The Stochastic Modelling of Trees

Alain Fournier
David A. Grindal

Computer Systems Research Institute
Department of Computer Science
University of Toronto
Toronto, Ontario
M5S 1A4

ABSTRACT

We present here a fast method for the modelling of trees which brings together two interesting techniques. The trees are modelled as convex polyhedra for the description of the gross. shape, and three-dimensional texture mapping is used for the detailed features.

The "essential" volume of the tree is represented as the convex intersection of half spaces. The advantage of this representation is that it allows an adaptive level of detail in the display. We use a special algorithm for the display of the convex intersection which computes it directly in the frame buffer. The algorithm also allows the computation of intersecting polyhedra.

To transform the convex polyhedra into a more realistic representation of trees, we use three-dimensional texture mapping to "modulate" the shape and the colour of the basic polyhedra. We then obtain an irregular non convex object, which is consistent in shape and general appearance regardless of the point of view and the size on screen. Three dimensional fractional Brownian motion is one of the procedural texture used.

KEYWORDS: tree modelling, half space intersection, stochastic modelling, frame buffer algorithms, adaptive modelling.

RESUME

Nous prèsentons ici une méthode rapide pour le modelage des arbres qui réunit deux techniques intèressantes. Les arbres sont modelés par des polyhèdres convexes pour la représentation de la forme globale, et une texture à trois dimension est utilisée pour modeler les détails.

La forme "essentielle" de l'arbre est réalisée par dès polyhèdres convexes, résultats du calcul de l'intersection de demi-espaces. L'avantage de cette reprèsentation est qu'elle permet un niveau adaptif de détail. Nous avons développé un algorithme pour le calcul de l'intersection convexe directement dans la mémoire d'image. Avec une légère modification l'algorithme permet le calcul de polyhédres qui s'intersectent.

Nous transformons les polyhèdres en une reprèsentation plus réaliste des arbres en utilisant le "mapping" d'une texture à trois dimension pour moduler la forme et la couleur des polyhèdres de base. Nous obtenons ainsi un object non-convexe et irrégulier, dont la forme et l'apparence génèrale est consistante indépendemment du point de vue et de la taille de l'arbre sur l'écran. Le mouvement Brownien fractionel à trois dimensions est une des procédure de génèration de texture utilisées.

MOTS CLES: modelage d'arbres, modelage stochastique, intersection de demi-espaces, algorithmes de mémoire d'image, modelage adaptif.

1. Motivations

Trees are obviously very important in the modelling of natural scenes and landscapes. Problems are caused by the large number of trees needed and their considerable variety of shapes. The main criteria for a good model are to be realistic, easy to compute (both in terms of the basic operations needed and of the time complexity), flexible (capable of generating the intra- and inter-species variations in shape), adaptive (generating various level of details as needed) and compact. Of course, depending on the application, one or more of these criteria can be relaxed if not all can be met. The techniques used so far include grammar generation systems [AoKu84, Smit84], particle systems [ReBl85], polygonal description plus two-dimensional texture mapping [Bloo85] and simple volume primitives plus two-dimensional texture mapping [Gard84, Gard85]. The technique we will describe here, which is close in spirit to the ones used by Gardner, uses simple volume primitives (convex polyhedra) computed in the frame buffer associated with stochastic three-dimensional texture mapping. Table 1 gathers a subjective evaluation of these different techniques with regard to the above criteria. A scale of 0 (not at all) to 5 (best possible) is used.

Refs.	Real.	Easy (ops)	Easy (time)	Flex.	Adapt.	Compact
[AoKu84]	4	2	2	4.5	3	4
[ReBl85]	4.5	2	2	3.5	2	4
[Bloo85]	4.5	2	2.5	1	1	1
[Gard84]	3.5	2	3	2	3	4
Here	2	4	4.5	3	4	4

Table 1. Subjective comparison of tree models

To achieve flexibility, we will use a mixture of generative techniques, as in [AoKu84, Smit84] and stochastic techniques, as in [FoFC82, Reev83, ReBl85]. The goal of compactness will therefore be achieved, since the actual description for each tree is very small. We will have to pay special attention to the problems of *consistency*, that is keeping the appearance constant as the level of detail, the point of view and the size on screen of the displayed objects are changed.

It is highly desirable that the entire process of generating, rendering, and colouring the tree(s) be done in a reasonable time and with moderate amounts of computing power. Since our goal here is not *ultra-realism* but a balance between realism and time, we hope to be able to approach the conditions for real-time display. The system described here will not create images in real-time. However, it should be possible, with hardware and minor software improvements, to bring it close to or achieve real-time performance.

