Realtime Forest Animation in Wind

Nimish J. Oliapuram
Computer Sc. & Engg.
IIT Delhi
New Delhi 110016 INDIA

Subodh Kumar[*]
Computer Sc. & Engg.
IIT Delhi
New Delhi 110016 INDIA

ABSTRACT

We present a system for interactive animation of trees in a wind-field. Wind forces are simulated by Navier-Stokes equations, solved in real-time using multi-processor CUDA architecture. The dynamics of the tree is also modeled and simulated in real-time using elastic body physics on CUDA. Due to the animation being entirely physics based, it is straightforward to design wind-fields by adding artificial sources and obstacles and have the animation respond. By using an efficient parallel algorithm we are able to animate over a hundred trees in real-time. We demonstrate animation of a forest of trees in a variety of wind conditions under interactive control of a user.

1. INTRODUCTION

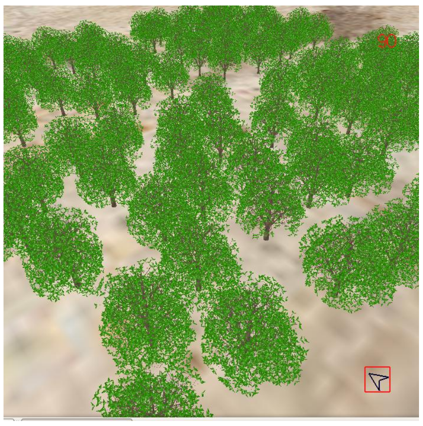

Figure 1: A forest scene (note the wind controller icon at the bottom right)

Trees are among the hardest to model in an outdoor scene. Nonetheless, trees and grass are also among the most essential ingredients of an outdoor scene. Growth models and L-systems are commonly used to create the basic structure of a tree. However, realistic animation of trees remains a challenge. Our focus in this paper is not on the actual shape of the tree but on its response to wind.

Visually convincing animation of trees in real time is hard because of the size of the problem. Tree shapes are geometrically complex with many thousand branches and leaves, Moreover, these branches are connected in a complex dynamic system that is hard to solve. Furthermore, the jittery movement of leaves is a high-frequency phenomenon and difficult to model and even approximate. As a result most animation algorithms require off-line pre-processing.

Ones that are able to efficiently animate trees generally rely on heuristic approaches, largely dispensing with the physics. These algorithms, in principle, generate random motion with the emphasis on controlling the randomness so that the generated motion matches the characteristics of some real animation.

Such heuristics-based approaches require a careful design of the heuristic for each scenario with different trees and wind patterns. Artful hit and trial may be required. On the other hand a physics-based approach lends itself to a natural framework that is able to directly model a variety of trees and wind conditions. This also leads to easier integration into existing animation work-flows. Furthermore, a physics-based approach lets us model the effect of the presence of the trees on the wind itself. We present the first interactive algorithm for physics based tree animation.

1.1 Our approach

We exploit the parallel architecture of modern graphics hardware (exposed through CUDA) to help our goal of real-time physics based tree animation. In addition to the hardware itself, the reasons for our efficiency is two-fold: a careful choice of the models that is suitable for parallel processing and an efficient parallel algorithm enabling us to achieve high parallelism from end to end, including wind simulation, tree dynamics, and rendering. Our main contributions include:

1. devising an end to end parallel algorithm suitable for CUDA
2. reducing or eliminating serial parts without introducing significant errors
3. exploring techniques to estimate physical properties used in the simulation

[*]Contact Email:subodh@cse.iitd.ac.in

1.2 Previous work

Previous approaches for simulating the dynamics of trees can be roughly categorized into full physics simulation [1, 15, 20, 8], heuristic design [22, 6, 21, 3] and hybrid approaches [16, 5]. Among these, [22, 6] harness GPU to speed up their systems.

Although we use Navier-Stokes fluid equations to generate the wind field, [19] employs the Lattice Boltzmann Model (LBM) instead, which is considered more parallelizable. However, that also requires smaller time time steps and lattice spacings. We have found the Navier-Stokes solution more stable for larger steps and thus faster overall.

