, thus yielding a solid texture stored as a simple 3D grid. Despite the obvious memory issue pertaining to high-quality textures (yet recent computers can handle large amount of data), voxel discretization of 3D space has other drawbacks. For example it tends to favour growth along axis directions, so this issue has to be taken into account and compensations introduced.

1.3 Our Approach

The main idea developed during our internship is a mix between a procedural approach and a discrete-space approach. But beforehand we introduce the global framework that drove our work. A Ph.D. student of the team, Luc Leblanc, developed a system for modeling different objects with blocks [LHP11], which provides an easy way to build tree-shaped objects. Without getting into too much details, let us just say that the principle is to assemble simple blocks together, connect them with some additional information, and then run the system to create a polygonal mesh of an arbitrary level of details for the desired object to model.

The two advantages of this system are the adjustable refinement of the output mesh, and the easy correspondence between the shape (skeleton) of a tree and the final surface. It also provides a volumetric representation and a surfacic parametrization. Thus, if we could use a classical L-system to produce the shape of a tree of a given species (note that we do not speak here of the variant from Terraz et al. [TGM+09]), it would be straightforward to create a polygonal mesh for the « shell » of that very tree.

Such was the footing of our work ; by generating textural information on cross-sections of its polygonal model, we seek to devise the internal structure of a tree. From then on, the idea is to compute textures for a few cross-sections in key locations, and then interpolate the obtained information (like cells color) to create a solid texture covering the whole tree. Thence, if we have to cut a branch for example, we can imagine having enough data on the internal composition of the tree to guess where the split is to be made, so that the cut can be rendered properly.

The first step consists in generating a *satisfying* 2D texture for any cross-section of a branch or a trunk. Thus in the next sections we will detail the ins and outs of this first 2D texturing, to eventually tackle the aforementioned interpolation problem.

2 Growth Algorithm

The first step of our texture generation method is to create a texture for any cross-section of wood. Formally we work in a 2D space defined by a cartesian coordinate system $\mathcal{R} = (O, \vec{\imath}, \vec{\jmath})$, local to the desired texture. The outer border of the cross-section may be defined by a set of points s_i given in clockwise or counterclockwise order, but we will see later how it matters. The goal here is to sample enough points of $\mathcal{R}$ to generate an image (a texture) of a targeted resolution.

2.1 A First Procedural Generation

The first method we tried is a simplification inspired from Norell [Nor09], which only attempts to model the *inside* of a tree (not the cracks or other features that can appear when we slice a real trunk). See Figure 3 for a sample result. The advantage is that it is a fast and simple way to create multi-resolution textures, albeit being hardly customizable: for example, adding knots or fitting the texture to an arbitrary shape require some additional reflection.

The gist of the algorithm is to map each point $p = (x, y)$ of $\mathcal{R}$ to a certain distance $D(p)$, describing the distance from the pith of the trunk. If we associate a color c_1 to *earlywood*, and c_2 to *latewood*, then we can map the distance $D(p)$ to a 1D growth pattern, and interpolate the color between c_1 and c_2 according to a certain periodic *pattern* $\Gamma(p) \in [0, 1]$ in the following manner:

$$D(p) = D_e(p) + D_l(p) + D_f(p) + D_n(p) \tag{1}$$

$$\Gamma(p) = \left(\frac{1}{2}\sin(2\pi f D(p)) + \frac{1}{2}\right)^{\frac{1}{\zeta(p)}} \tag{2}$$

where:

- $D_e(p) = \|p\|_2$ is the euclidean distance from p to the origin (pith).
- $D_l(p) = k \cdot p$ where $k \in \mathbb{R}^2$ indicates a preferred growth direction.
- $D_f(p) = \frac{S}{1+\exp(-s(\omega - s_0))}$ where $\omega = D_e(p) + D_l(p)$ and S, s, s_0 are parameters describing the amplitude, slope, and position of the distance change. The idea is to have a pattern whose amplitude varies smoothly when the distance from the origin increases.
- $D_n(p) = \mathcal{N}(0, \sigma)$ is a simple Gaussian blur. We took $\sigma = 0.1$.
- $\Gamma(p)$ is a certain power of $\zeta(p) \in [2, 4]$ to simulate an *earlywood* wider than *latewood.*
- f is the frequency of the pattern. For instance it can be drawn from a uniform distribution $U(0.05, 0.10)$.