2. The three-dimensional shape

Primitives to model three-dimensional objects range from points to lines to polygons to higher degree surfaces. Most of these have been used to model trees. Polygons, because they are linear objects, and because most rendering systems ultimately deal with polygons at the display level, are a tempting choice. They have many drawbacks, however. Many polygons are needed to represent a complex shape, such as of a tree, and they constitute a very inflexible model, hard to parametrise or modify adaptively. There is another representation scheme which has most of the qualities of polygonal models and some additional advantages. The volume of the tree can be represented as the *convex intersection of half-spaces*. A *half-space* is the area of space all on one side of a plane. Formally, a half-space HS_i is the locus of points (x,y,z) such that $a_i x + b_i y + c_i z + d_i \geq 0$. If several half-spaces are intersected, the result is $V = \bigcap_i HS_i$ or $V = \left\{ (x,y,z) \in \mathbf{R}^3 : \forall i \; a_i x + b_i y + c_i z + d_i \geq 0 \right\}$. This volume, usually enclosed, is convex, and its faces are all convex polygons.

This form of representation is rather different from any conventional means of storing three dimensional polyhedra. The most radical departure from the norm is that it does not store the vertices of the polyhedra. The only entities stored are the equations of the intersecting planes.

One benefit of storing planes is that there is added information stored in the equation. For the plane ax + by + cz + d = 0 the vector (a,b,c) is the normal to the plane. This fact will be used later, for the generation of the trees. It turns out that by using the normals, a user can create a wide range of trees easily and quickly. If the same normals and the same parameters are used, the procedure can also consistently generate the same tree.

Another advantage of the half space representation is its flexibility. When the object is defined by a set of half spaces, it is possible to get a finer representation by splitting the planes. This splitting can be done to any one plane, without greatly affecting the total volume or overall shape. With a polygon mesh it is a difficult process, because the criteria to merge and split polygons are not obvious, and a change can affect many polygon boundaries.

This scheme has another (minor) advantage over the polygon mesh, in that the amount of storage needed for the same polyhedra is a little less.

2.1. Generating the Tree

Using the normals (*ie* planes) to generate the trees gives more freedom in generating trees randomly. Many schemes could be thought of for splitting normals, in order to create a convex hull. In fact any grammar can drive the process. We will only describe one method here.

The principle is illustrated by Figure 1. The existing normal $\vec{N}_1$ defines the current plane P_1. The normal $\vec{N}_1$ will be split into $\vec{N}_2$ and $\vec{N}_3$. which will define the planes P_2 and P_3. It is desirable that the area, or volume, described by P_2 and P_3 be approximately that described by P_1. There should be some "natural" breakdown of the normals so that the end result after several splits, is roughly the same as the original plane.

In addition to the manner in which a given plane is split in two, there is the further choice of which plane is to be split. There are many possible rules which could be followed here. The "oldest" plane could be split each time. A "lifetime" could be assigned to each plane, with a probabilistic chance of it being split when its life is over (the most likely probablily here would be a negative exponential), or planes could simply be chosen at random. In the system described here, the planes were split in generations. All the planes were split at each stage. Thus all the resulting planes are of the same "age", and there are always 2^n*N of them, where n is the number of generations and N is the number of initial planes. This is equivalent to applying the production rules of a parallel grammar at each generation.

The equation that governs the splitting of the normals can be read off of Figure 1. The normals should be split so that in the average case,

$$l_2 \cos\alpha_1 = l_3 \cos\alpha_2 = |BC|.$$

Since this equality only holds in the average case,

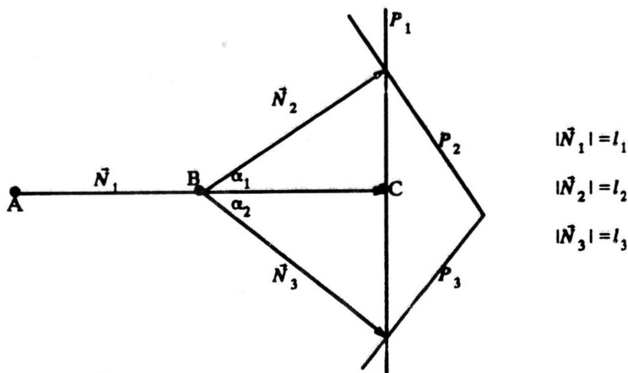

Figure 1 Splitting the Normal

there will be some random perturbation around the exact values. Even with this equation restricting the splitting method, there are still a great number of parameters to control. The following algorithm was used:

Step 1) Choose point B. This point will be the origin for the two new normals, $\vec{N}_2$ and $\vec{N}_3$. The parameters here are μ_B and σ_C Point B will be chosen a distance down the normal from C: $|BC| = l_1 * \mu_B + gauss() * \sigma_B$.