[6] uses Euler-Bernoulli Beam Model [14] to compute bending of tapered branches. However, the results are only valid at equilibrium and do not actually apply in a dynamic environment. Their animation is driven by user provided 2D motion textures.

[21] introduces three kinds of level of detail representations. *Geometry LOD* is achieved by clustering branches and leaves. *Animation LOD* is achieved by deactivating joints of the tree. *Wind-field LOD* is achieved by generating a mipmapped texture for wind field.

[1] employs incompressible fluid Navier-Stokes equations for simulating the wind. They introduce the concept of a boundary-conditions map to build a resistance model for leaves and branches that enhances the speed of computation of the interaction of wind with the tree. They treat the leaves and branches as virtual resistive bodies. However, they employ a highly simplified dynamics model for the branch deformation.

The Navier Stokes based wind simulation used by our algorithm is similar in spirit to that of [1] and the tree dynamics model is inspired mainly by [15]. We have adapted their physics model for wind-tree interaction and devised the appropriate geometric algorithms suitable for the CUDA platform. This enables us to significantly improve the simulation speed and obtain interactive animations. For readers unfamiliar with the CUDA model, we briefly describe it here.

1.3 CUDA model

CUDA is a parallel computing infrastructure developed by nVIDIA to access GPUs (graphics processing units) as co-processors [11].

Execution model.

Computation is organized into *blocks* of *threads*, with the hardware capable of keeping thousands of thread contexts simultaneously. Each block runs concurrently on one Stream Multi-processor (SM). The underlying hardware may consist of many SMs executing independent blocks – there is no GPU synchronization across running blocks (the only synchronization is at the CPU level). A block can be organized into 1, 2 or 3 dimensional array of threads. This allows the thread identifier to be a 1, 2, or 3 dimensional entity, whichever is natural for the application. For example, for a 3D-grid solution of Navier-Stokes equation, 3D block fits. Threads in a block may be synchronized using barriers.

Multiple blocks form a *grid*. A grid may also be organized as 1 or 2 dimensional group of blocks. nVIDIA recommends hundreds of blocks to be scheduled on each SM concurrently. Each block of threads is further subdivided into groups of 32 threads: *warps*. A warp of threads execute in a SIMD [4] fashion and proceeds in lock-step except when threads execute different blocks of the if-then-else construct.

To parallelize our algorithms we need to design *kernels* executed by each thread and specify the configuration by providing the block and grid dimensions.

Memory Model.

CUDA has a non-uniform memory access model. *Registers* are the fastest; each thread gets its own private set. *Shared memory* can be as fast as registers when there are no bank conflicts or when all threads read from the same address. Each block gets its own private set. *Global memory* is two orders of magnitude slower than shared memory [11]. It is accessible from both the host (CPU) program and all active GPU blocks. It has the scope of the entire application. *Local memory* resides in global memory and therefore has the same performance overhead. However, like registers, it has the scope of a thread. It may be used for local variables, especially arrays. We have limited such variables to avoid using local memory.

Global memory coalescing.

Global memory supports 32, 64, and 128 byte transactions. Access by a thread are in smaller units. Simultaneous global memory accesses by half a warp (16 thread) can be coalesced into as few as a single memory transaction. Coalescing is important for good performance [10] but requires that [11]:

1. threads access 32, 64, or 128 bit data-types.

2. all 16 accesses are aligned, i.e., they in the same segment of, respectively, 32, 64 or 128 bytes (or twice for 128-bit accesses).

1.4 Tree generation

Although tree generation is not the subject of this paper, we briefly describe our implementation. For animation, one needs the complete structure of the tree. We employed a *Bracketed L-system tree*[12, 18] generator using a randomized recursive grammar.

