REAL-TIME TREE SIMULATION USING VERLET INTEGRATION

by

BOBAK MANAVI
B.S. University of Cincinnati, 2001

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Modeling and Simulation
in the College of Sciences
at the University of Central Florida
Orlando, Florida

Summer Term
2007

ABSTRACT

One of the most important challenges in real-time simulation of large trees and vegetation is the vast number of calculations required to simulate the interactions between all the branches in the tree when external forces are applied to it. This paper will propose the use of algorithms employed by applications like cloth and soft body simulations, where objects can be represented by a finite system of particles connected via spring-like constraints, for the structural representation and manipulation of trees in real-time. We will then derive and show the use of Verlet integration and the constraint configuration used for simulating trees while constructing the necessary data structures that encapsulate the procedural creation of these objects. Furthermore, we will utilize this system to simulate branch breakage due to accumulated external and internal pressure.

TABLE OF CONTENTS

LIST OF FIGURES iv

INTRODUCTION 1

MATERIALS AND METHODS 4

Methodology Overview 4

Simulation Forces 5

Mathematical Analysis 6

Verlet Integration 6

Iterative Constraints 8

Tree Structure and Algorithms 11

Branch Structure 11

Branching Plane Structure 12

Connecting Branches 12

Root Structure 15

Tree Generation 15

Solving the Forces 17

Breaking Branches 18

Adjustable Variables 20

RESULTS 22

CONCLUSION 25

LIST OF REFERENCES 27

LIST OF FIGURES

Figure 1 - Simulation Tree Model 11

Figure 2 – Branch Parenting 14

Figure 3 – Simulated Oak Tree From SimTree Application.. 16

Figure 4 – ~80 mph Wind Effect on Oak tree 23

INTRODUCTION

In the field of real-time simulation great effort has been expended to simulate trees, both in realistic visual representation and in physically accurate force-dependant simulations. Some researchers have focused on correct representation of the tree [22] while others have focused on the physical simulation of the tree regardless of its shape [12]. And some researchers have focused on the link between the two, where the shape can directly influence the physics simulation [2]. The ultimate underlying goal would be to have a tree simulated that is visually correct and responds physically accurately to external forces.

Research relevant to the visual representation of trees is abundant. Over the past decades similar work like Wessl´en and Seipel [22] has taken advantage of the increase in rendering performance on computers, by using specialized hardware available to the general consumer, to address issues relative to visual representation of trees. This has greatly advanced the perspectives of future researchers in this field by showing how complex processes can be offloaded to dedicated hardware for greater speed using shaders. Classifications of tree generating algorithms have also been widely studied over the past few decades. Geometric tree growth algorithms and tree growth algorithms based on botanical principles have been adopted by different researchers for their advantages and disadvantages in rendering trees [1, 3, 4, 10, 11, 16, 17, 21].

But, as mentioned earlier, there is a relative dependency between the shape of a tree and the effect of external forces on the branches of the tree. Akagi and Kitajima's [2] research in this field focuses on this relative connection, and uses Navier–Stokes equations of motion in fluid dynamics [19] to simulate the wind current coupled with other techniques that focus on the

homogeneity of the branches and leaves and models their resistance. Their focus is on simulating the wind and not just the tree. Other work by Ota et al. [12] uses mode functions to animate the deformation of the branches and leaves affected by the forces of wind, while excluding the changes in airflow, and focusing computation time solely on the tree animation.

A common data structure used for representing trees – for rendering or simulating, is one that includes position and rotation information, which indicate the recursive child's position and orientation relative to its parent branch [2]. This information can be easily used to animate the swaying of the tree using techniques such as vertex morphing [22] or animation using joints [2], which can be physically derived, or not [22].

Comparable to the vast amount of research done in the field of real-time tree visualization and simulation is research done in the field of cloth and soft body simulation [5, 6, 7, 9, 13, 15, 18, 20]. Previous work done in this area of real-time simulation can be dated as far back as 1930's in the textile industry [13]. More recent work in this field has broadened the horizon for future research. The underlying principle that shapes the foundation of research in this field is finite-element modeling with spring constraints [5, 6, 9, 13]. A recent proposal presented by Dr. Jos Stam, at the Montreal International Game Summit, extends the particle and constraint system by exploring the possibilities of incorporating this approach in simulating all forms of non-rigid and rigid body physics models [18].

In this paper we will combine both fields into one uniform approach for simulating trees. A combination of tree simulation techniques and cloth simulation algorithms will be utilized to accurately simulate, in real-time, the effects of external forces on trees. Also, a key component of this research is the ability to sever branches from the tree based on external forces, an effect made possible because of the finite-element composition approach.

The idea to develop this system was brought on by on going research done at University of Central Florida in the Hurricane Visualization Project at IST. The key element that can make simulating trees in real-time almost impossible is the number of factors that can affect the shape and orientation of individual branches. Accurately simulating a tree is well beyond real-time processing with current technology. But assumptions and optimizations can be made when modeling a tree that can allow real-time simulation that behaves accurate enough.

The approach taken in this paper is similar to approaches used in simulating cloth. The subject of cloth simulation has led to a reasonably solid solution for updating and rendering cloth in real-time. The same method can be used to effectively simulate soft-body physics. This process requires the strategic positioning of particles to encapsulate the shape of the desired object, and the creation of constraints (springs) between particles to help the mesh keep its shape. Due to external forces, these particles will shift and move. Once a particle moves, the springs connecting the particle will get stretched or compressed, and forces are calculated based on these modifications in length. The forces are then used to update the particle's position, and to help keep all particles in the relative position from which they initially started from.