Step 2) Choose the angles at which the new normals will split from the present normal:

$$\alpha_1 = \mu_\alpha + gauss()*\sigma_\alpha$$

$$\alpha_2 = \mu_\alpha + gauss()*\sigma_\alpha$$

Step 3) Chose the length of the two new normals:

$$l_2 = |BC|/\cos\alpha_1 + gauss()*\sigma_l*l_1$$

$$l_3 = |BC|/\cos\alpha_1 + gauss()*\sigma_l*l_1$$

The lengths are designed so that the end of the normal is in the plane P_1.

Step 4) Reduce the angle at which the new normals are created:

$$\mu_\alpha = \mu_\alpha * ratio$$

$$\sigma_\alpha = \sigma_\alpha * ratio$$

This maintains the user's control over the creation process. If the splitting angle were not reduced, then the normals resulting after two or three levels of recursion would have no resemblance to the original.

Since the representation being used is that of convex intersection, it is possible for one errant plane to chop the tree in half. This occurs if a normal is split far enough away from its predecessor's original direction. The problem is roughly similar to that of self-intersection in two dimensional stochastic interpolation. Figure 2 demonstrates how this can happen if the normals split in just the wrong way. In fact, the problem occurs more often if the splitting is taking place in three dimensions (as is being done) instead of two dimensions (as is being shown). In the diagram the seven "outside" normals have been shortened for sake of clarity.

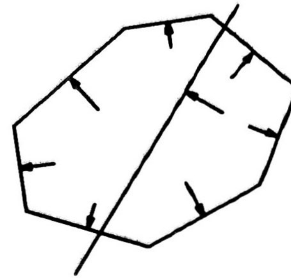

Figure 2 The Effect of One Errant Normal

In order to prevent this from occurring, one more restriction is added to the creation process. The procedure keeps only the normals that point "outwards". The algorithm ensures that if a normal's direction is into a certain octant, that the origin of the normal is also in that octant. If the normal is (a,b,c) and its point of origin is (x,y,z), then the normal is retained if and only if

$$ax \geq 0 \text{ AND } by \geq 0 \text{ AND } cz \geq 0.$$

This process of pruning the normals is demonstrated in two dimensions by Figure 3. In this diagram, normals $\vec{a}$, $\vec{b}$, and $\vec{c}$ would be retained, where $\vec{d}$ and $\vec{e}$ would be rejected. Using this pruning method it can be seen that an occurrence such as that in Figure 2 is not possible. This means that the convex hulls should be fairly well proportioned. One "bad" normal can not cut away half of the volume.

The creation procedure lets the user define any number of normals to start. Empirically, it turns out that beginning with three to six normals gives the best results. This process gives a large amount of control over the result. If the input included a long vector, the result was usually a long thin tree. The input angles are additional parameters which permit wide control of the overall shape. In fact the sample space is large enough that it has not yet been fully explored.

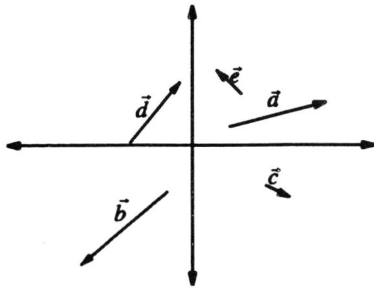

Figure 3 Example of Pruning the Normals

2.2. The Half-Space Intersection Algorithm

We now have a collection of planes to model the tree. What is needed is a visible-surface algorithm for the intersection of half-spaces. The problem of finding the convex intersection of half spaces has been explored by Brown, among others [Brow79] and is $\Theta(N)$. Although his method was not a visible surface algorithm, it could be adapted to this purpose. However, Brown treated the problem as one of geometry, not of graphics, and his solution is in *world space*. A visible surface algorithm for convex intersection that uses the frame buffer was presented in [FoFu86]. It is similar in many ways to the standard Z-buffer algorithm used for many polygon based systems. In the terms of [FoFu86], each pixel needs two registers. With a large frame buffer, providing enough bits for two registers is not too difficult as long as the stored values can be bounded.

At the beginning of the algorithm, in Pass 0, the value of *current_back* is set to the farthest possible value. This represents the background depth. The back-facing planes are scanned out first. If a plane is in front of the current farthest-forward back-facing plane, then that depth is stored for that pixel. For the sake of clarity the equation for z was used in the description of the algorithm, but in practice the the depth value is calculated incrementally at each pixel. Thus the calculation costs only one addition for each point.