Following is an example of a recursive grammar rule, in which each branch has 5 children, of which one is a continuation of itself:

$$F = G\,[R_1F]\,[R_2F]\,[R_3F]\,[R_4F]\,[F]$$

- G = Stem geometry & Translation matrix
- R_1, R_2, R_3, R_4 = Rotational matrices
- [= Push matrix
-] = Pop matrix
- F = Recursive rule

Everytime a recursion terminates (based on a random variable), leaves are generated. We gradually reduce the branch thickness and height at increasing recursion depths. We also add a random small rotation along the axis of the branch at each step. Finally, randomly choosing from a family of generation rules results in a more realistic looking hierarchy. We are able to generate a variety of trees as shown in Figure 2.

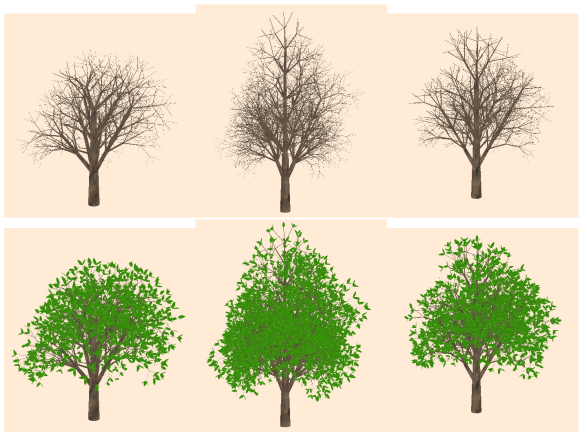

Figure 2: Examples of trees generated by the Bracketed L-system generator. Top row shows only the branches, bottom row has leaves added.

Tree model storage:.

One can choose to store the grammar rules and generate some part of a tree on the fly every time it is needed. Storage is low but the generation takes long. Alternatively, pre-generating the entire forest of trees leads to a large storage but a lower processing time. We instead store each tree in an intermediate form. Rather than storing the actual vertices and faces, we only store the branch dimensions and the hierarchy structure. The information stored is sufficient to derive the geometry of the tree and leaves later during dynamics simulation and rendering. More importantly, it also leads to a lower overall time for dynamics simulation using CUDA as we trade off slower memory bandwidth for more efficient in-parallel tree generation.

2. WIND FIELD GENERATION

Most heuristic approaches simplify the wind parameters by only considering the global wind direction and strength at any time. However, such an approach would not be able to easily generate complex wind patterns such as stormy weather, a passing tornado or a landing helicopter. Much hand tuning becomes necessary to generate desired wind textures.

We seek to know the wind velocity at each point in the 3D world to correctly simulate complex wind patterns. Instead of an artist designing such a *wind field*, we want to dynamically generate it depending on external forces or obstacles. We employ Navier-Stokes equations to sample these velocities on a 3D grid.

2.1 Navier-Stokes solution

We assume wind to be an *incompressible, inviscid, homogeneous* fluid. Incompressibility ensures that the resultant flux at any point is always zero, i.e., incoming flux matches the outgoing flux. This ensures that sources and sinks don't appear inside our wind field. The *inviscid* assumption is valid as the viscosity of air is negligible. The *homogeneous* assumption forces the density of the wind to remain uniform. This easily holds for a small domain area, say, less than a kilometer square.

Keeping these assumptions in mind we employ a stable Navier-Stokes solver to calculate the wind field. Solving Navier-Stokes equations have become synonymous with fluid physics simulation and have been widely used for various wind animations. We borrow the algorithm outlined in [17, 7, 2]. A brief description follows. The equations for inviscid fluid are:

$$\frac{\partial u}{\partial t} = -(u.\nabla)u - \frac{1}{\rho}\nabla p + f. \tag{1}$$

$$\nabla.u = 0, \tag{2}$$

where u is the velocity, p is the pressure, ρ is the density, and f is the external force. Eq. 1 is known as the *momentum equation* and Eq. 2 as the *incompressibility constraint.*

The pressure and the velocity fields that appear in the Navier-Stokes equations are in fact related. A single equation for u can be obtained by combining Eq. 1 and Eq. 2 as follows:

$$\frac{\partial u}{\partial t} = P(-(u.\nabla)u + f), \tag{3}$$

where P is an operator that projects any vector field onto its divergence free part.

