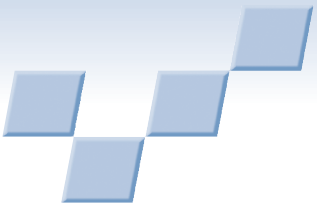


Fast Simulation of Realistic Trees

Jason P. Weber

By leveraging research in cloth dynamics and building from established mathematical algorithms, the proposed process realistically drives detailed vegetation with minimal computation. A method of dimensionally separate semi-implicit solvers allows quick reaction to not only simple force fields, such as wind and gravity, but also hard collisions against solid obstacles.

Trees in 3D interactive applications began mostly as textured billboards, scaled semitransparent pictures that spun around their vertical axis to face the viewer.[1] Those situations where vegetation began to play a stronger foreground role required higher fidelity, and simple geometric structures emerged. Although initially rudimentary and sometimes combined with textured subshapes, these new forms signaled a focus on increasing requirements for botanical structures.

In addition to a vastly improved spatial presence, these articulated bodies allow for motion beyond simple scaling and shearing. Artistically driven animations or programmed sinusoidal oscillations potentially let the viewer imagine light winds, but these techniques have difficulty portraying heavy or gusty winds and can't react to unpredictable events. At some point, plants will have to directly interact with other scene components by detecting and resolving collisions.

Rigid-body simulations, a way to deal with collisions, are becoming increasingly popular in interactive applications such as games. General rigid-body methods have demonstrated success using a variety of basic solids, such as boxes and barrels. However, more intricate objects have benefited from more specialized simulations—in particular, the implementation of vehicles, particle effects, and cloth[2] using spontaneous environmental constraints.

Simulation of vegetation has been generally limited to the influence of force fields such as wind and gravity,[3] without considerations for collisions.

We've developed a method that uses separable projections and streamlined mechanics to efficiently simulate the motion of vegetation. The associated algorithm is about 15 to 2,000 times faster than recently published methods.

The approach

For comparative benchmarking and validation of our results, we need a proven model for tree structure. We briefly describe this structure to establish the branching patterns, number of segments, and angles involved in producing the geometry.

Structure

We base the model on fixed-length rods attached end to end, each held up by angular springs that resist deflection from a rest angle relative to its parent.

For each tree's structure, we started with an artistically driven model.[4] This provides an easy method to generate the length, radius, and angles of all the stem segments. To enhance the realism of motion, we've made the two following improvements to the model, which are based on observations in art references and were previously used in a botanical branching model.[5] Early references to these properties date back to Leonardo da Vinci.

First, we strictly enforce the continuity of the sum of cross sections across each fork in the structure. In the case of a single branch from the main trunk, this means that the difference in the cross-sectional area of the trunk from just before the fork to right after the fork is exactly equal to the cross-sectional area of the new branch at its root.

Second, when a branch forks off at a particular angle, the parent is pushed away at an opposing angle with a magnitude inversely proportional to the ratio of the parent's cross section after the fork to the total cross section before the fork.

Both of these features can be scaled at each branching level to temper their effects.

Mechanics

An alternate approach[6] built a hierarchical simulation that employs some of the same concepts as a well-known robotics model,[7] although the motion was based purely on unit *quaternions*. A quaternion is a particular four-component hypercomplex number useful for encoding rotations in 3D space. This approach also used sprung chains of rigid segments to

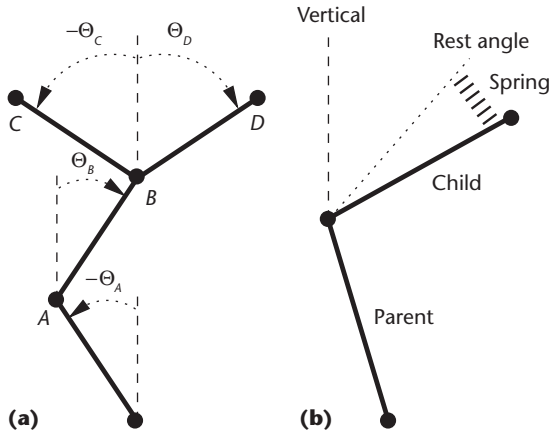


Figure 1. A (a) four-node tree and an (b) angular spring. These are the basis for a very simple tree in a 2D plane.

represent plant stems. The quaternions produced a twisting motion in addition to basic bending. The model used precise equations for the mass moment of inertia of truncated cones. Yet, by using chains of rods, the underlying model already deviates fundamentally from the microscopically continuous properties of natural wood.

Our new approach is faster and more stable. To maximize performance, we first determine what potential capabilities we can eliminate. First, we can ignore axial twisting of branches. Whatever motion might have resulted from true rotation about the axis of a stem can be expected to often occur, at least to some extent, by a combination of bending motions of other segments. Additionally, literal torque computation is unnecessarily expensive. We're already replacing a continuous natural object with a model of rigid rods. Absolute mechanical precision isn't essential. Finally, we neglect general self-collision. The potential intersecting motion of numerous branches is unlikely to be noticed in the context of the greater object.