The same process is followed for the front-facing planes. Each is scanned out incrementally and at each pixel the depth is compared to the current depth. If this plane is further back than the old one, then it becomes the current depth. However, if the plane is behind the most forward back-facing plane, then that pixel is not in the convex intersection. This is indicated by placing the same depth in both the current front and back registers. All the points at which this occurs are then set to the background colour in a quick Pass 3. Note that this third pass only scans the screen once, as did Pass 0.

The above procedure leaves on the screen the depth values for each visible point of the convex intersection. A fourth pass coloured the polyhedron for the purpose of Figure 4.

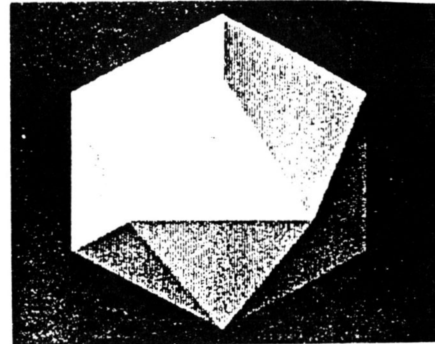

Figure 4 Example of Convex Intersection

There is one serious problem with the algorithm as defined: each plane must be scanned out across the entire screen. One can easily assume frame buffer hardware that accomplishes that in constant time. In fact this is very close to the algorithms used in *Pixel-Planes* [FGHS85]. As the current system was implemented with a general purpose graphics processor, this could be a limit on the performance of the algorithm. A way to avoid this extra work is evident from classic graphics algorithms. The polyhedron has some maximum and minimum x and y values on the screen. Simply "box" the polyhedron and only scan out the planes inside the box. Boxing the solid, however, leads to a new problem. The box is not quickly determined from a set of plane equations. The solution we adopted is to create the box dynamically. The first plane or two will be scanned out normally. By the third or fourth plane, there will be scanlines on which

Pass 0

For all pixels
 current_back = MAXDEPTH

Pass 1

For each back-facing plane ($c>0$)
 $z = -\frac{a}{c}x - \frac{b}{c}y - \frac{d}{c}$
 if $z <$ *current_back* then
 current_back = z

Pass 2

For each front-facing plane ($c<0$)
 $z = \frac{a}{c}x - \frac{b}{c}y - \frac{d}{c}$
 if $z >$ *current_front* then
 if $z <$ *current_back* then
 current_front = z
 else
 current_front = *current_back*

Pass 3

For all pixels
 if *current_back* = *current_front* then
 Colour = Background-colour

no part of the convex hull can possibly be. For example a back-facing plane could have cut in to a depth less than zero (i.e. behind the screen). In practice this eliminates a great many scanlines from consideration. The same process applies vertically. If the equations of the original planes are retained, then some beginning box can be computed from these. Since the number of initial planes is small (from 3 to 6) this is easy, and it has only to be done once. Then as the program runs, the box will be shrunk dynamically. The combination of the two boxing methods is quite efficient.

The half-space intersection algorithm then will take the output from the creation program to give a visible surface and depth values. The algorithm from [FoFu86] can be generalized to work on several convex hulls during the same run. The generalization only requires another register. This gives a total of three registers, which causes a problem for most frame buffers. As the entity stored represents depth, three registers in a 24-bit frame buffer means only 256 units of depth per register. This is not a great deal of room to work with. But it is only a temporary hardware limitation. We expect most future frame buffers to be more generous in bits/pixels. In fact there are already some with 48 bits, like the *Pixar* [LePo84].

The multiple convex intersection algorithm works very much like the single. The polyhedra are processed individually. This takes up the same two registers as before for *current_back* and *current_front*. The difference is that after a polyhedron is finished, it is then merged with those already scanned out. At each point, the depth of the just created surface is compared to the depth of the surface already there, if any. The surface closer to the viewer is kept in the third register. It should be noted that this algorithm not only allows multiple convex hulls, but that the hulls may actually intersect each other and the correct result will be obtained.

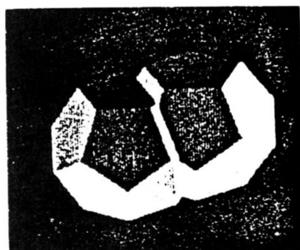

Figure 5 Example of Multiple Convex Intersection

The algorithms described so far results in an adaptive convex polyhedral shape to be written into the frame buffer for each tree. To make this shape more realistic, several methods can be used. One is to use stochastic interpolation [PiFo84, FoMi85] to "roughen" the hull by adding stochastic variations to the depth of the visible faces. That will create rough (non-convex) edges, and possibly holes in the shape of the tree [Grin84]. On other is to use three-dimensional texture mapping. This is the technique that will be described in the next section, but it should be noted that they can and have been used concurrently.