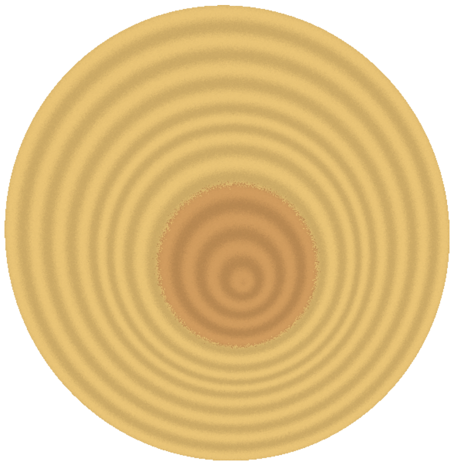
Figure 3: Sample rendered texture using a simplification of Norell's method [Nor09]

The main advantage of this method is its low computational overhead. It is fast and the sampling of a predefined region of space requires no additional simulation. However, we are still limited by the fact that it cannot fit a procedural texture to an arbitrary shape we would like the tree to adopt (imagine for example a tree that would grow around a pole or a fence). If we simply truncate $D(p)$ for the points p that do not fit in our imaginary shape, the result will appear as a section "carved out" of the material. Thus we need another simple, yet more versatile approach: the particle-based method.

2.2 The Particle-based Approach: An Overview

The idea of this method is to store information into particles on the plane — these are called *cells* —, and to simulate their evolution locally. The information stored can be of any type: age and speed of the cell, angular parameter, color interpolated from c_1 and c_2, etc.

Assume we have groups of cells, and at each iteration each cell on the boundary gives birth to a new cell, thus filling space and avoiding obstacles such as knots of dead cells located at arbitrary places: this is the general idea of our algorithm. Now we define more formally the terms used throughout the rest of the document:

Definition 2.1 (Cell)**.** An elementary unit of wood e, represented by its position $p(e)$ in the plane, and associated with additional information (age, angle, speed $v(e)$, etc.). A cell has exactly one *left* and one *right* neighbours, denoted respectively $\mathcal{N}_l(e)$ and $\mathcal{N}_r(e)$. It can have zero, one, or more parents $\mathcal{P}(e)$, as it can have zero or one child $\mathcal{C}(e)$. Together $\mathcal{N}_l, \mathcal{N}_r, \mathcal{P}$, and $\mathcal{C}$ form the neighbourhood of the cell, called $\mathcal{N}(e)$. The relation is symmetrical, meaning that $e_1 \in \mathcal{N}(e_2) \Leftrightarrow e_2 \in \mathcal{N}(e_1)$.

Remark. To simplify notations, from now on we confound the cell e and its position $p(e)$.

Definition 2.2 (Group of cells)**.** A (finite) set of cells bound together. If p is a cell in the group S, then it can be written as $S = \{\mathcal{N}_l^i(p), \mathcal{N}_r^i(p), i \in \mathbb{N}\}$.

Definition 2.3 (Active cells)**.** The cells of the newly bred (or *active*) groups at a certain iteration of the algorithm. The other cells that already have given birth form the *inactive cells*. At each step of the algorithm, the active cells generate new cells according to a set of rules defined hereafter.

Definition 2.4 (Dead cells)**.** A dead cell represents a cell that has been blocked by an obstacle (border of the tree, knot, etc.), and thus that does not generate children (more exactly, it does generate a child, but at the same position where it stands). Remember that a dead cell can belong to an active group. A group whose every cell is dead is called a dead group, and can be safely removed from the set of active groups.

Definition 2.5 (Generation)**.** A set of cells that has been generated at a same iteration of the algorithm. The number of generations is the number of iterations the algorithm has run through.

Definition 2.6 (Skeleton)**.** The graph $G = (V, E)$ formed by all cells and their adjacency relations is the *skeleton* of the resulting wood texture. Two cells p_1 and p_2 are adjacent in the graph iff $p_1 \in \mathcal{N}(p_2)$. We ought to keep this graph planar as we look to build a coherent wood texture. See Figure 4 for an illustration.

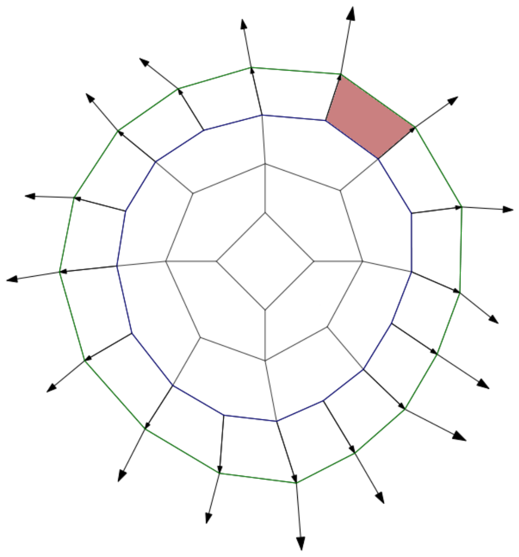

Figure 4: Sample graph (skeleton) with 4 generations of cells. The arrows represent the cell's speed. The outermost polygon (in green) is the current active group. In red, a polygonal *parcel* is defined by 4 adjacent cells.