In our prior implementation,[6] we used an explicit forward Euler method, where the future state is based purely on the forces at the current state. Explicit models tend to convert error into an accumulation of energy. For example, in a planetary simulation of an orbiting satellite, the use of only the present state to project motion forward will cause the orbit to expand. Without constant restraint, these systems can quickly explode.

While we could restrain the system for trees, we had difficulties getting the objects to settle. In the context of massive main branches, the nearly weightless terminating twigs would refuse to reach a completely still state. In contrast, a semi-implicit model will tend to convert this persistent error into a dissipation of energy. With the orbiting satellite, the use of the future state over the entire time step cuts into the perfect curvature of motion, and the orbit will shrink and collapse. This results in a damping of velocity much like drag. The settling tendency this produces will consistently result in a much more stable system with little need to impose fine substeps in time or to apply unrealistically heavy viscosity to the environment.

To reduce the computational load, we aggressively minimize the degrees of freedom (DOFs). Although generating plausible motion with only one DOF per node is inconceivable, we propose a method that's faster than two general DOFs. We model the system as a pair of separable 1D simulations in 2D space, for which each segment contributes one DOF to each simulation. During each time step, this method can recombine the computations' results into the motion of a fully 3D structure.

Semi-implicit form

Published methods for vegetation dynamics aren't nearly as common as for other domains, such as cloth. Fortunately, we can still leverage the similar research. An equation popularly used to simulate cloth[8] can be equally helpful in solving hierarchical structures:

$$\left(\boldsymbol{M} - h\frac{\partial f}{\partial v} - h^2\frac{\partial f}{\partial x}\right)\Delta\boldsymbol{v} = h\left(\mathbf{f}_0 + h\frac{\partial f}{\partial x}\mathbf{v}_0 + \frac{\partial f}{\partial x}\mathbf{y}\right) \quad (1)$$

At a particular time step, the only unknown in this equation is the change in velocity $\Delta\boldsymbol{v}$, a column vector. $\boldsymbol{M}$ is a diagonal matrix representing the masses in the system. The column vector $\mathbf{f}_0$ represents the forces on each node at the beginning of the current time step. h is the advancement in time for the next step forward, often selected as the period between the video refreshes. The partial derivatives $\partial f/\partial x$ and $\partial f/\partial v$ are square matrices that describe the change in force relative to a change in position or velocity. These factors are key to the stability. The column vector $\mathbf{v}_0$ is the velocity at the beginning of the current time step. The optional column vector $\mathbf{y}$ lets us impose a positional change in the current time step, such as for reactions to collisions.

We solve Equation 1 for the velocity change $\Delta\boldsymbol{v}$ and then compute the position change Δx as follows:

$$\Delta x = h(\mathbf{v}_0 + \Delta\boldsymbol{v})$$

The planar model

First, we describe a single 1D simulation in 2D space for which recombination is unnecessary. To introduce the method, we use a simple example system of four connected rods constrained to a plane (see Figure 1a).

Each node has one DOF, the angle from vertical in world space, positive to the right. Each node has a rest angle relative to its parent. Angular displacement

from that rest angle causes a force against a spring (see Figure 1b).

Overall, a time step is processed in three stages. The first stage accumulates the forces for the current time step while detecting and correcting collisions. The second stage applies a conjugate-gradient solver. The final stage runs kinematics over the full model. For this, it moves each segment's base point to its parent's end point. It computes that end point from the parent's world angle and length.

In an interactive simulation, the advancement of a time step is usually followed by a rendering of the object. Although the two jobs don't have to be synchronized one-to-one, the simulation's stability should easily allow whole time steps at the display rate, such as 1/60 of a second.

The first step is to determine the force on each node. Each relationship between a parent and its child involves a spring, with a spring constant k, which presses equally on both nodes in opposite directions. At the rest angle relative to the parent, denoted as R, neither the parent nor the child receives any spring force from the relationship. There's also a structural drag c that resists motion relative to the parent. This isn't a viscous drag but a resistance to bending changes.

At this point, we shift from treating the variables as angles to just positions of weights on a line. For node N with parent P and a number of children n iterated by i, we compute the force using "positions" $\bar{\Theta}$ and velocities $\bar{v}$:

$$f_N = -k_N(\Theta_N - \Theta_P - R_N) - c_N(v_N - v_P) + \sum_{i=0}^{n} k_i(\Theta_i - \Theta_N - R_i) + c_i(v_i - v_N)$$