2.2 Implementation issues

Navier-Stokes equations are solved discretely by considering an initial state $u_0 = u(x, 0)$ and marching through time with a time step of $\triangle t$. We start from the solution $w_0(x) = u(x, t)$ of the previous step and then sequentially resolve each term on the right hand side of Eq. 3, followed by a projection onto a divergent free field. The general procedure is outlined below,

$$w_0(x) \overbrace{\longrightarrow}^{advect} w_1(x) \overbrace{\longrightarrow}^{add\,force} w_2(x) \overbrace{\longrightarrow}^{project} w_3(x)$$

The solution at time $t+\triangle t$ is then given by the last velocity field, i.e. $u(x, t+\nabla t) = w_3(x)$.

Advection.

Artifacts in the fluid propagate according to the expression $-(u.\nabla)u$. This is modeled as a backward advection. That is, to obtain the velocity at a point x at the new time $t + \triangle t$, we back-trace the point x through the velocity field w_0 over a time $\triangle t$ at its "old" position: v'. The new velocity at the point x is then set to v'

$$w_1(x) = w_0(x - \triangle t.w_0(x)). \tag{4}$$

External forces.

If we assume that the force does not vary significantly in period $\triangle t$,

$$w_2(x) = w_1(x) + \triangle t.f(x, t). \tag{5}$$

Projection.

Projection keep the velocity field divergence free, according to Eq. 2. By Helmholtz-Hodge Decomposition Theorem, the vector field w_2 can be uniquely decomposed into the form.

$$w_2 = u + \nabla p, \tag{6}$$

where u has zero divergence: $\nabla.u = 0$ and p is the scalar field that corresponds to the pressure. This result lets us define the projection operator P as follows :

$$u = P(w_2) = w_2 - \nabla p. \tag{7}$$

To obtain the pressure field we apply the divergence operator to both sides of Eq. 6, to obtain

$$\nabla.w_2 = \nabla.(u + \nabla p) = \nabla.u + \nabla^2 p. \tag{8}$$

Enforcing the incompressibility constraint of Eq. 2, we get

$$\nabla.w_2 = \nabla^2 p, \tag{9}$$

which is a Poisson equation for the pressure of the fluid, also known as the *Poisson-pressure equation.*

We solve Eq. 9 using the Jacobi iterative method. More sophisticated methods such as conjugate gradient and multigrid methods converge faster, but we use Jacobi iterations because of its parallelizability.

Once we solve Eq. 9 for p, we can use w_2 and p to obtain the new velocity field $u(x, t + \triangle t)$ using Eq. 7.

Boundary conditions.

At the solid-fluid boundaries, we impose a *free-slip* boundary condition, which requires that the velocities of the fluid and the solid are the same in the direction normal to the boundary:

$$u.n = u_{solid}.n \tag{10}$$

Thus the wind can't flow into or out of a solid, but it is allowed to flow freely along its surface.

2.3 Controlling the wind field

User control of the wind field is through external forces. We propose two intuitive trackball based interfaces to control the wind patterns. In both the current wind direction and strength decide the magnitude and direction of the force added to the wind field.

The first method adds a Gaussian ball of the required force into the wind field. The size of the ball decides how widespread the impact is. Trees in the direct path of the wind are the most affected by the added force and the rest will have a decreasing effect. We can also add a random translation to the centre of the Gaussian ball to simulate turbulent wind. Figure 3 shows the changes in the wind field as we add wind in this fashion.

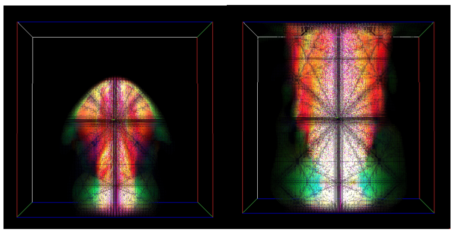

Figure 3: left: wind flowing from South to North; right: Wind after a while

The second method adds the external force to only the border voxels of the grid. This indirectly simulates external wind coming into the domain of interest.