Remark. By varying the number of generations in an annual ring pattern, we can change the level of details of the resulting texture. Likewise, by changing the speed of the initial generation of cells, we can favour a direction of growth. We may want to have wood grow outward (to fit a particular shape), or inward (for example to draw a small knot in the wood) ; this matter will be discussed later.

2.2.1 Algorithm Description

The basic growth algorithm can be summarised as follows:

Algorithm 1 Growth Simulation

```
Require: An initial generation of cells $p_1, \ldots, p_n$, and a number of iterations $k_{max}$
Ensure: A coherent skeleton $G = (V, E)$
1: for $k = 1$ to $k_{max}$ do
2:    for $S \in$ ActiveGroups do
3:       BreedGroup($S$)
4:    end for
5: end for
```

2.2.2 Image Rendering

To generate an image from a given skeleton of a wood texture, we can use different methods to interpolate color information. A simple one associates each point x in the plane with the color of its nearest neighbouring cell p. A smoother result is obtained with a weighted sum (in the RGB space) of the colors of all neighbours within a radius r. A *kd*-tree structure can be used to efficiently retrieve such neighbouring cells. Though our implementation proved this rendering process to be quite slow in definitive.

Another method splits each polygonal area defined by the skeleton G into triangles, and then renders directly those triangles on the screen using OpenGL. The splitting of a polygon into triangle strips may be done greedily in quadratic time, but also in linearithmic time using a sweep-line algorithm, or even in linear time using more complex routines. We used the naive greedy method in our implementation, as the polygons to split usually contain few vertices. Finally color at each triangle vertex can be bilinearly interpolated directly by OpenGL.

Rendering Process During this internship, we developed two different rendering programs to generate a raster texture from the skeleton representation. The first one is a pure 2D application which uses *Qt* to render an image on screen, while the second program combines *Qt* with the OpenGL framework to render 2D textures in a 3D space, allowing visualisation alongside other 3D objects (loadable `.obj` files). The first program uses a *kd*-tree structure for the rendering process and was a bit slower than its OpenGL counterpart, while the latter showed more clearly visual artifacts. A result comparison of the different rendering method can be seen in Figure 5.

2.3 Heuristics

Function `BreedGroup` defined in Algorithm 1 ensures a coherent skeleton (i.e. without crossing edges). We start creating new groups the following way. Each active cell $p_i^{(k)}$ of generation k gives birth to a new cell $\mathcal{C}(p_i^{(k)}) = p_i^{(k+1)}$ located at the position $p_i^{(k)} + v(p_i^{(k)})$. We connect the newly created cells such that $\mathcal{N}_r(\mathcal{C}(p_i^{(k)})) = \mathcal{C}(\mathcal{N}_r(p_i^{(k)}))$, and idem for $\mathcal{N}_l$. Note that *dead cells* act as cells with null velocity $v(p_i^{(k)}) = 0$ (except they are not subject to modifications of their velocity).

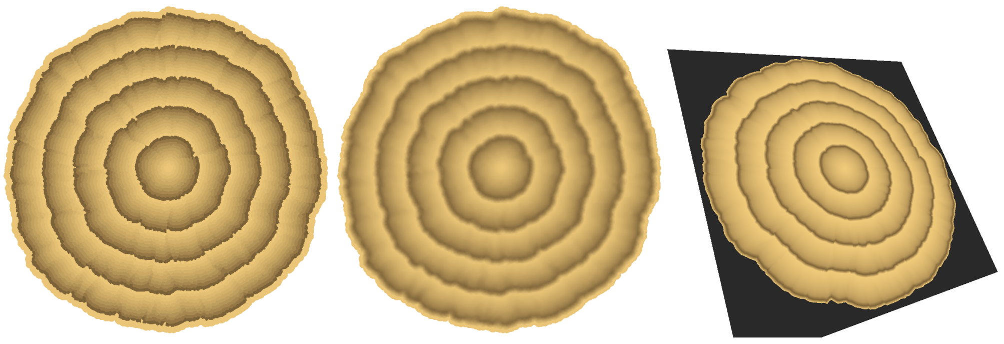
(a) Nearest neighbour (b) Weighted sum (c) OpenGL

Figure 5: Result comparison of 3 different rendering processes.