One major difference between common algorithms used for cloth simulation and the algorithm used by this system is the choice of the integrator. This modification of Euler integration used in this system is known as Verlet integration, and is derived in the next section.

MATERIALS AND METHODS

Methodology Overview

The main components of this approach are the choice of integrator and the configuration of constraints. The goal is to speed up calculations enough to be able to simulate a tree with sufficient detail in real-time. First, we will remove the common concept of a tree or a branch that is connected to other parts of the tree using a joint. A joint is a local offset transformation matrix for each section relative to its parent. In this system the concept of a joint is replaced with a particle. A particle on its own has no orientation information, and if required, it can be derived when necessary from its constraint connections.

In a joint based system one will have to compute local forces – linear, angular momentum and velocity, to derive a rotation matrix that can be applied to the joint, and rotate the branch. These are calculation steps that can be avoided with the use of Verlet integration as our preferred integrator for updating the particles in conjunction with the use of iterative constraints for simulating the springs connecting these particles. Verlet works off of the change in previous time-steps' positions to figure out its next position in time, while iterative constraints estimate and modify the position of the particles they are connected to based on a difference in distance between two time-steps. This setup will result in implicit application of force onto particles and Verlet integration will handle the rest.

A key point to be made here is the residual effect of multiple constraints on a single particle, and that is why the configuration of constraints is also an important factor in this approach. More specifically, since a single particle can be connected to multiple particles via constraints, it is possible that once the particle position is solved at the end of a given time-step

that this position may vary by the time the beginning of the next simulation time-step has been reached. This is due to the fact that other particles connected to our particle of interest may have also moved in that same simulation time-step. Once the constraints connecting these particles are solved the result is a pull or a push of our particle of interest, and therefore, a change in position, which would also mean an implicit application of force and calculation of velocity.

Simulation Forces

As mentioned earlier, the proposed system uses an adaptation of cloth simulation techniques in terms of the algorithm used for solving the constraints and updating the particles [9, 13]. In the simulation cycle, for each time step, the particles are affected by the accumulated forces on them. These forces can be local to the specific particle, or global to the entire tree system. A global force is similar to gravitational force or wind forces that affect every particle in the system. A local force is similar to someone grabbing a tree branch and pulling it down towards them. These forces are accumulated and applied to the corresponding particles in the system prior to calling the integrator on them. The next section will derive the mathematical basis for Verlet integration for better understanding of its implicit computation of velocity.

Mathematical Analysis

Verlet Integration

Verlet integration is the preferred integrator used for solving the positions of the particles. The integrator used in this simulation is one that implicitly calculates the velocity. The use of this integrator takes into account the particles position change between every frame, and utilizes this information to calculate a new position for the particle. The new calculated position is then used to update the particle's position in the next simulation time step. We will derive Verlet integration to better understand the correlation between velocity and its effect in this system, and to define the impact of Verlet integration on the overall effectiveness of its use in this system. Using Taylor Series we can represent a function $f(x)$ in terms of the series expansion of the function about a point as displayed in Equation 1 below:

$$f(x+\Delta x)=f(x)+f'(x)\Delta x+\frac{f''(x)}{2!}.\Delta x^2+\frac{f'''(x)}{3!}.\Delta x^3+\ldots+\frac{f^n(x)}{n!}.\Delta x^n \quad (1)$$

Each particle in the finite system of the tree can be defined by its position as a function of time (t), where $f(x)=\vec{r}(t)$. Using this simple substitution we can simply rewrite Equation 1 to obtain a more constant relevant version of the same equation shown below in Equation 2. Here $\vec{r}(t+\Delta t)$ is the position of the particle as a function of time at (Δt) time in the future with $\vec{v}(t)$ as the first derivative of our particle's position – velocity, and $\vec{a}(t)$ as the second derivative of our particle's position – acceleration.

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \vec{v}(t).\Delta t + \frac{\vec{a}(t)}{2}.\Delta t^2 + \varepsilon.\Delta t^3 \tag{2}$$

Higher order expansions can be neglected from our calculations since they will have little impact on the accuracy of the final result. In this case any component past acceleration is not important to us – such as the third derivative of position jerk/jolt $\vec{j}(t)$. Once we have defined our function in terms of position at time (Δt) we will need to figure out a way to remove the velocity factor from the equation. This can be done by expressing the equation using time ($-\Delta t$). Equation 3 shows position as a function of $(t-\Delta t)$.

$$\vec{r}(t-\Delta t) = \vec{r}(t) - \vec{v}(t).\Delta t + \frac{\vec{a}(t)}{2}.\Delta t^2 - \varepsilon.\Delta t^3 \tag{3}$$

Finally, we will add Equation 2 and Equation 3 to get the final equation of motion shown in Equation 4.

$$\vec{r}(t+\Delta t) + \vec{r}(t-\Delta t) = 2\vec{r}(t) + 2\frac{\vec{a}(t)}{2}.\Delta t^2 \tag{4}$$

And after rearranging the variables we derive the final form of the Verlet integration function displayed in Equation 5, where $\vec{r}(t+\Delta t)$ is the final position of the particle after Δt. $\vec{r}(t)$ is the current position of the particle at time t. $\vec{a}(t)$ is the sum of all forces applied to the particle at time t. $\vec{r}(t-\Delta t)$ is the old position of the particle prior to previous integration step. Equation 5 is used to predict the position of a particle based on its old position at time ($-\Delta t$).

$$\vec{r}(t+\Delta t) = 2\vec{r}(t) - \vec{r}(t-\Delta t) + \vec{a}(t).\Delta t^2 \tag{5}$$

As we can see, velocity is removed from the final form of the equation. Velocity can be obtained easily if required. To get velocity we can evaluate the following term $\vec{v}(t)$ = $\vec{r}(t) - \vec{r}(t-\Delta t) / \Delta t$.