For this example, we keep all values in world space. For root nodes, we omit the parent terms Θ_P and v_P. For terminal nodes, there are no child terms. The complete tree with four nodes yields the following system, used explicitly to provide $\mathbf{f}_0$ for Equation 1:

$$f_A = -k_A(\Theta_A - R_A) + k_B(\Theta_B - \Theta_A - R_B) - c_A v_A + c_B(v_B - v_A)$$

$$f_B = -k_B(\Theta_B - \Theta_A - R_B) + k_C(\Theta_C - \Theta_B - R_C) + k_D(\Theta_D - \Theta_B - R_D) - c_B(v_B - v_A) + c_C(v_C - v_B) + c_D(v_D - v_B)$$

$$f_C = -k_C(\Theta_C - \Theta_B - R_C) - c_C(v_C - v_B)$$

$$f_D = -k_D(\Theta_D - \Theta_B - R_D) - c_D(v_D - v_B)$$

Next, we assemble the partial derivatives of the force on each node relative to the angle and velocity of every node, including the node itself, into two matrices. For our example, this results in two 4×4 matrices:

$$\frac{\partial f}{\partial \Theta} = \begin{pmatrix} -k_A - k_B & k_B & 0 & 0 \\ k_B & -k_B - k_C - k_D & k_C & k_D \\ 0 & k_C & -k_C & 0 \\ 0 & k_D & 0 & -k_D \end{pmatrix}$$

$$\frac{\partial f}{\partial v} = \begin{pmatrix} -c_A - c_B & c_B & 0 & 0 \\ c_B & -c_B - c_C - c_D & c_C & c_D \\ 0 & c_C & -c_C & 0 \\ 0 & c_D & 0 & -c_D \end{pmatrix}$$

All k_N and c_N are constants. For the effect of gravity, we can use the following equation: on node N,

$$f_{gN} = m_N g \sin(\Theta_N)$$

where m_N is a node's mass and g is the constant acceleration of gravity. This term is added to the force for each node and contributes solely to $\mathbf{f}_0$ in Equation 1. We don't reflect this term in the partial derivative matrices because it's a nonlinear effect in this angular space and its contribution to those matrices would be substantially smaller than that of the springs. This inaccuracy simply means that we use a fixed contribution over each small time step even though the force might be changing slightly. We can add wind in the same manner, even if it isn't uniform.

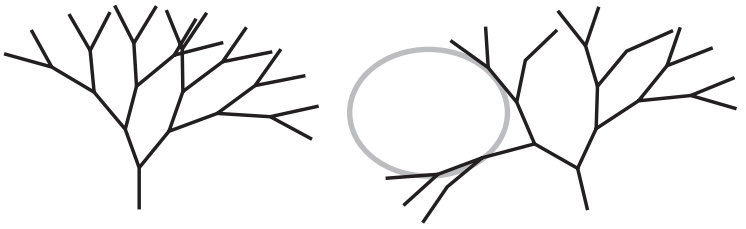
Looking back to Equation 1, where $\bar{\Theta}$ represents x, these additional equations provide the remaining values except $\mathbf{y}$. We'll use $\mathbf{y}$ in collisions. After substituting known values into Equation 1, the result adheres to a common form:

$$A\bar{x} = \bar{b} \tag{2}$$

In this abstraction, $\bar{x}$ refers to the $\Delta\mathbf{v}$ vector in the model, not Δx. In small systems, such as the four-node example, we could simply rearrange Equation 2:

$$\bar{x} = A^{-1}\bar{b}$$

However, inverting A can be prohibitively expensive when the node count increases from four to a few dozen, such as in Figure 2 (next page), and later on to several thousand. Later, we describe a popular method to solve such systems of equations.



(a) (b)

Figure 2. Simulated planar tree (a) before and (b) after collision. The system scales easily and produces stable results.

In the planar case, we can handle collisions simply by pushing the segments that penetrate the collider away from the obstacle until they no longer collide. Because there's only one angle, it's readily apparent which direction leads immediately away from the obstacle. For the current time step, we implement the change in position by immediately adjusting each affected angle Θ_N in an angular treatment to where it no longer impacts the collider surface. Situations might exist where a single pass of corrections won't completely clear the obstacle, but there might not be time to run additional correction passes. Occasionally, correcting a collision might take a few frames, or in the case of competing obstacles, a clear path might not even exist. Altering Θ_N should propagate a change to the nodes' descendants. To also affect the ancestors, we impose this displacement into the solver for the current time step by adding the position change to y_N in Equation 1. Without this factor, heavily tensed springs would pick up noticeable vibrations. We can adjust these reactions with simple scalars to soften the impact or rebalance the response of ancestors and descendants.

This model determines collisions from the root outward. If any correction occurs, each node in that node's descendant subtree is tagged as dirty. As the collision phase visits all the nodes, all dirty nodes undergo a repeated kinematic computation to generate start and end points. This ensures that the positioning is always current. So, if a node's correction brings some of its descendants closer to or further from an obstacle, those nodes will react on the basis of the adjusted position. Without this, collisions usually tend to cause excessive energy with redundant reactions.

The spatial model

You usually can't extend a simulation from one to two dimensions by simply repeating the operations a second time. But if you can frame the method so that the axes of motion are solvable separately, you can very nearly achieve that goal. In fact, truly independent procedures can run in parallel.