A random perturbation can be applied to the speed of each new cell, to introduce some variations in the resulting skeleton. We applied an angular perturbation combined with an amplitude perturbation to the new speed, and these perturbations follow a normal distribution whose parameters can be customized.

We assume the cells are given in counterclockwise order. Meaning that for two neighbouring cells p_a and p_b (implying $\mathcal{N}_r(p_a) = p_b$), the previously generated cells are supposed to be located locally in the half-plane on their *right*. This plane is defined by $P_r(p_a, p_b) = \{x \in \mathbb{R}^2, \det(p_b - p_a, x - p_a) > 0\}$.

A set of newly created groups $S_j^{(k+1)}$ will be refined to ensure a certain degree of coherence. For that purpose we use several heuristics described hereafter.

2.3.1 Single Group Verifications

Intermediary cell generation When two new neighbouring cells p_a and p_b are too far from each other, for example separated by a distance $d > d_{sep}$, then we insert new intermediary cells to refine the polygonal mesh. If we simply insert them on the segment $[p_a, p_b]$, it tends to create artifacts such as straight lines instead of a circular ring. An idea is to place an intermediary cell $\tilde{p}$ on a circle $\mathcal{C}$ containing p_a and p_b, whose center is chosen such that each cell's speed is as *radial* to $\mathcal{C}$ as possible. In practice, we intersect each cell's speed direction with the perpendicular bisector of p_a and p_b, then we take the middle for the center of $\mathcal{C}$. See Figure 6 for a geometrical illustration.

To determine the velocity of the intermediary cell, we average the directions of $v(p_a)$ and $v(p_b)$, which yields a new vector whose magnitude is averaged over those of $v(p_a)$ and $v(p_b)$. Then we repeat the process until each pair of neighbouring cells is separated by at most d_{sep}.

Speed readjustment Due to random perturbations and other factors, the speed of a cell p can sometimes drift and point backward past the lines defined with its two neighbours. We detect this situation with the counterclockwise order of the cells mentioned previously,

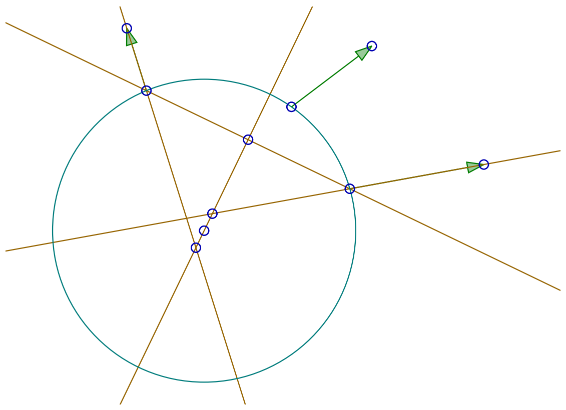

Figure 6: Geometry of the intermediary cell generation process.

and reinterpolate the new speed direction as the bisector of the two lines defined by $(p, \mathcal{N}_l(p))$ and $(p, \mathcal{N}_r(p))$.

Group splitting An important part of the generation algorithm is the management of multiple active groups of cells. We can initiate the simulation with multiple active groups, or split existing groups when they reach a certain configuration during the simulation process. It can happen for example when some cells grow inward, and the shape of the section is that of a ∞ (see Figure 7).

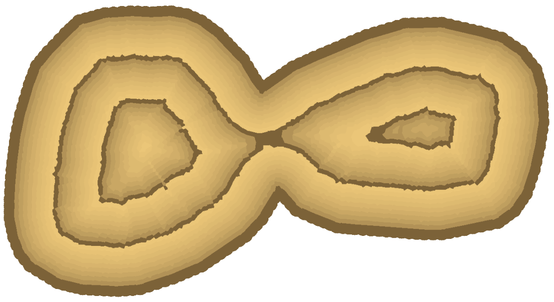

Figure 7: Cells initially placed on the contour grew inward, split and formed two piths.

To detect when it is appropriate to split a group S, we use the following metric: for any two cells p_a and p_b in a group, check whether $2\|p_a - p_b\| < \max_{p \in S} \|p_a - p\|$. If so, then it means the cells p_a and p_b form a sort of strait regarding the span of the whole group S. Then we split S along (p_a, p_b), join the cells properly with their neighbourhood, and readjust their speeds if necessary. If one of the two created groups S_1 or S_2 are too small (in size or in span), we simply delete it.