Again, the key information to take note here is the term $\vec{r}(t-\Delta t)$. This is the old position of the particle that needs to be saved from the previous time-step. This old position does in fact hold the velocity information implicitly, and its difference from the current position $\vec{r}(t)$ determines the estimated implicit velocity of the particle at time ($t+\Delta t$). So if the position of the particle at time (t) is modified, the velocity will implicitly increase or decrease for that time-step, and its effect will carryover to the next time-step. This is the underlying detail that allows velocity calculations to be dropped from the integration step, and allow the constraints to work hand-in-hand with the integrator – more on constraints in the next section.

Iterative Constraints

Verlet integration works in conjunction with the constraint solving method to simulate stretch and sheer effects on the tree structure. The constraints use a simple estimating approach to keep the particles within a normalized distance [7]. Every two particles have a defined distance from each other, and after forces are applied to them, the standard distance between them will change. The constraint will take the change in the distance, either being stretched or compressed, and will add half of the total change in the length of the two particles to each particle's position, connected to the constraint, in opposite directions. As explained in the previous section, this shift in position of the particles will result in the creation of velocity during

the next iteration of the Verlet integrator, and therefore, a simple adjustment of a system of constraints will have a global effect on all the particles in the system.

Since one particle is connected to more than one constraint, this solution will result in an already positioned and solved particle to move multiple times during one simulation time-step. In the end, this will not cause a problem, provided that during every simulation time-step the constraints are solved multiple times – hence the word Iterative Constraint. Allowing the system to converge to a stable state, where the preferred distance between all particles is close to the normalized distance, dictates a minimum required number of iterations needed per simulation time-step. If the number of iterations on the constraints is high, then the system will be stiff and the possibility of the structure reverting to its original shape is greatly increased. In contrast, if the number of iterations is low, then the system will be flexible and fluid in terms of reverting to its original shape. A high level of iteration is used to simulate strong trees that will not bend easily under pressure, and low iterations are used to simulate structures similar to grass or long, thin, and flexible trees and/or branches.

The seven steps in calculating and updating the constraints are listed in code logic below:

Step 1: Get particle A and B's position

Step 2: Calculate distance between A and B

Step 3: $\Delta = \textit{rest length} - \left\|\overrightarrow{AB}\right\|$

Step 4: *Stretch multiplier* = *elasticity* * (1.0 – *rest length* * length of (Δ))

Step 5: $\Delta = \Delta$ * *Stretch multiplier*

Step 6: If (particle A can be updated) particle A *position.xyz* += Δ.xyz

Step 7: if (particle B can be updated) particle B *poistion.xyz* $-=$ Δ.xyz

The variable *elasticity* is the level of correction reduction ratio when calculating the change in length. By default it is set to 1.0 but a reduction in this value will reduce the effects of the iteration on a particular constraint – keeping the number of iterations the same across a group of constraints while reducing the effect of a select few when needed. The final *stretch multiplier* is applied to Δ in order to scale its effect. Although the variables *elasticity* and *stretch multiplier* are being used in the steps above it is recommended that they are not modified to ensure stability of all branches in the final simulation.

Tree Structure and Algorithms

The configuration of the constraints and particles is essential to this system. The tree structure is broken down into 3 sections: the branch section, the Branching Plane section, and the root section. Figure 1 shows the overall constraint configuration and major tree sections.