We treat the simulation of the spatial case as two independent instances of the planar case, each with an independent solver. When we construct a tree's initial state, we project the rest angle onto the *x*-*y* and *y*-*z* planes. After each simulation step, we reconstruct the hierarchy in 3D world space.

As a result of the two simulations, each node has an absolute angle in each 1D solver space. Standing alone, these represent the angle away from the vertical *y*-axis, one initially toward the *x*-axis and one initially toward the *z*-axis. But even though we solve these angles independently, they act collectively in world space. To recombine these two angles from separated solver space into world space, we map them into a single rotation.

An exact reverse projection would involve taking these lines on the *x*-*y* and *y*-*z* planes and extruding them along the perpendicular axes, in *z* for Θ_x and in *x* for Θ_z. The intersection of the resulting tilted planes might produce the desired line. But this works only for small angles. If one angle is approximately 90 degrees, the other angle has virtually no effect. A mathematically optimal solution could involve a commutative linear combination of the two rotations,[9] but the required matrix exponential computation would be prohibitively expensive. We could simply concatenate the two rotations, like a universal joint in mechanics, but we get better behavior by mapping them through an angle-axis form:

$$\Theta_w = \sqrt{\Theta_x^2 + \Theta_z^2} \tag{3}$$

$$axis = \begin{bmatrix} \Theta_z & 0 & -\Theta_x \end{bmatrix} \tag{4}$$

This isn't a derived mathematical conversion but an arbitrary mapping chosen for continuity and smooth behavior. It's much like the angle-axis rotations common in computer graphics, except that the magnitude of rotation is driven by the Pythagorean hypotenuse of the two inputs. We choose the axis so that when one angle is fixed at zero, the resulting motion follows the expected degenerate behavior. Because changes of sign in either direction will pass through zero, the output stays continuous.

In Figure 3, an input of positive Θ_x, where Θ_z is zero (shown in red), results in a vector in the *x*-*y* plane. An input of positive Θ_z, where Θ_x is zero (shown in green), results in a vector in the *y*-*z* plane. An input of positive Θ_x, equal to Θ_z (shown in blue), is a rotation about the line $x = -z$.

We would run into problems if we directly mapped the absolute node angles to a world rotation, such as we described with the four-node planar tree. As Θ_w approaches π, the directions of change of an increasing Θ_x or Θ_z converge and become nearly aligned. This causes poor dynamic behavior and severely hinders the system's ability to react to collisions. In fact, the orthogonality

of these direction vectors is a loose metric of the validity of the solvers' separation. Although the mapping breaks down at 180 degrees, this method preserves proper behavior well past 90 degrees.

To solve these problems and keep all the mapping angles sufficiently small, we apply this mapping only for each node relative to its parent. This technique will result in a slightly different geometric structure because it distributes the imprecise mapping's effect in slight increments over every segment instead of in potentially increasing independent jumps toward the outlying segments. In Equations 3 and 4, instead of using the absolute angle in the *x*-*y* or *y*-*z* planes, we use the angular difference for each component from the like component of the node's parent, still in the 1D solver space:

$$\bar{\Theta}_{cr} = \bar{\Theta}_{ca} - \bar{\Theta}_{pa}$$

We define these subscripts as child-as-relative, child-as-absolute, and parent-as-absolute. From this $\bar{\Theta}_{cr}$, this approach computes a relative quaternion using Equations 3 and 4. It concatenates the result to the parent's absolute quaternion, resulting in an absolute quaternion for the child:

$$Q_{ca} = Q_{pa}Q_{cr}$$

Because of this indirect mapping, it's difficult to implicitly predict how changing either angle would affect the absolute world position. This approach produces a pair of direction vectors by using Equations 3 and 4 three times for each node at the end of each time step. In addition to using the inputs (Θ_x, Θ_z), this approach runs additional passes for ($\Theta_x + a$, Θ_z) and (Θ_x, $\Theta_z + a$), where a is a small change in angle. The corresponding difference of the endpoints of the stem from the first pass to each additional pass produces a pair of directional vectors. The result isn't normalized. At small angles, these vectors are nearly orthogonal. These direction vectors are analogous to discrete partial derivatives where you can approximate the change in the endpoint for any small change in either angle.

Collisions in 3D space are a bit trickier than in the planar scenario. The magnitude and direction of interpenetration of the collision object into any particular node provides a desired correction vector. To mitigate the reaction, the adjustment scales from full intensity for a collision at a segment's end, down to no effect for a collision at the base. We do this by projecting the contact point onto the segment's axis and using the projection's distance from the base of the segment scaled as a fraction of the full length. We can also impose a multiplicative tweak factor here to soften or amplify the reaction. Overly soft reactions might cause persistent penetration; excessive reactions can cause vibration.