3. Three-dimensional Texture Mapping

Texture mapping in two-dimension is a simple and powerful idea that has a long history in computer graphics [Catm74, BlNe76, Blin78]. More recently the idea was generalized to three dimensional texture [Perl85, Peac85]†. What is needed is a texture solid, and a method to map it to the screen. The texture solid can be created by any process desired, as in the two dimensional case for the texture tile. The cube can be pre-computed, run-time computed, hand-drawn, or digitized from a real image. The creation is a process separate from the mapping. The mapping itself is simple in principle. The face of the object to be mapped has a set of coordinate values for its position. At each point on the object face, the (x,y,z) coordinate values are mapped into the (i,j,k) values of the texture cube.

One problem inherent to the idea of a three dimensional texture map is the sheer amount of storage necessary to hold the texture cube. One solution to the problem is to recognize that the frame buffer itself is a large block of memory. Assume a 32-bit frame buffer, not unreasonable by today's standards. This means that 32 bits of information are needed at each point in the cube. If the texture cube is stored in the top eight bits of each pixel, then four screen pixels store one texture pixel. Thus a 32 X 32 X 32 bit texture cube would take up $(2^5)^3 * 2^2 = 2^{17}$ screen pixels. A 512 X 512 frame buffer contains 2^{18} points. It can be seen that even a sizable texture cube stored only in the top bits will easily fit into the frame buffer. By taking only the top eight bits, the lower 24 are left. Thus, the normal red, green and blue planes are untouched.

A second difficulty with three dimensional texture mapping is that of aliasing. This problem occurs, as it does in the two dimensional case, when a large scale difference between the texture cube and the object being mapped causes sampling problems. Solutions used in two-dimensional texture mapping can be applied here too. In particular the MIP map technique [Will83] directly translates to three dimensions. As in the two dimensional case the texture tile is repeatedly replicated at half resolution. Initially the texture cube takes up half of 512x512x8 bit buffer. If the cube is averaged into a cube half its length per side, it will only be one eighth of the size of the original. This process can be repeated and the eventual result will not even fill the buffer. This form of pre-computed averaging is a viable solution for at least some of the aliasing problems.

† The work described here was completed before these papers appeared, and thus our use of three-dimensional texture was developed independently. See [Grin84].

The fact that only the top of the frame buffer is used by the texture cube, has an important meaning. If the rest of the processing does not require the upper eight bits, then the texture cube can be pre-processed and read in before beginning the rest of the work. This means a considerable savings in run-time. Unfortunately, in this implementation the frame buffer contained only 24 bits per pixel. Since the half-space intersection algorithm needed 24 bits, this meant that the texture cube had to be read in only when it was to be used. This is yet another incentive to get as many bits in a frame buffer as you can afford. You will always find uses for them.

This method of storing the cube leads to a simple mapping function. Assume that a 2^n element-per-side cube is stored in a word-addressable 512 X 512 frame buffer. If the screen address is (x,y) with a depth value of z, then the mapping is a simple

$$addr = (x + y*2^n + z*2^{2n})*4.$$

In other words, the texture cube is treated as a large three-dimensional array. To find the exact (i,j) position in the frame buffer, the result above is split bitwise. The lower 9 bits are the i position; the upper 8 bits are the j position. The multiplication by 4 is because 4 screen pixels store one texture point. Thus the mapping takes only three shifts and two additions per point.

The inputs (x,y,z) to the mapping function above must be contained in the cube. That is, with the above assumptions, $0 \leq x,y,z < 2^n$. To achieve this all that needs to be done is take the original (x,y,z) values modulo 2^n. This is equivalent to creating a large enough texture cube by repeating the smaller one over and over. It should be noted that because of the nature of the three dimensional cube, it is unlikely that there will be some undesirable macroscopic pattern created by this repetition, as often occurs in the two dimensional case. This is so, because in the two dimensional case, the same picture is repeated exactly. With a texture cube, this can only occur if the surface is at the same angle and position across several cubes, which is less likely to happen.

4. Mapping the Texture to the Polyhedra

In effect three-dimensional texture mapping allows the faces of the polyhedra we have defined previously to determine the boundaries of the tree in the *texture space*. The three dimensional texture cube can be generated by randomly placing small "chunks" of colour in three-space. The colours are chosen by the user, as well as the number of chunks and the percentage of each colour. It can also be generated using three-dimensional fractional Brownian motion [MaVN68, FoFC82], or other suitable procedural texture.

Each pixel which displays a part of the tree, contains three coordinates: the (x,y) position on the screen and the z value in the frame buffer. Each of these points is put through the inverse of the transformation applied to the objects to give the (x,y,z) real-world coordinates which will then be used as indices into the texture cube as described above. This process give the consistency of colour desired and is also done with reasonable speed. The important point from the point of view of efficiency is that the mapping can be done incrementally.