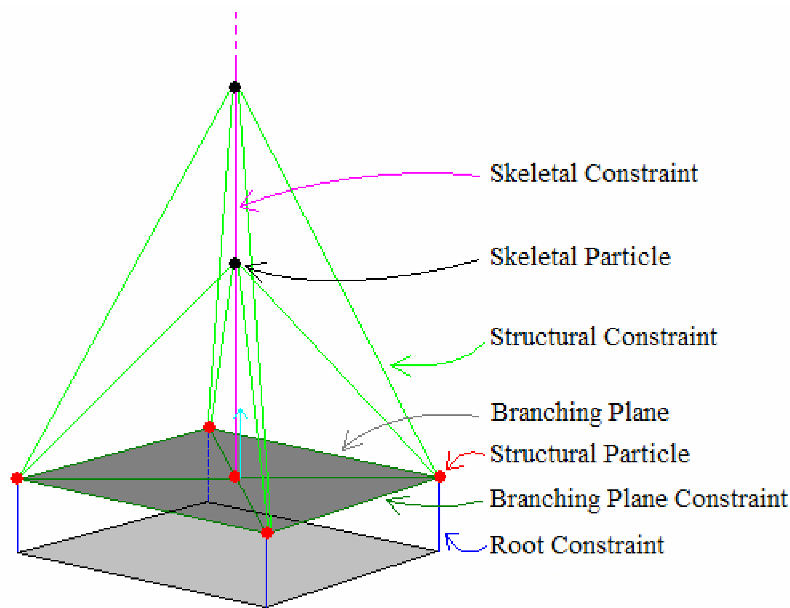

Figure 1 - Simulation Tree Model

Branch Structure

The branch section is a generalized structure that will also be used to define the tree's trunk. The branch structure is the defining structure for the skeleton of the tree. It consists of the Branching Plane (dark gray), Branching Plane constraints (dark green), five structural particles (red), the skeletal particles (black), skeletal constraints (purple), and the structural constraints (green). The use of four particles for the skeletal constraints is intentional in order to take into account the forces in every direction applied to the branch's skeletal structure. The possibility of using three particles was considered, but during the simulation testing phase, it proved to be

unstable under a limited number of iterations and the result would cause the tree trunk to bend uncontrollably in one direction and fall over at a reasonable iteration count, regardless of the stiffness of the constraints.

Branching Plane Structure

The Branching Plane is a structure used to define the direction of a branch's growth, and allow branches to connect to other branches as children. The Branching Plane has a normal that defines the direction of creation for the skeletal particles and constraints. The five structural particles are derived from the Branching Plane's position, normal, and tangent vectors. Once the five structural particles are defined, the skeletal particles are generated, and their constraints are created and connected. From every skeletal particle, a structural constraint is branched out and connected to the structural particles. These constraints are the main elements in simulating stretching and bending along the branch object.

Connecting Branches

The Branching Plane is the key in creating and generating branches, and if this structure is attached to a skeletal particle of another branch then it can shift and move with that particle while keeping its general angle of orientation relative to that particular skeletal particle. In order to connect a branch as a child to another branch we define 3 shared particles and 2 shared constraints between the branching plane of the child branch and the skeletal structure of the parent branch. The center particle of the branching plane and 2 of the structural constraint – one above and one below the center particle are actually direct references (pointers) to particles on the parent branch. Also, the constraints created between these particles are maintained by the parent branch since the particles origination is in the hierarchy of the parent branch. The

resulting effect of this unique setup is evident when forces are applied to the hierarchy. This effect would be simulated by one branch moving under force and affecting all the child branches attached to it, or one child branch moving under some force and causing its parent branch to move.

Great care must be taken when constructing these connections since, from a programming point of view, many references are pointing at the same shared memory. This is clearly evident when branch breakage is employed. Due to all the cross-referencing connection between constraints and particles in parented branches it is extremely easy to simply not properly de-reference a pointer from the child branch to the parent, or to forget to allocate the shared constraints and particles in a child branch once it is disconnected from its parent. Therefore, it is highly suggested that some form of a reference table should be maintained internally for each branch structure so it can properly identify who its parent is and what connection need to be released and allocated locally. Figure 2 shows a two branch configuration with the Branching Plane system.