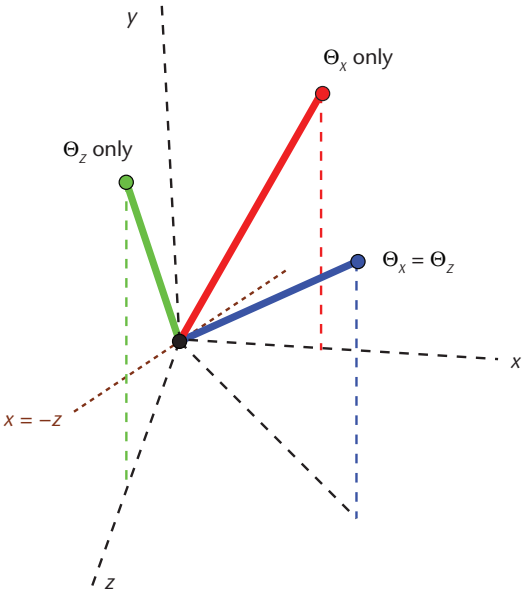


Figure 3. Conversion from separate simulations to angle-axis. This provides fully spatial results from a pair of planar solvers.

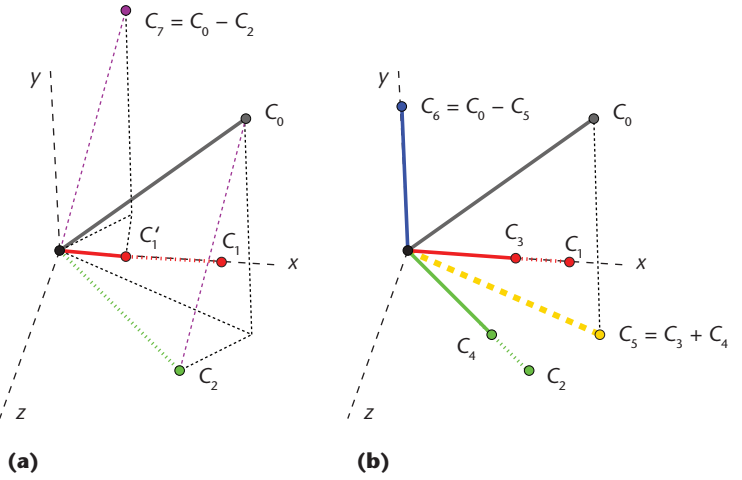
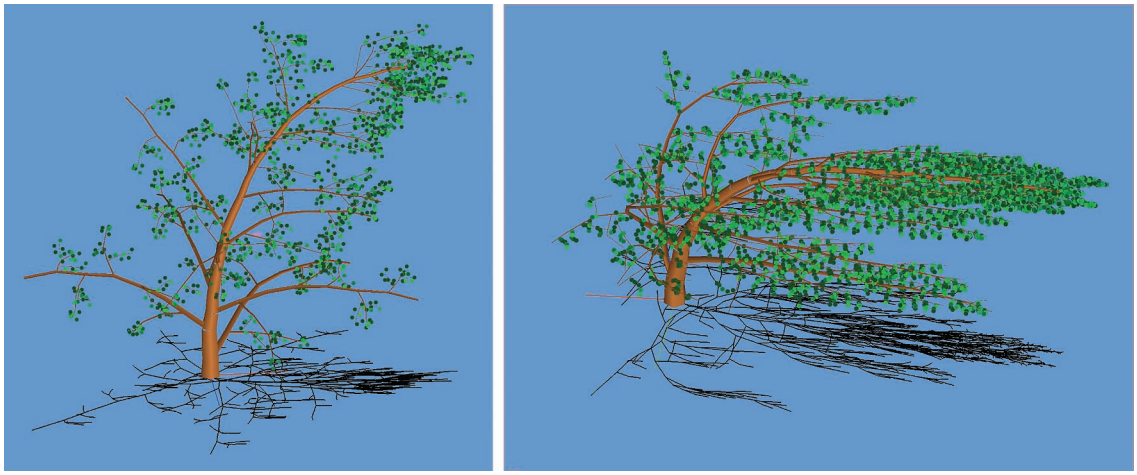


Figure 4. (a) Weighting the direction vectors and (b) approximating the correction vector. These operations confine a desired correction to an optimally scaled combination of two possible corrections.

We can adjust a node position by using a linear combination of the direction vectors we just generated. Because we have only two vectors, this can only be an approximation and generally results in a remainder that will be discarded. To fairly balance the contribution of the two directions, which aren't perfectly orthogonal, we iteratively take the difference of each scalable direction vector from the desired correction and then project that result onto the other direction vector, producing a new scale for that other vector.

Figure 4 shows an attempt to approximate the desired correction vector C_0. Suppose the direction vectors, C_1 and C_2, are along the *x*-axis and the



(a) (b)

Figure 5. (a) Heavy wind in a simple tree. All branches react based on their size and angle into the wind. (b) Ridiculous wind in a dense tree. Even under extreme conditions, the simulation remains very stable.

line $x = z$. This substantial deviation from orthogonality would represent a near approach to the 180-degree failure case. Figure 4a examines the first vector for the first iteration. To scale C_1, we take the difference of the other vector, C_2, from C_0. We then project this C_7 onto C_1, scaling it to C_1'. Afterward, this approach then uses C_1' to scale C_2, and so on, for multiple iterations, eventually converging to a C_3 and C_4 such as in Figure 4b. In practice, four iterations of both vectors converges reasonably well, at least to the extent that the given two vectors are even capable of combining to reach an arbitrary correction vector.