The colouring of the tree is done in one pass through the screen. Each point is put through the inverse transform, and then mapped into the texture cube. At the top of the screen a current position in world space, (x_c,y_c,z_c) is caculated. This is obtained by putting the first point through the transform. Let the transformation be $\mathbf{M}:\mathbf{R}^3 \rightarrow \mathbf{R}^3$ Then for some Δx, since $\mathbf{M}$ is linear,

$$\mathbf{M}(x+\Delta x,y,z) = \mathbf{M}(x,y,z) + \mathbf{M}(\Delta x,0,0) = (x_c,y_c,z_c) + \Delta\mathbf{M}_x$$

$\Delta\mathbf{M}_x$ is a constant for a constant Δx. This, of course, generalizes to $\Delta\mathbf{M}_y$ and $\Delta\mathbf{M}_z$. If the depth value changes non-linearly in the frame buffer, as it would if the tree has been stochastically "roughened", then an increment for a changing z value is needed. Again, the linearity of $\mathbf{M}$ allows an incremental computation of $\mathbf{M}(x+\Delta x,y,z+\Delta z)$.

With this use of the three dimensional texture mapping, the tree has been coloured, with the ability to both reproduce the shading in place, and shade it correctly as the viewpoint moves. This all was accomplished with reasonable speed.

5. Adding the Trunk

Now that the crown of the tree has been shaped and shaded, the trunk of the tree is to be added. Part of the information stored during the processing of the three dimensional crown is the position of the centre of the tree. This is usually the base of one of the plane normals generated or given in the creation of the tree by splitting normals. After the centre position is put through its transforms, the resulting depth value lets a perspective mapping be done which scales the trunk to a size that fits the rest of the tree. This perspective mapping is a standard transform.

To shade the trunk, a modified version of Blinn's wrinkled surface technique was applied [Blin78]. The trunk is given a base colour, usually some dark brownish-red. Then ranges are given for each of the component colours (red, green, blue). A random amount within that range is added to the base colour at each point. For example, if the base colour is (60,30,10) and the ranges are (40,20,10), then the colour at each point of the trunk would be an r,g,b triple with red $\in(60,100)$, green $\in(30,50)$, and blue $\in(10,20)$. Values are uniformally distributed within these ranges. When the parameters were chosen well, this scheme gave a very acceptable simulation of tree bark. This method also lets different kinds of trees be modelled properly. Poplars, for example, have a smooth, light-green coloured bark, oaks a rough brown bark.

There remains to determine the visibility between crowns and trunks. On possibility is to use a reverse

painter's algorithm. The trunks are painted after all the crowns, from front to back, and they are not painted over anything already there. If the point of view is from above, such as every crown has priority over every trunk, this will give the correct priority.

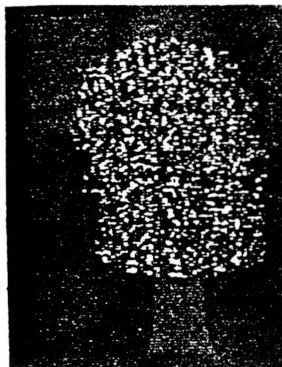

Figure 6 Completed Three Dimensional Tree

A more general method is to model the trunks as convex polyhedra, and use the algorithms applied to the crowns. The trunk can be described as a hexagonal cylinder, or cone, rendered with the half space intersection algorithm, and coloured as before. This approach gives an exact solution to the visibility problem, but adds seven or eight planes to scan out for each tree. It is not a large additional burden, especially since boxing is easier and more efficient given the shape of the trunk. It should be mentioned here that in our context we do not worry about modelling branches.

6. Implementation issues

We will describe in this section how the system was implemented, give numbers to indicate the system performance, and discuss ways in which this performance can be improved.

6.1. Implementation Description

The work of the system is split between two machines. The mainframe is a PDP VAX 11/780 running UNIX†. The other machine is an ADAGE RDS-3000 Graphics Processor and Raster Display System. This is a modular system with its own bus and it is interfaced to the VAX. The ADAGE bus is synchronous with a 32-bit data path. The basic cycle time is 200ns. The frame buffer is 512 by 512 pixels, each with 24 bits. It can also be organized in a 1K by 1K mode with 6 bits per pixel. Much of the power of the ADAGE comes from the use of the 200ns cycle, 32 bit, bit-slice processor. The processor is supported by a 4K by 64-bit wide microcode memory and an 8K by 32-bit wide scratchpad memory. The processor also includes a 16 X 16 bit hardware multiplier which does a signed multiplication in two cycles (400ns). The code for the graphics processor was written in a C-like language for a compiler developed at the University of North Carolina [Bish82]. While allowing only integer arithmetic, this language was of immeasurable help to the implementation.