Cells merging This step is non-essential, but still useful and do not introduce extra overhead in our algorithm. The idea is to merge together cells that are too close to each other or cells that would cross at the next step of the algorithm (i.e. neighbouring cells p_a and p_b such as $[p_a, p_a + v_a]$ intersects $[p_b, p_b + v_b]$). To merge cells p_a and p_b we create a new cell $\tilde{p}$ whose neighbourhood is defined as follows:

- $\mathcal{N}_l(\tilde{p}) = \mathcal{N}_l(p_a)$ and $\mathcal{N}_r(\tilde{p}) = \mathcal{N}_r(p_b)$
- $\mathcal{P}(\tilde{p}) = \mathcal{P}(p_a) \cup \mathcal{P}(p_b)$

Self-intersection suppression The form of an active group is usually far from convex, with many irregularities along the boundary. Thus unexpected flips in the hull can occur for various reasons. It is important to detect when a group self-intersects in order to correct this behaviour. We want the current active groups to be correct, but also their next generation to be consistent.

Thus, for every pair of neighbouring two cells (p_a, p_b) and (p_c, p_d) of the same group S, we check if the quadrilaterals defined by $(p_a, p_a+v_a, p_b+v_b, p_b)$ and $(p_c, p_c+v_c, p_d+v_d, p_d)$ do overlap each other. If they do, then we suppress a whole part of S, and join either p_a to p_d or p_c to p_b. To choose between the two, we keep the part of greater cardinality (to avoid accidentally deleting the whole group). The choice can be done in constant time using a wise set of counters during the traversal.

Finally we also readjust the speed of the newly joined cells, and then repeat the process until no overlapping can be detected, or until the group reaches the critical size of less than three cells. As at least one cell is deleted with each overlap, clearly this algorithm always terminates.

Orientation check As the cells are given in counterclockwise order, they are expected to grow outward if the group's polygon is oriented counterclockwise, and they are expected to grow inward otherwise. An unwanted behaviour occurs when an inward-growing group is split and gives birth to an outward-growing group, because the cells order have been flipped.

Thus for each created group, we check whether its polygonal orientation is consistent with the expected direction of growth (inward or outward). The orientation of a group S can be computed in linear time by looking at the sign of $\sum_{p \in S} \det(p, \mathcal{N}_r(p))$.

2.3.2 Collision between Groups

As stated previously, the algorithm is expected to handle multiple groups of cells that are allowed to grow in parallel. It means that at some point, two groups may eventually cross each other, or encounter obstacles along the way (the external environment is represented by dead groups). Thus we have to check, for any two groups S_1 and S_2, if the polygons of the two groups overlap.

We use bounding boxes to speed up the collision detection. $\mathcal{B}(S)$ is bounding a group S if we have the following property: $S \subseteq \mathcal{B}(S)$, and $\mathcal{B}(S) \cap X = \emptyset \Rightarrow S \cap X = \emptyset$ for any area $X \subseteq \mathbb{R}^2$.

The exact correction process applied when a collision is detected will not be detailed for the sake of conciseness, but it makes use of the expected counterclockwise order of the vertices to detect which sides are already filled and which are not. Thus it can represent obstacles such as knots, but fails to encompass strait and narrow segments, as it can lead to unexpected behaviours. Also, when a cell is detected to be part of a collision, it is marked as *dead* and follows the properties defined earlier for dead cells.

2.3.3 Inter-generation Overlaps

A lately noticed issue still uncovered by the previous heuristics is *inter-generation overlaps*. That is, when creating cells $p_i^{(k)}$ and checking the coherence of the bands made with the hypothetical cells $p_i^{(k+1)}$, we introduce changes in the positions of cells of this k-th generation that can result in intersections with the already in place $(k-1)$-th generation of cells.

Thus, for each pair of cells $p_{i_1}^{(k-1)}$ and $p_{i_2}^{(k-1)}$ with an existing child $\mathcal{C}(p_{i_1}^{(k-1)}) = p_{j_1}^{(k)}$ and $\mathcal{C}(p_{i_2}^{(k-1)}) = p_{j_2}^{(k)}$ that define a *parcel* (see Figure 4), we check if the parcel is a simple or a complex (self-intersecting) polygon. In the second case we change the positions of cells between $p_{j_1}^{(k)}$ and $p_{j_2}^{(k)}$ to turn the parcel into a simple polygon, as if we were stretching a rope on the parts formed with cells from $p_{i_1}^{(k-1)}$ to $p_{i_2}^{(k-1)}$.