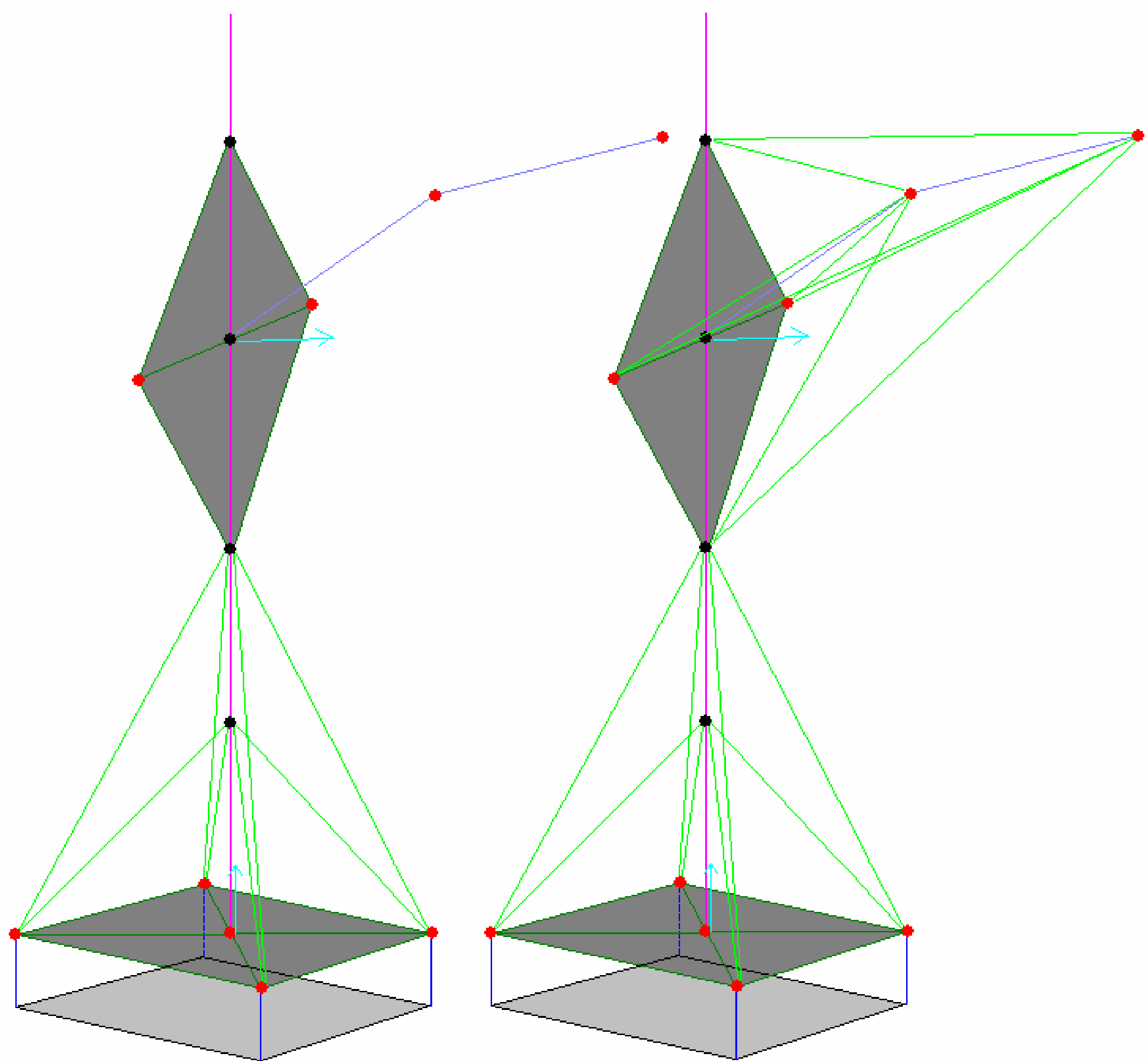

Figure 2 – Branch Parenting
Right – Structural Constraints Shown
Left – Only Skeletal Constraints Shown

Root Structure

The root section is connected to the branch section to simulate the forces applied at the collar and root of the tree. There is only one root system in the entire tree structure that is indicated with the light gray color in Figure 1 above. The root system is static and is not affected by forces in the system, which means it will not move under any circumstances. There are 4 constraints generated from the root system that connect to the main trunk's Branching Plane. These constraints will represent the forces applied to the collar and roots of the tree. The strength of the root system is defined by the number of iterations the constraints use to solve forces, and the maximum stretch length the constraints allow. The maximum stretch length is the length at which the constraint will still be able to solve and modify its connected particles. Once the length of a root constraint stretches beyond the maximum stretch length, the constraint is cut, resulting in a weight shift on the remaining root constraints and therefore simulating a tree collapsing under its own shifted center of mass.

Tree Generation

In the introduction we had mentioned the existence of many different approaches used to accurately generate a visual model of a tree. It was mentioned that geometric tree growth algorithms and tree growth algorithms based on botanical principles have been adopted by different researchers for their advantages and disadvantages in rendering trees [1, 3, 4, 10, 11, 16, 17, 21] for some time now. In this research project the focus was not on accurately modeling the visual representation of the tree, but indirectly there was a relationship. To be able to determine if this model was accurately representing a tree some assumptions had to be made. The model tree adopted by this project was an Oak tree. To model the tree a total of 11 braches

were defined that were static in nature, which means they were always going to be created at the exact same spot of the tree with the exact same number of particles for their lengths. This guaranteed that the model tree would at the least be close to an Oak tree in shape and structure. Once this foundation was established a recursive algorithm was utilized to systematically generate random branches on each of the static branches defined earlier. The final result from this assumption, seen in Figure 3, generated trees that were random in some elements while still maintaining the desired overall shape of an Oak tree.