† UNIX is a trademark of Bell Laboratories

Almost all of the actual processing work for the tree creation system implemented was done on the ADAGE bit-slice processor (herein called simply simply the Adage). The VAX processor was used only as a driver, loading microcode into the Adage and starting the routines, and to perform the basic geometric operations of splitting the normals.

In some parts of the system it was necessary to do non-integer arithmetic on the Adage. The best example of this was the convex intersection routine. A series of fixed point routines (one 16 bit word for the integer part and one 16 bit word for the fraction) were implemented. In addition to needing non-integer arithmetic, several of the Adage routines needed random, or at least pseudo-random, numbers. We used a multiplicative congruential routine to generate the pseudo-random numbers. To ensure that the routine did not loop, a new random seed was used every 512 iterations. This method did not consume excessively large amounts of time to feed seeds down to the Adage but did generate satisfactory random numbers for the Adage routines.

6.2. System performance

Detailed timimg information can be found in [Grin84]. For the icosahedron of Figure 4, with an initial box of 512x512, the rendering takes roughly 12 seconds. These assumptions give a time of approximately 50μsec per pixel, or about .7 to .8 seconds per plane.

The roughening step, if applied takes about 1.0 second for a 250x250 pixel object. The texture generation, takes also about one second, but again this is a preprocessing step if sufficient storage is available for the texture.

The other important factor is the time to load each separate program in the processor, when the microstore is not big enough, which was the case in our system. This is also dependent on the load on the VAX and can take several seconds.

6.3. Possible Speed Improvements

At present the tree creation system is several orders of magnitude away from being real-time. The key to improve the performance is in a combination of specialized processors and a suitable multiprocessor architecture.

Specialized processors already exist for the type of operations used in the system. For the creation of the planes by normal splitting, most of the operations are floating point operations, with calls to a normal distribution function, and to trigonometric functions. The functions can be replaced by lookup tables. In this case, each splitting operation takes less than 20

floating point operations and/or lookup steps. The number of splits necessary depends on the number of trees, and their size on screen. It also depends on how many different trees the system uses. There can be many trees on the picture sharing the same convex polyhedron. To take a numerical example, assume a 512x512 display, 200 trees, each on the average 20x20 pixels, and covering 1/4 of the screen, that is an average depth complexity (for the trees only) of 1.22. Further assume that a tree on the average goes from 6 planes (in the initial master) to 12 on the picture, that is needs 6 plane splitting operations. At 60 frames/second, that means 1.4 MFLOPS for the processor in charge of the splitting. This is easily achievable on a custom VLSI.

The second step, and the main bottle-neck in the current system, is the computation of the convex intersection. As mentioned before, an architecture such as used in the Pixel-planes is suitable for the basic operations used in this step. Making the assumptions in [FGHS85], that is a 10Mhz clock, and reasonable values for the number of bits in the plane equations, we obtain about 60 clock cycles per plane scanned out, that is each plane is scanned out in 6 μs. The trees can then be scanned out in 14 ms, which is fast enough. Note that this is independent of the size of the trees.

The stochastic values needed for the roughening step and the three-dimensional texture generation can be supplied by a processor like the *STINT* [PiFo84]. The current implementation of the STINT generates two-dimensional texture, and can only generate a 70×70 texture in real time, but most of the textures needed can be precomputed. It also should be noted that specialized hardware for real time texture mapping is already in use in flight simulators such as Evans & Sutherland CT6 or General Electric Compuscene.

Remains to organize these processors into a suitable display architecture. This is a complex task, especially since there are other parts of the display system to consider (terrain, buildings, moving vehicles, atmospheric effects, etc.). This is left, as they say, to further research.

7. Conclusions

Within the stated limits: reasonably realistic trees, simple operations, adaptability and flexibility, we feel that the techniques described here succeeded fairly well. One interesting lesson is also that the system distinguishes clearly between the modelling of the shape, which is done with the implicit intersection of half-spaces, and the rendering method, which is the combination of a frame buffer algorithm and three-dimensional texture mapping.

We saw also that the simplicity of the operations and their modularity led to the conclusion that with suitable specialized processors, the real-time generation and display of several hundreds such trees is possible.

Acknowledgements

We gratefully acknowledge the support of the Natural Sciences and Engineering Research Council of Canada. Alain Fournier also wants to thank the Department of Computer Science of Stanford University for its hospitality and the use of its facilities to write this paper.