2.4 CUDA implementation

We have discretized the wind domain into a 256x256x16 grid. x and y dimensions are larger mainly to cover a forest of trees spread over an area. Grid-based Navier-Stokes solver parallelizes well. We need to solve the same set of equations (see Section 2.2) for each voxel of the grid. CUDA allows the maximum block size of 512. We are able to use maximal blocks, choosing 16x8x4 threads per block. The grid dimensions to 16x32 blocks. We would like blocks to be as close to cubical as possible, i.e., 8x8x8, but setting the x dimension of the block to 16 results in better global memory coalescing as explained later in Section 1.3. The following kernels solve Navier-Stokes equations:

Advection implements backward advection velocity as mentioned in Section 2.2. We tri-linearly interpolate velocity values from the 8 nearest voxels to $(x - \triangle t.w_0(x))$ to find the velocity at any point. This allows us to use a rather sparse grid. Without this interpolation, if $|w_0(x)|$ is low and the nearest voxel to $(x - \triangle t.w_0(x))$ turns out to be x itself, the velocity would freeze.

External_Force adds velocity into the desired areas of the velocity field as mentioned in Section 2.2. The direction and magnitude of velocity is dictated by the trackball interface.

Divergence calculates the divergence of the velocity field after applying the external force.

Jacobi implements a single Jacobi iteration. We call this kernel in a loop to solve *Poisson-pressure equation* (Eq. 9). Our experiments suggest that 4 iterations are generally sufficient and there seems to be little improvement in the tree animation quality after this.

Projection eliminates the divergence in the velocity field as mentioned in Eq. 7.

Velocity, pressure and other temporary variables are all stored in the global memory as each step is in a different kernel. Combining the kernels does not help as all threads must finish each step before anyone can start the next. Data remains persistent in the global memory.

To maximize global memory coalescing, we store all values either as *float4*s (128-bit) or *float*s (32-bit). For storing values such as the velocity field, we actually require only 3 *float*s per voxel. However, using *float4*s results in better global memory coalescing. The fourth component can be used for other auxiliary data.

To meet the alignment requirement, we set the x dimension of the block size of the fluid simulation kernel to 16. This results in each thread in a half warp making aligned global memory accesses from successive memory locations and therefore getting coalesced.

3. BRANCH DEFORMATION DYNAMICS

To correctly simulate the deformation of branches, we must compute for each branch, the influence its parent as well as the children branches have on it. For accurate computation of the cause-and-effect relationships among all connected branches, techniques such as the finite-element method are used [20, 8]. However, computational costs are large and these do not easily lead to efficient rendering later.

We adapt the approach of [15], in which each branch (long branches are subdivided into smaller ones) is modeled as a rigid body. Like them we assume that they are fixed at the parent branch end and may only rotate with respect to that end. Thus we are able to employ their equations of motions occasionally modified as described below. The force that propagates from the parent branch is expressed by making the fixed end of the child branch follow the movement of the the parent branch. The restoration force that propagates from the child branch to a parent is appended to the external force at the parent branch.

We have decomposed the simulation of dynamics into two phases (to map it well to CUDA). *Dynamics calculation phase* begins from the tips of the tree and proceeds to the root. *Integration phase* proceeds from the root to the tips of the tree and integrates the movements of all branch segments and calculates the final position and orientation of each branch.

3.1 Dynamics calculation

We model each branch as a tapered cylinder. The equation of motion of a cylinder about a fixed end O is given as follows :

$$N = \frac{ml^2}{3}\frac{d\omega}{dt}. \tag{11}$$

where N is the moment of the force acting on the cylinder, ω is its angular velocity, m its mass and l its length. For short tapered cylinders, this approximately holds.