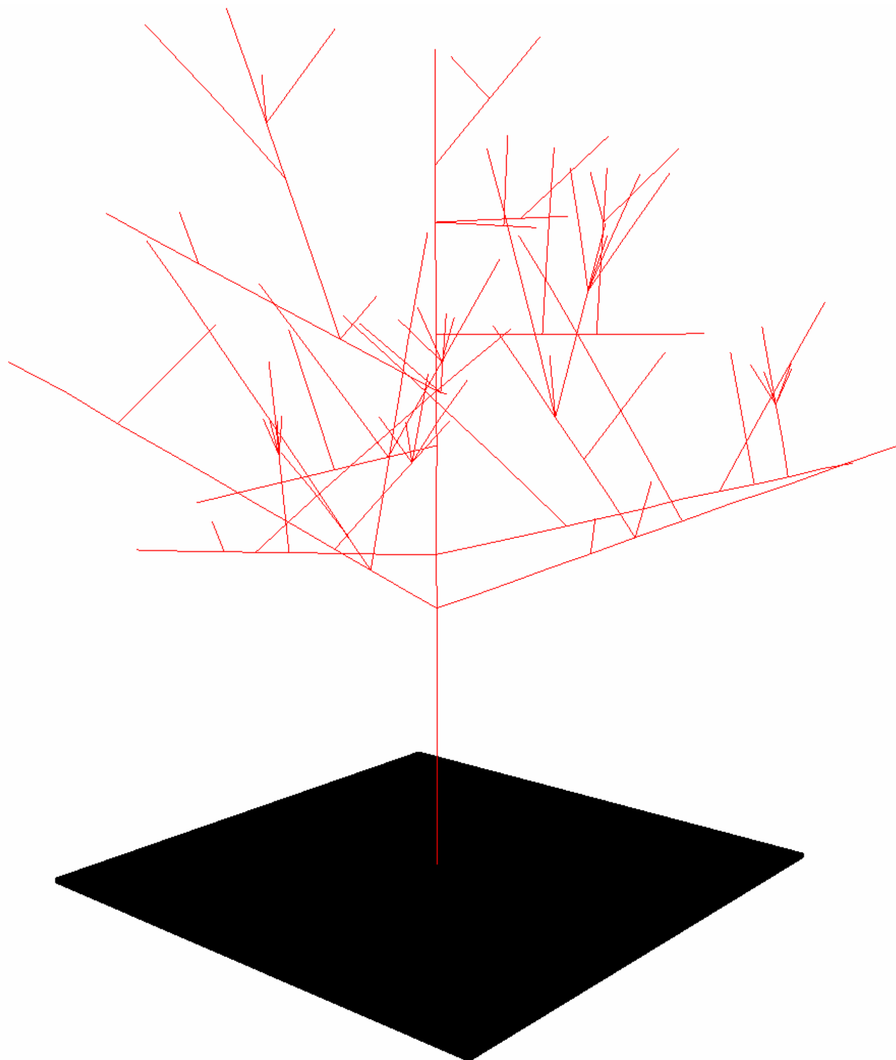

Figure 3 – Simulated Oak Tree From SimTree Application

Solving the Forces

Using the three structures defined – branch, root, and Branching Plane, one can now construct a tree. A tree object encapsulates the root, all the branch objects, and all the corresponding Branching Planes associated with each branch. When any form of global force is introduced into the system – like gravity, it is accumulated and saved in a single variable. After all forces have been accumulated the update function is called on the tree object. This function takes a time-step, which is used to update the Verlet Integration function in all the particles creating the tree object.

The next step in the update process is constraint solving. Constraints are solved for each branch starting from the main (trunk) branch independently. Once all constraints have been solved for their required amount of iterations, the root constraints are solved. At the end any major shifts in the branches will cause the trunk of the tree to shift in that direction, and any major shifts in the trunk of the tree will affect the final length of the root constraints. So in essence, the tree is solved from the inner most branches outward and the final forces are applied to the root. This will allow the possibility of breaking the root constraints upon receiving too much force from the tree it is supporting.

Every 3 particles in a branch will do a simple point-to-plane half-space collision test at the end of every loop when a constraint is solved. This is to ensure that during a time-step a branch did not get forced in an opposite direction at a great velocity due to constraint solving, and that any breakage in the branch is only caused by external forces applied to it. The following code sections show the logic flow of each structures update call.

Tree structure updates code logic:

Step 1: For each branch accumulate forces and update

Step 2: For each root's constraint iteration solve the root constraints

Step 3: Reset force

Branch structure updates code logic:

Step 1: For each particle apply forces and do Verlet integration

Step 2: Update branch's Branching Plane Verlet Integration

Step 3: For each constraint iterations solve branch constraint and do collision, solve Branching Plane constraint, and if it is a child branch do collision with parent Branching Plane.

Breaking Branches

A branch is considered broken when the final direction of its skeletal constraint is beyond a threshold relative to a cone around the initial direction of the branch. Breaking off branches is not a time dependant process but rather an iteration dependant process which means at a given time-step a branch could potentially move from a broken state to a non-broken state multiple times. This is possible due to the fact that constraints are solved one at a time, and at any given iteration a single constraint could have caused the branch to go beyond its threshold for breakage. Therefore, checking to see if a branch is broken should happen only after all constraints have been solved – generally at the end of an entire simulation time-step. This will guarantee that if the branch returns positive for being broken then there is no doubt that the final state of all constraints had placed it in that situation, and removal of the branch is valid.

At the end of every simulation time-step the branch will internally cycle through all the child branches attached to it directly and will determine if they are broken. If a branch is broken then it will detach itself from its parent internally while the parent drops any reference to that child in its local children list.

The detachment process involves multiple steps. First, the child branch will remove all attached constraints and shared particle references to its parent. The child branch will need to allocate these missing particles and redefine its Branching Plane, while keeping its direction the same as the last updated direction from the simulation time-step. The second step is to tell its parent to remove references to it so an invalid child branch is not updated by the parent by the next time-step. The third step is to remove any constraints between this broken branch's children and their grand parent branch and redirect these connections to be made with their parent instead. Once all three steps are completed the final branch will be an isolated entity that can be inserted into a global list of tree objects set for updates. A key note that needs to be taken into consideration here is the lack of a root structure. Even though once the child branch is isolated a root structure is automatically generated, it is considered a false root since it will not have any static particles to hold the branch down to the ground.

If none of the first level branches to the parent branch are detected as broken the function will continue to recursively traverse down each child branch until it finds a broken branch that can be removed. This effect happens every time-step at the end of update for all child branches until a list of all possible broken branches is constructed and added to the global list of independent tree objects in the environment.

The next section will introduce all adjustable variables used in this simulation that can be modified in order to properly model any type of tree with its unique physical characteristics.

Adjustable Variables

Some of the configurable elements in this simulation model of a branch structure are listed below:

- The distance between structural particles from the center of the Branching Plane can be modified. The modification of these elements will affect the bending force required for the branch object. The closer the particles are to the center of the Branching Plane the less force is required to bend the branch object, and more bending force will get applied under its own weight.
- Distance between each skeletal particle when creating the skeletal constraints can also be adjusted. The distance between two skeletal particles affects the flexibility in the branch structure. The farther apart two skeletal particles are, the more flexible the branch structure. This variable can be modified at different heights of the tree. For example, we can increase the distance as we create higher branch segments farther away from the Branching Plane's origin. This will allow the top segments of the branch to bend more easily relative to the stiffer segments of the branch that are closer to the collar of the branch.
- Constraint solver iterations are modifiable relative to every skeletal particle. At every branch segment we can set the number of iterations we want to use for solving the skeletal constraints independent from other skeletal particles. With this mechanism we can directly affect the flexibility of a branch segment – the higher the iteration the more rigid the branch segment, and the lower the iterations the more flexible the branch segment.