References

[AoKu84] Aono, M. and Kunii, T. L., "Botanical Tree Image Generation", IEEE Computer Graphics and Applications, **4**, **5**, (May 1984), 10-34.

[Bish82] Bishop, G., *Gary's Ikonas Assembler Version 2 Differences Between Gia2 and C*, Technical Report, University of Northern Carolina, 1982.

[BlNe76] Blinn, J., and M. E. Newell, "Texture and Reflection in Computer Generated Images", Communications of the ACM, **19**, **10**, (Oct. 1976), 542-547.

[Blin78] Blinn, J. F., "Simulation of Wrinkled Surfaces", in *Proceedings of SIGGRAPH'78*, also published as Computer Graphics, **12**, 3, (Aug. 1978), 286-292.

[Bloo85] Bloomenthal, J., "Modeling the Mighty Maple", in *Proceedings of SIGGRAPH'85*, also published as Computer Graphics, **19**, 3, (July 1985), 305-311..

[Brow79] Brown, K. Q., *Geometric Transforms for Fast Geometric Algorithms*, PhD. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1979.

[Catm74] Catmull, E., *A Subdivsion Algorithm for Computer Display of Curved Surfaces*, University of Utah Computer Science Dept., UTEC-CSc-74-133, (Dec. 1974).

[FGHS85] Fuchs, H., Goldfeather, J., Hultquist, J. P., Spach, S., Austin, J. D., Brooks, F. P., Eyles, J. G. and Poulton, J., "Fast Spheres, Shadows, Textures, Transparencies and Image Enhancements in Pixel-planes", in *Proceedings of SIGGRAPH'85*, also published as Computer Graphics, **19**, 3, (July 1985), 111-120.

[FoFC82] Fournier, A., Fussell, D. and Carpenter, L. "Computer Rendering of Stochastic Models", Comm. ACM, **25**, 6, (June 1982), 371-384.

[FoFu86] Fournier, A. and Fussell, D., "On the Power of the Frame Buffer", to appear in ACM Transáctions on Graphics.

[FoMi85] Fournier, A. and Milligan, T., "Frame Buffer Algorithms for Stochastic Models", IEEE Computer Graphics and Applications, **5**, 10, (October 1985), 40-46.

[Gard84] Gardner, G. Y., "Simulation of Natural Scenes Using Textured Quadric Surfaces",

in *Proceedings of SIGGRAPH'84*, also published as Computer Graphics, **18**, 3, (July 1984), 11-20.

[Gard85] Gardner, G. Y., "Visual Simulation of Clouds", in *Proceedings of SIGGRAPH'85*, also published as Computer Graphics, **19**, 3, (July 1985), 297-303.

[Grin84] Grindal, D. A., *The Stochastic Creation of Tree Images*", Master Thesis, Department of Computer Science, University of Toronto, (April 1984).

[LePo84] Levinthal, A. and Porter, T., "Chap, a SIMD Graphics Porcessor", in *Proceedings of SIGGRAPH'84*, also published as Computer Graphics, 18, 3, (July 1984), 77-82.

[MaVN68] Mandelbrot, B. B. and Van Ness, J. W., "Fractional Brownian Motion, Fractional Noises and Applications", SIAM Review, **10**, **4**, (October 1968), 422-437.

[Peac85] Peachy, D. R., "Solid Texturing of Complex Surfaces", in *Proceedings of SIGGRAPH'85*, also published as Computer Graphics, **19**, 3, (July 1985), 279-286.

[Perl85] Perlin, K., "An Image Synthetizer", in *Proceedings of SIGGRAPH'85*, also published as Computer Graphics, **19**, 3, (July 1985), 287-296.

[Pipe84] Piper, T. and A. Fournier, "A Hardware Stochastic Interpolator for Raster Displays", in *Proceedings of SIGGRAPH'84*, also published as Computer Graphics, **18**, 3, (July 1984), 83-91.

[ReBl85] Reeves, W. T. and Blau, R., "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems", in *Proceedings of SIGGRAPH'85*, also published as Computer Graphics, **19**, 3, (July 1985), 313-322.

[Reev83] Reeves, W., "Particle Systems - A Technique for Modelling a Class of Fuzzy Objects", in *Proceedings of SIGGRAPH'83*, also published as Computer Graphics **17**, 3, (July 1983), 359-376

[Smit84] Smith. A. R., "Plants, Fractals and Formal Languages", in *Proceedings of SIGGRAPH'84*, also published as Computer Graphics, **18**, 3, (July 1984), 1-10.

[Will83] Williams, L., "Pyramidal Parametrics", in *Proceedings of SIGGRAPH'83*, also published as Computer Graphics, **17**, 3, (July 1983), 1-11.