2.4 Some Results

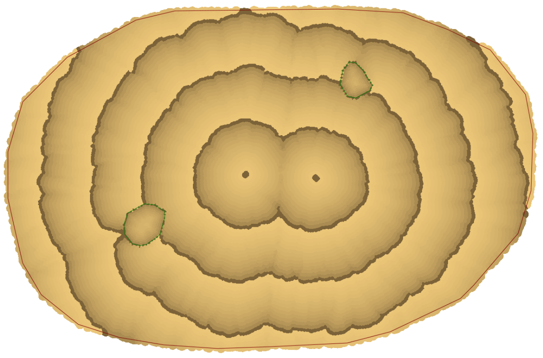

Figure 8: Sample result showing two sources (piths) from which cells grew. A contour blocks the cells' growth. Two knots are modelled with inward-growing groups.

Here we present some of the simulated phenomena. A wood texture can be generated from a tree grown from a single source, or from multiple sources (see Figure 8). A particular border can block growth in further directions (as used in Figure 8), or wood can grow inward, from that border to an origin (see Figure 7).

Knots in the tree can be represented by small inward-growing groups, as shown in Figure 8. Demarcation between knots and other parts of the trunk is not yet clearly visible, but it should be conceivable to add some bark-like separation at this point. Likewise the bark could be outlined on the borders of the generated texture to add to realism, but that was not the point of our work.

2.4.1 Remaining Issues and Future Work

When starting from a straight line, wood cells tend to shape like a curve (see Figure 9). This is good for outward-growing groups, but may lead to unexpected results with inward-growing groups whose polygonal hulls lack a certain level of details. Moreover, with inward-growing groups we cannot control where the pith will appear (it can also have multiple convergence centers, see Figure 7). So inward-growing groups should be used to model knots, and outward-growing ones to populate the shape of a tree with cells emanating from a predefined origin.

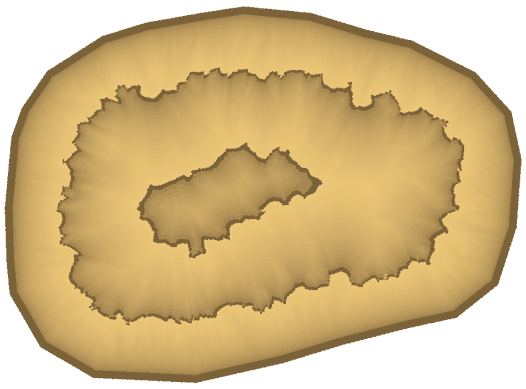

Figure 9: Texture created with an inward-growing group. We can see jagged patterns emerge along the annual growth ring.

Still, as we can see in Figure 8, the result is not what one would expect. The reason is that the simulation algorithm works as if the tree grew into a mold: once a cell reaches the border, it dies and lets its neighbours advance until the whole place is populated. Conversely, when we draw a contour we expect the final tree to grow and to adopt this shape, the growth being captured by several factors such as mechanical forces applied by the wind, or other environmental conditions. Thus once the primary growth is completed, we considered several methods discussed hereafter to correct the skeleton obtained.

Spring-mass system Fix the positions of the dead cells in the skeleton, and let the rest of the graph (skeleton) evolves like a spring-mass system, using forces computed locally. Thus cells that are concentrated around an early-met border would have their positions redistributed along a path to the center of the tree. It sounds nice but does not preserve the *coherence* of the skeleton, and may lead to overlapping elements and crossing edges.

Elements remapping Find a shortest path (according to a certain metric) in G from the center of the tree to each dead cell on the border. The vertices on the paths from the border cells to the center are key vertices that could be remapped uniformly on their respective paths. As with spring-mass systems, it is not easy to ensure that the coherence of the new skeleton is preserved. We would also be compelled to drop the non-key vertices of the skeleton to avoid unnecessary crossing edges.

Biased regrowth Find a shortest path like previously, but store instead a map matching the angular parameter of a border cell with the resulting distance from the pith. That way we could rerun our simulation algorithm, but with an handicap for the cells whose angular parameter indicates that a region would reach a border too rapidly.

2.4.2 Algorithm Overview

The aforementioned algorithm for growing cells to fit a predefined contour has not been implemented in `C++` due to lack of time, but its general idea can be summerized as follows:

Algorithm 2 Contour-driven Growth Simulation

```
Require: Initial cells $p_1, \dots, p_n$, polygonal contour $x_1, \dots, x_m$, and $k_{max}$
Ensure: A coherent skeleton $G = (V, E)$
 1: Run initial simulation with $S_i = \{p_1, \dots, p_n\}$
 2: Build skeleton $G = (V, E)$ accordingly
 3: Let $w(a, b) = \|a - b\|$ for $\{a, b\} \in E$
 4: Let $S_f$ be the dead cells that met the border $x_1, \dots, x_m$.
 5: Find shortest paths from $S_i$ to $S_f$ with respect to $w$ using Dijsktra's algorithm
 6: For each cell $e$ in $S_f$, Dijsktra yielded a distance parameters $d_{border}(e)$
 7: For each cell $e$ in $S_f$, we also had an angular parameter $\theta(e)$
 8: Record them in a sorted map $M : \theta(e) \to d_{border}(e)$
 9: Re-run the simulation with speed magnitudes biased according to $M$ and $k_{max}$
10: Repeat step 2 to 9 until a visually satisfying result is obtained
```