- The root structure also has a particle at every constraint. These particles are static and do not get modified when the root constraint are solved, and the entire change in length (increase/decrease) of the constraint's length is applied to a single particle on the branch. The further the distance of these particles the more strength is added to the root system in general.
- Maximum constraint length and iteration can be used to determine the strength of the root system. The higher the maximum constraint length the more flexible the root system, and the less likely it will break under pressure.

RESULTS

In order to test this research some assumptions were made and, as mentioned earlier, some of these assumptions were necessary so that a correct model for the tree could be generated. One of these assumptions was the static 11 branches generated on top of the trunk. These were used to get a relative correct shape for an Oak tree. On top of each of these static branches a few randomly generated branches were created. To generate a random branch a function was developed that would take a set of parameters, and based on those inputs it would recursively generate multiple branches. The function is initially fed a branch pointer to the static branch, a total number of particles that can be used to make branches from, a total maximum number of branches desired on this limb, a starting branch index, and a counter for the current number of branches created. The code snippet below shows this recursive function's logic:

Step 1: If current branch count is greater than total number of branches to create or there are only 3 particles in the parent branch then skip creation and return index of branch.

Step 2: For each number of branches to create:

Generate a random center of Branching Plane on parent

Generate a random number of segments from particles left in particle pool

Generate a random direction using template < rand 0-1, 1, rand 0-1>

Create branch and add to child list of parent

If we still have room to create branches on the entire tree total branch count then call the recursive function again using parent branch

Figure 4 shows the breakage of branches in action when a force of ~80 mph wind is acting on the tree.

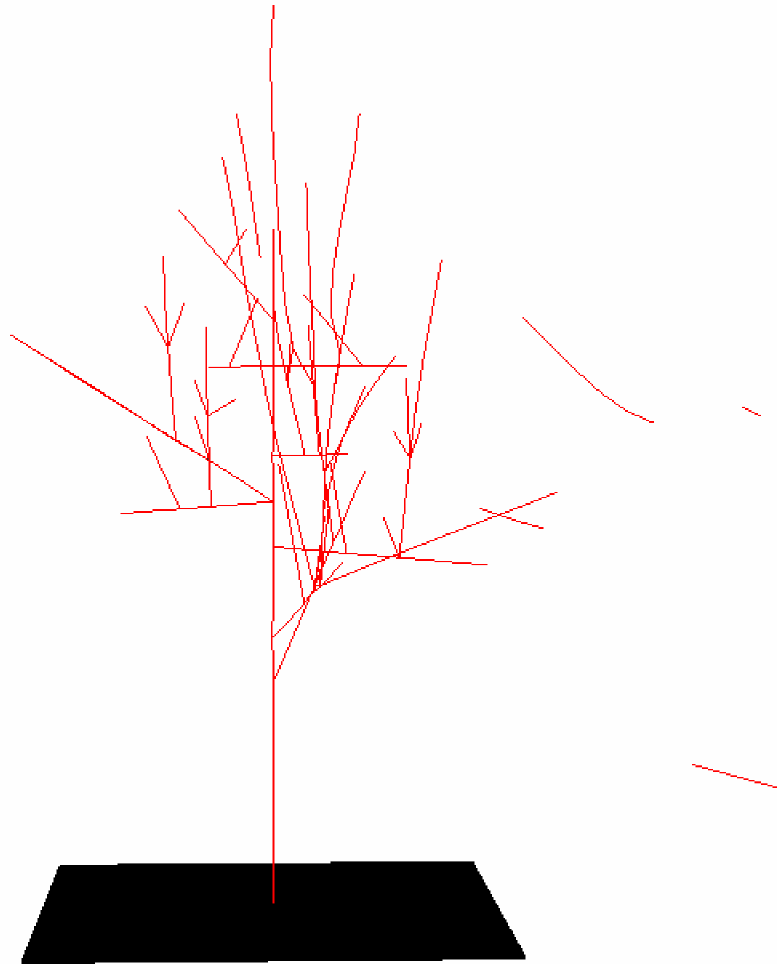

Figure 4 – ~80 mph Wind Effect on Oak tree

The current un-optimized developed application prototype is capable of simulating a tree with 130 branches, and run at a minimum of 60 frames per second. For an application running at 60 FPS we have determined that a time-step of 0.043 seconds performs the best where the tree branches react more realistically to global forces applied to them – future work will explore

time-steps that are FPS independent, but for the purpose of this research it is constant at 0.043 seconds for 60 FPS with standard gravitational force of 9.8g.

It was observed that the system was stable and all general structural branches are maintaining a relatively correct direction in relation to their parent branches. The tension on the root system was also visible when high winds were applied to the system.

CONCLUSION

The goal of this project was to fully simulate a tree in real-time with both external and internal forces acting on it. This not only requires simulating the forces that affect a single branch, but also the forces between branches. To achieve this multithreaded force application a finite-element system of particles and constraints was introduced that encompasses all the necessary interactions. With use of Verlet integration implicit application of force can be achieved by simply modifying the positions of particles in the system.