The iteration process we just described results in scaled versions of C_1 and C_2 such as C_3 and C_4 in Figure 4b, shown as solid red and green (upscaling and negative scales are allowed as well). The approximation of C_0 is the sum of these vectors, C_5, which will likely never match C_0 exactly. The difference, C_6 (shown in blue), is lost. This generally represents force applied down the shaft of a stem and is usually small and much less desirable. Like in the planar case, the scalars from C_0 and C_1 to C_3 and C_4 provide the y values for equation 1.

Although we demonstrate collisions with a sphere, the contact point and penetration depth of each intersection can come from any collision detection system.

As with the planar model, this approach determines collisions from the root outward during the accumulation stage. If a correction occurs at any point, that node's descendant subtree is marked dirty, which provokes a new kinematic calculation of the start and end points.

We can also take this opportunity to add a wind force $\bar{w}$ scaled to the surface area facing the wind source. We compute the wind force on node N along each of the separable axes, with its unit correction direction $\hat{d}_N$, length l_N, and approximate radius r_N, as

$$f_{wN} = l_N r_N \left(\hat{d}_N \bullet \bar{w}\right)$$

This wind force is demonstrated in Figure 5.

The solver

The conjugate gradient is a popular method for iteratively solving systems of linear equations that conform to Equation 2, repeated here:

$$A\bar{x} = \bar{b}$$

Given A and $\bar{b}$, $\bar{x}$ is solved. The matrix A is generally always sparse, meaning that most elements are zero. This can provide substantial opportunities for optimization. A conjugate-gradient solver takes a series of steps, orthogonal in A, to reduce a residual initialized with $\bar{b} - A\bar{x}_0$ and seeded with an initial state $\bar{x}_0$.[10]

For this approach to safely use the conjugate gradient, the matrix A must be symmetric and positive-definite. Also, the diagonal should be fully populated with nonzero values. But the wide variation in mass in a botanical structure can preclude the solver from quickly converging to a solution, if at all. The ideal matrix would have all its diagonal elements equal to one. To achieve this, we use a carefully selected preconditioner:

$$C = \sqrt{A_D}$$

A_D is A with all the nondiagonal elements set to zero. We use this to modify Equation 2 to an equivalent form:

$$\left(C^{-1}AC^{-1}\right)\left(C\bar{x}\right) = \left(C^{-1}\bar{b}\right)$$

The first parenthetical term results in a matrix with a unit diagonal that solves easily. To recover the true $\bar{x}$, we premultiply the resulting solution of $C\bar{x}$ by C^{-1}.

Performance

The goal of this method is a substantial increase in speed relative to prior published methods. We describe the circumstances of our testing and provide quantitative results.

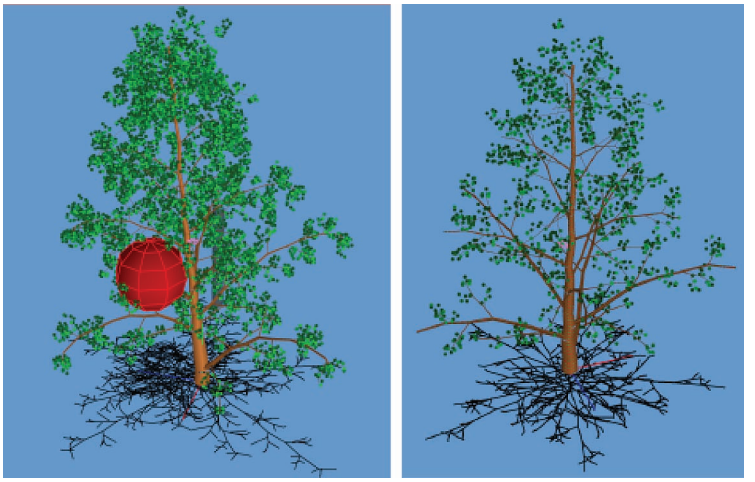
Threading

With the emergence of multicore processors and the increased availability of multiprocessor platforms, it has become important to demonstrate how an algorithm can be distributed among multiple threads.

Because we eliminate any dependence between Θ_X and Θ_Z during the solver stage, we can solve the system in two parallel parts using two fully independent conjugate-gradient solvers. On a loaded system, using the two solvers in parallel can reduce the solver time by nearly half.

The kinematic stage can be broken down into basically any number of threads, although too many threads could pick up excessive overhead. For this procedure, we set up a job queue of subtrees. As the master thread processes the kinematics on the zeroth level, whenever it reaches a child not of the zeroth level, it adds that subtree to the job queue instead of running its kinematics directly. When the master thread finishes the nodes of the zeroth level, it can participate in the job queue along with any number of worker threads.