3 3D Interpolation

3.1 Interpolation between two Cross-sections

Suppose we have generated two skeletons corresponding to two cross-sections made at different places in a tree. We need to interpolate the information we have on the two cross-sections to fill the space in-between. That is, we want to know the color, age, etc. of each point between the two sections.

3.1.1 Naive Interpolation Method

The first naive interpolation method that comes to mind works as follows: create two raster images, and use a pixel-based interpolation to produce each intermediary frame — on-the-fly, only when needed. Although very fast, the result happens to be blurry and not at all what we could expect for an interpolated frame.

As the problem can be seen as a morphing between two textures, modern morphing techniques could also be used to achieve a seamless transition between two cross-section textures. However we did not look closely at such techniques, as they are difficult to automatize satisfactorily. It is also of greater benefit to rely directly on skeletons and not on pixelated images produced afterwards.

3.1.2 Proposed Interpolation Method

Suppose both cross-sections were generated using only one group of cells that grew from a predefined origin until it reached the contour of the tree, with k_{max} iterations in both cases. Then the texture interpolation problem can be seen as finding a matching between each cell of the k-th generation from the first skeleton to the second. We can also add dummy vertices to make each generation in both skeletons having the same size.

More formally, we have generated two skeletons $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$, and we are to find a mapping $G : t \to (V(t), E(t))$ such that $G(0) = G_0$, $G(1) = G_1$, and $V(t) = \{x_1(t), \ldots, x_m(t)\}$ where each $p(x_i(t))$ is a continuous function from $[0, 1]$ to $\mathbb{R}^2$.

Let us call $S_l^{(k)} \subseteq V_l$ the cells from the k-th generation in G_l, with $l \in \{0, 1\}$. We assume $|S_0^{(k)}| = |S_1^{(k)}|$ for any k, because otherwise we can add dummy vertices the smallest set to restore balance. Now fix a generation k ; the idea is to find a bijection between $S_0^{(k)} = \{p_i\}$ and $S_1^{(k)} = \{q_j\}$, in order to build $G(t)$ using a simple linear interpolation between matched vertices.

We try to assign each cell to its closest equivalent in the other skeleton, so as to minimize the global displacement that will be performed during the interpolation process. As each cell e is embedded with an angular parameter $\theta(e)$ describing its *drift* in the current generation, our goal will be to minimize the sum of the differences of each matched pair's drifts. We can define a weight function as follows:

$$w(p_i, q_j) = \begin{cases} 0 & \text{if } p_i \text{ or } q_j \text{ is a dummy cell} \\ \min(\alpha_{i,j}, 2\pi - \alpha_{i,j}) & \text{otherwise, where } \alpha_{i,j} = \bmod(\theta(p_i) - \theta(q_j), 2\pi) \end{cases} \tag{3}$$

The weight is null for dummy cells because we do not know initially where to place them (their position will be determined after the matching is found). Our goal is now to find a bijection $f : S_0^{(k)} \to S_1^{(k)}$ which minimizes $\sum_{p_i} \in S_0^{(k)} w(p_i, f(p_i))$, this is the *assignment problem.*

3.2 The Assignment Problem

This is a very classical combinatorial optimization problem, which can be seen as finding a maximum weight matching in a weighted bipartite graph. As an abundant literature can be found on the subject, we will only describe it briefly with a solution outline.

The assignment problem asks to find a bijection f between two sets A and B of size n, which minimizes $\sum_{a \in A} c(a, f(a))$, where $c : A \times B \to \mathbb{R}_+$ is the associated cost function.

This problem is a special case of the minimum-cost flow problem, which is in turn a special case of linear programming. Yet it can be solved in a cubic running time using the *Hungarian method*, also known as the *Kuhn–Munkres algorithm.* The original algorithm was $\mathcal{O}(n^4)$, but has since been reduced to $\mathcal{O}(n^3)$.