Future versions of this simulation software will include simulation of tree growth using L-System theories for generation of new branches, while dynamically manipulating the size of older branches. This physical manipulation of the tree object will also introduce unique circumstances for the branches such as knots and curves in the structures.

Also, a more realistic visual representation of the tree should be produced. This criterion will be achieved via a technique that for now we will call *Depth Mapping*. The general idea is to simulate the semi-cylindrical shape of a branch using a single billboard that is axis bound for rotation. Then with the use of GPU specific functionalities, routines will be developed to simulate depth and roundedness to the simple billboard quad. This rendering technique coupled with foliage generation using static particle systems should introduce the detail necessary.

Literature research in the area of tree simulation has determined that this approach is unique, and previous work done in the field of tree simulation has not fully considered the use of cloth simulation techniques in representing semi-rigid flexible bodies, such as trees with firm branches. We hope that this paper will further advance studies in the field of real-time simulation using finite-element systems, and provide a much more needed detail to an area being

advocated by physicists and programmers, where entire physics engines are developed that can simulate all forms of physical behaviors by simply implementing the most basic components – which are particles and constrains.

LIST OF REFERENCES

[1] Aitken M., Preston M. (2003), Grove: a production-optimised foliage generator for "The Lord of the Rings: The Two Towers". In: Proceedings of the 1st international conference on computer graphics and interactive techniques in Australasia and South East Asia. *ACM Press*, *New York*, pp 37–38

[2] Akagi Y., Kitajima K. (2006), Natural Phenomena Computer animation of swaying trees based on physical simulation. *Computers & Graphics* 30, pp 529–539

[3] Bloomenthal J. (1985), Modeling the mighty maple. In: SIGGRAPH '85: Proceedings of the 12th annual conference on computer graphics and interactive techniques. *ACM Press, New York*, pp 305–311

[4] de Reffye P., Edelin C., Françon J., Jaeger M., Puech C. (1988), Plant models faithful to botanical structure and development. In: SIGGRAPH '88: Proceedings of the 15th annual conference on computer graphics and interactive techniques. ACM *Press, New York*, pp 151–158

[5] Eberhardt B., Weber A., and Wolfgang S. (1996), A Fast, Flexible, Particle-System Model for Cloth Drapping. *IEEE Computer Graphics and Applications*

[6] Eischen J.W., Deng S., Clapp G.T. (1995), Finite-Element Modeling and Control of Flexible Fabric Parts.

[7] Jacobsen T. (2003), Advanced Character Physics, Article on *Gamasutra.com*.

[8] Jakulin A. (2000), Interactive Vegetation Rendering with Slicing and Blending. Short presentations at *Eurographics'2000*.

[9] Lander J. (1999), Devil in the blue-faceted dress: Real-time cloth animation. *Game Developer Magazine, September Issue*. pp. 25-30.

[10] Marshall D., Fussell D., Campbell III A.T. (1997) Multiresolution rendering of complex botanical scenes. *Canadian Human-Computer Communications Society*, pp 97–104

[11] Oppenheimer P. E. (1986) Real time design and animation of fractal plants and trees. In: SIGGRAPH '86: Proceedings of the 13th annual conference on computer graphics and interactive techniques. *ACM Press, New York*, pp 55–64

[12] Ota S., Tamura M., Fujita K., Muraoka K., Fujimoto T., Chiba N. (2003), 1=f b noise-based real-time animation of trees swaying in wind fields. *Proceedings of the computer graphics international*. p. 52–9.

[13] Pierce F. T. (1937), On the Geometry of Cloth Structure. *In Journal of the Textile Institute*, 28: 45 – 97

[14] Prusinkiewicz P., Lindenmayer A. (1990), The algorithmic beauty of plants, *New York, Ed. Springer-Verlag*.

[15] Remolar I., Chover M., Belmonte Ó., Ribelles J., Rebollo C. (2002), Real-Time Tree Rendering Technical Report, *DLSI*

[16] Sakaguchi T., Ohya J. (1999), Modeling and animation of botanical trees for interactive virtual environments. In: Proceedings of the ACM symposium on virtual reality software and technology. *ACM Press, New York*, pp 139–146

[17] Smith A.R. (1984) Plants, fractals, and formal languages. In: SIGGRAPH '84: Proceedings of the 11th annual conference on computer graphics and interactive techniques. *ACM Press, New York*, pp 1–10

[18] Stam J. (2006), A General Dynamical Solver for computer graphics. A presentation at the *Montreal International Game Summit*.

[19] Sugiyama A., Endo S., Arai J. (1995), Fluid dynamics. *Morikita*.

[20] Volino P., Thalmann N. M., Jianhua S. and Thalmann D. (1996), An Evolving System for Simulating Clothes on Virtual Actors, *IEEE Computer Graphics and Applications, September Issue*.

[21] Weber J., Penn J. (1995) Creation and rendering of realistic trees. In: SIGGRAPH '95: Proceedings of the 22nd annual conference on computer graphics and interactive techniques. *ACM Press, New York*, pp. 1-10

[22] Wesslén D., Seipel S. (2005), Real-time visualization of animated trees. *Visual Computer*, July 2005, Vol. 21 Issue 6, p397-405.