In an interactive situation, we can also do the rendering on a separate thread. The interactive application sends abstracted rendering commands, such as polygon lists, to a command cache. Analogous to the double buffering in OpenGL, a synchronization command signals that the current writable cache is complete. After a possible wait for the rendering thread to finish processing the current rendering cache, the simulation handler swaps the write cache with the rendering cache, and both threads continue.



(a) (b)

Figure 6. (a) Dense tree with 2,246 segments and 6,840 leaves and (b) simple tree with 907 segments and 1,530 leaves. Behavior is consistent with various simulation densities.

Profiling results

We compare the performance of the dense tree in Figure 6a (2,246 segments) with that of the simpler tree in Figure 6b (907 segments). We run each tree using a single thread first and then with multiple threads. Table 1 shows the results with no collider; Table 2 shows the results with a collider that orbits closely around the trunk. Using

Table 1. Frame cost without a collider (microseconds). Higher density has only a small effect on cost per node. Multithreading provides a substantial improvement.

Phase	Simple		Dense	
	Single-threaded	Multithreaded	Single-threaded	Multithreaded
Populate	316	431	1,150	1,316
Solve	430	207	1,023	914
Kinematics	901	457	2,512	1,159
Cost per node	1.81	1.21	2.08	1.51

Table 2. Frame cost with an orbiting collider (microseconds). Costs increase evenly with a slight favor toward multithreading.

Phase	Simple		Dense	
	Single-threaded	Multithreaded	Single-threaded	Multithreaded
Populate	935	1,150	2,758	3,004
Solve	1,594	813	3,737	2,266
Kinematics	980	576	2,595	1,114
Cost per node	3.86	2.79	4.04	2.84

a fixed collider in contact with the tree results in nearly the same performance as with no collider, as long as the simulation is given enough frames to settle.

We performed the tests on a Linux 2.6.9 kernel with two AMD Opteron 275 dual-core processors. All nodes were active at all times. We didn't influence the measurements with any multiresolution or hibernation techniques, which can substantially increase performance under the right conditions.[6,11] This is an important consideration because a whole forest of trees is clearly too much load for current platforms, even if the system has several processor cores to divide the effort. But using carefully chosen distance and occlusion metrics, likely based on camera position and properties, would allow much of the plants to be still, to be approximated by deformable or oscillating curves, or to be simplified with variable physical resolution. Also, some applications might be able to tolerate much lower simulation resolution than we demonstrated, perhaps with only 100 or so active nodes.

This simplified method recreates the natural behavior of vegetation with a minimum of computational expense.

The populate phase involves processing contacts from any collisions, redoing minimal kinematics as needed, and then forming the dynamic $\bar{b}$ for the current time step. The solver stage applies preconditioning and runs the two solvers. The kinematic stage includes quaternion and direction calculation, leaf transformation, recalculation of the tree surface geometry, and an accumulation of forces pushing back on any colliders. This phase doesn't include the minor kinematic adjustment from collisions.

For the dense tree with an orbiting collider, running the separable solvers in parallel takes 61 percent of the time that those solvers take serially. Also, running the kinematics with four threads in parallel consumes only 43 percent of the corresponding serial time.

In these tests, the system scales almost linearly. With the increased load of the collider, a 150 percent increase in nodes raises the cost per node less than 5 percent. A conjugate gradient has $O(m)$ complexity, where m is the number of nonzero elements in A.[10] Because the system consistently contributes a base of three or four nonzero elements per node, this suggests $O(n)$ complexity. The variation in element count is due to forking, so higher amounts of branching can have increased complexity.

For comparison, consider a noncolliding model of a 2D nonbranching minimally rigid pendulum[11] using a Featherstone[7] implementation. That system takes 6 milliseconds to solve 300 nodes, each with only one DOF. If we linearly adjust from that 2.8-GHz processor to our 2.2 GHz, this equates to a cost per node of more than 25 microseconds. Using our model to simulate a limp, branchless stem of 300 segments, the total simulation time is 440 µs single-threaded, which is 1.5 µs per segment and 0.7 µs per DOF.

For a complex branching tree, consider another Featherstone implementation, specifically optimized for hair and vegetation.[12] In a simple system of 135 segments, this implementation achieved a best case of 250 ms, about 2 ms per node. For a larger system of 4,500 segments, it achieved a best case of 20 seconds, about 4 ms per node. This is comparable to the noncolliding case of our model, which achieves 1.8 µs per node single-threaded and 1.2 µs per node multithreaded. Adding an aggressive collider raises the cost per node to only 3.8 µs single-threaded and 2.8 µs multithreaded.

Memory use consists of two primary parts: the geometry and the solver. The geometry data is populated mostly with segments and their various properties. In our sample implementation, a segment uses approximately 60 32-bit words. The solver data consists of sparse matrices and fixed-length vectors. The implementation of the sparse matrices uses two words per nonzero entry: the value and the horizontal position. Including the dual solvers' expense, our implementation uses 10 matrices and 21 vectors (although if we neglect code readability, we could probably reuse or eliminate several of these). This works out to approximately 80 to 100 words each for the solver, for a total expense of 140 to 160 words per segment. A well-represented tree could consume about 1 Mbyte.