The idea of the algorithm is to compute a potential with maximum value that induces a matching of the associated bipartite graph. A *potential* is a function $y : A \cup B \to \mathbb{R}$ such as $y(a) + y(b) \leqslant c(a, b)$ for all $(a, b) \in A \times B$. The algorithm starts with a null potential. A matching M restricted to *tight edges* (edges (a, b) for which the equality $y(a) + y(b) = c(a, b)$ holds) is then computed. When the matching cannot be improved, the potential is updated. Then a new matching is calculated, and so on until M is perfect (i.e. covers every element).

This outline is voluntarily short, as extensive documentation is available on the subject. The unacquainted reader is referred to [Fra05] for further readings on the matter.

3.3 Applications and Limitations

Now we have a feasible algorithm for matching two skeletons of different cross-sections of a tree. Once the matching is established, say $p_i \in S_0^{(k)}$ is matched with $f(p_i) \in S_1^{(k)}$, we have to position dummy cells in the graph so that we can interpolate seamlessly between $S_0^{(k)}$ and $S_1^{(k)}$. Without loss of generality, suppose dummy cells have been added to $S_0^{(k)}$. Then for each neighbouring pair $\{p_{i_1}, p_{i_2}\}$ in G_0, if there are cells between $f(p_{i_1})$ and $f(p_{i_2})$ in $S_1^{(k)}$, place the dummy cells they have been matched with regularly on the segment $[p_{i_1}, p_{i_2}]$.

This yields a method to interpolate two cross-sections, although it has not been tested due to lack of time. Yet at first sight it would not work with Y-shaped parts of a tree, where a trunk divides into two limbs, or when a branch grows off the main trunk. Indeed in such cases the matching is to be done between three or more cross-sections.

Still we can imagine a possible reduction for the problem. In case the trunk G_0 divides into two limbs G_1 and G_2, we could match G_0 with G_1 and G_0 with G_2 independently. When a branch G_2 grows from the main trunk between sections G_0 and G_1, we could match G_0 with G_1, and G_2 with a dummy skeleton $\tilde{G}$ reduced to a single point. As stated previously, new branches have a conic shape, so matching G_2 with a single point makes some sense.

Of course this is only a heuristic, as it does not lift every issue, and there may be many remaining unforeseen artifacts. We consider this to be a first approach, which can be refined later in future work. Moreover, here we thought of a simple linear interpolation for matched cells of the same generation. However if the pith of the tree follows a curve (as it usually does), instead of a simple line, it needs to be taken into account when computing colors of inner points between two cross-sections. We may use local coordinate systems to rotate the plane of the interpolated section accordingly to the curve followed by the pith, but rendering the color of each inner point by an inverted transformation without computing every interpolated cross-section is still challenging.

Conclusion

As we have seen throughout this document, wood texture generation is at a crossroad between biology and computer graphics, and can muster various approaches and techniques to craft proper tree textures and meshes. We presented a purely geometrical procedure to create textures of cross-sections of a tree, as well as a possible interpolation method to retrieve information on the inner parts of a tree.

Still, the current process is not fully satisfying yet, as it can be improved in various ways. Even beyond the biased growth or 3D interpolation algorithm that we described without implementing them, there is room for more research. The aforementioned interpolation along a curve is one example. Computing mechanical forces using the generated interpolated structure is another. We may also want to improve the existing rendering process by combining classical 2D texture coordinates with the constructed internal mesh of a tree.

References

[Buc98] John W. Buchanan. Simulating wood using a voxel approach. *Computer Graphics Forum*, 17(3):105–112, 1998.

[Fra05] András Frank. On Kuhn's Hungarian method — a tribute from Hungary. *Naval Research Logistics (NRL)*, 52(1):2–5, 2005.

[LHP11] Luc Leblanc, Jocelyn Houle, and Pierre Poulin. Modeling with blocks. *The Visual Computer (Proc. Computer Graphics International 2011)*, 27(6-8):555–563, June 2011.

[Lin68] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, 1968.

[Nor09] Kristin Norell. Creating synthetic log end face images. In *Image and Signal Processing and Analysis, 2009. ISPA 2009. Proceedings of 6th International Symposium on*, pages 353–358, Sept. 2009.

[Pea85] Darwyn R. Peachey. Solid texturing of complex surfaces. *SIGGRAPH Computer Graphics*, 19(3):279–286, July 1985.

[Per85] Ken Perlin. An image synthesizer. *SIGGRAPH Computer Graphics*, 19:287–296, July 1985.

[PL96] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants.* Springer, 1996.

[TGM$^+$09] Olivier Terraz, Guillaume Guimberteau, Stéphane Mérillou, Dimitri Plemenos, and Djamchid Ghazanfarpour. 3Gmap L-systems: an application to the modelling of wood. *The Visual Computer*, 25(2):165–180, 2009.