This simplified method recreates the natural behavior of vegetation with a minimum of computational expense. The model works best for stiff structures where the angles between segments don't approach 180 degrees. The current implementation isn't suitable for soft vine-like structures, including those in trees such as the weeping willow.

Performance tests have demonstrated the reduced system's benefits. Using two $N \times N$ systems instead of a combined system of at least $2N \times 2N$ not only substantially reduces the work that the solver requires but also lets us divide the solver and run it in parallel.

The full source code for the programs that produced the images in this article is available at www.freeelectron.org. This includes converted tree specifications from prior work,[4] as shown in Figure 7.

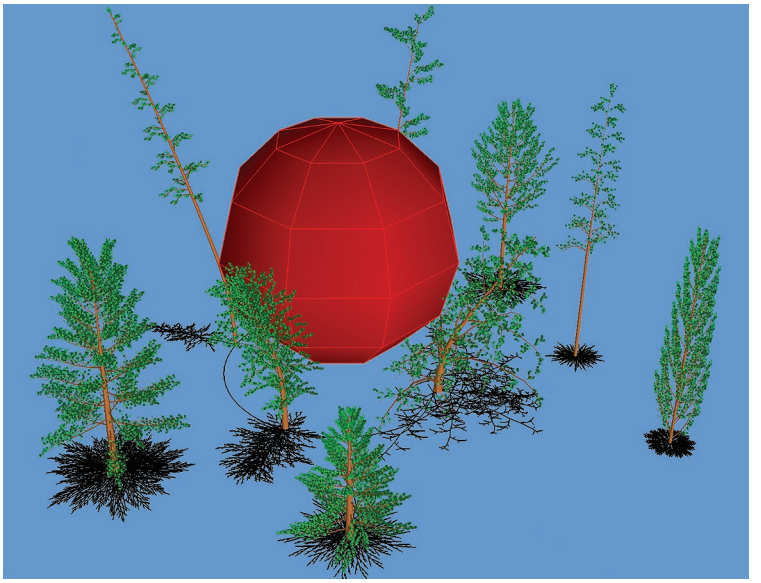
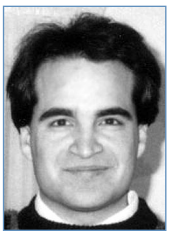


Figure 7. A collection of trees from the original structural model. Equivalent parameters are easily determined.

References

1. M. Jones, "Lessons Learned from Visual Simulation," *Siggraph 94 Course Notes: Designing Real-Time 3D Graphics for Entertainment*, ACM Press, 1994, pp. 18–19.
2. X. Provot, "Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior," *Proc. Graphics Interface*, A.K. Peters, 1995, pp. 147–155.
3. J. Beaudoin and J. Keyser, "Simulation Levels of Detail of Plant Motion," *Proc. Eurographics/ACM Siggraph Symp. Computer Animation*, ACM Press, 2004, pp. 297–304.
4. J. Weber and J. Penn, "Creation and Rendering of Realistic Trees," *Proc. Siggraph*, ACM Press, 1995, pp. 119–128.
5. M. Holton, "Strands, Gravity, and Botanical Tree Imagery," *Computer Graphics Forum*, vol. 13, no. 1, 1994, pp. 57–67.
6. J. Weber and A. Weber, "Real-Time Multiresolution Dynamics of Deeply Hierarchical Bodies," *Graphics Programming Methods*, J. Landers, ed., Charles River Media, 2003, pp. 27–36.
7. R. Featherstone, *Robot Dynamics Algorithms*, Kluwer Academic Publishers, 1987.
8. D. Baraff and A. Witkin, "Rapid Dynamic Simulation," *Cloth Modeling and Animation*, D.H. House and D.F. Breen, eds., A.K. Peters, 2000, pp. 145–173.
9. M. Alexa, "Linear Combination of Transformations," *Proc. Siggraph*, ACM Press, 2002, pp. 380–387.
10. J.R. Shewchuck, "An Introduction to the Conjugate Gradient Method without the Agonizing Pain," School of Computer Science, Carnegie Mellon Univ., pp. 30–41, www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.
11. S. Redon, N. Galoppo, and M.C. Lin, "Adaptive Dynamics of Articulated Bodies," *ACM Trans. Graphics* (Proc. Siggraph), 2005, pp. 936–945.
12. S. Hadap, "Oriented Strand—Dynamics of Stiff Multi-body System" *Proc. Eurographics/ACM Siggraph Symp. Computer Animation*, ACM Press, 2006, pp. 91–100.



Jason P. Weber *is a character effects developer at DreamWorks Animation. His research interests include real-time computer graphics, physical dynamics, natural environments, and game technology. Weber received a BS in electrical engineering from Virginia Polytechnic Institute and State University. Contact him at baboon@imonk.com.*