Let F be the sum of all forces applied to a segment. Then, the moment N can be expressed by $N = F \times c$, where c is the vector from the fixed end to the center of gravity of the branch. Hence, we get

$$F \times c = \frac{ml^2}{3}\frac{d\omega}{dt}. \tag{12}$$

3.1.1 Forces acting on a branch

The force F consists of the force applied by wind F_{wind}, the restoring force K, the axial damping force R and the back-propagation force from the child branches T, as in [15]. We also neglect gravity, which we incorporate in the stationary tree (as computed by the L-system). We also neglect the viscous resistance of air as it is significant only for lightweight objects, unlike trees.

$$\therefore F = F_{wind} + K + R + T \tag{13}$$

Wind force.

We sample the external force (interpolated from the windfield) for each branch segment at its centroid as the segments are small. Integration of true forces over the segment leads to no noticeable improvement in quality but a significant loss in speed. We integrate by simply scaling the centroid force by the surface area of the branch. Therefore, we get

$$F_{wind} = S_f \sigma v, \tag{14}$$

where S_f is the surface area in contact with air, σ is a constant considering the viscosity coefficient of air and v is the velocity of the wind.

Restoration force.

Elastic restoration force tries to restore the original position of a branch segment with respect to its parent segment. It is proportional to the angular displacement from the original orientation.

$$K = k(\theta - \theta'), \tag{15}$$

where k represents the rigidity of the branch and is a constant determined by the thickness of the branch, θ is the original orientation and θ' is the current orientation.

Axial damping force.

Due to strong binding forces which exist in branch segments, a force proportional to the square of the velocity damps the movement of the branches. It plays a role in gradually suppressing the vibrations of the branches caused by the external force.

$$R = -\mu\omega\left|\omega\right|, \tag{16}$$

where μ is a constant determined by the thickness of the branch.

By itself this axial damping force can be large and instead of damping the motion can even reverse it. To ensure that the negative acceleration caused by the axial damping force would not exceed ω, we clamp R:

$$|R| = min(\mu\omega^2, I\omega). \tag{17}$$

Back-propagation force.

This reaction force models the forces that propagate from the child segments to the parent.

$$T_{i-1} = -\sum k_i K_i, \tag{18}$$

where i corresponds to each of the children of segment $(i-1)$ and k_i is the propagation coefficient of the force and it is a constant determined as

$$k_i = k_c \frac{Th_i}{Th_{i-1}}, \tag{19}$$

where k_c is the fixed propagation coefficient and Th is thickness of the branch segment.

Note that if the child segments have no angular displacement, their restoration force would be zero and the back-propagation force would also remain zero. We exploit this.

3.1.2 Equations of angular motion

We take the classical coordinate system (x, y, z) as our basis. Hence, θ, ω and $\alpha = \frac{d\omega}{dt}$ all have three components. The equations of angular motion are:

$$\theta' = \theta + \omega(\triangle t) + \frac{1}{2}\alpha(\triangle t)^2, \tag{20}$$

$$\omega' = \omega + \alpha(\triangle t). \tag{21}$$

Once, we have calculated the new θ', we can find the actual orientation of the branch by rotating the branch about its fixed end by an angle of $|\theta'|$ about the axis along the direction of $\hat{\theta'}$.

3.1.3 Branch parameters

The restoration constant k and the axial damping constant μ are proportional to the average thickness of the branch r. This is because we expect thicker branches to provide more restoration force and thereby result in less bending as compared to thinner stems. Similarly, we expect thicker branches to damp vibrations more effectively than thinner stems. However, we could find no literature on the exact dependence k and μ had on the thickness of the branch, or the wood material.

Hence, we tested all combinations of k and μ for various branch dimensions and computed resulting deformations. Deformations that matched one observed in a population of real trees around our campus were saved as candidates. Once, we collected a database of acceptable combinations for various branch dimensions, we regressed a curve to obtain the following dependence of k and μ on r (for all branch lengths).

$$k \propto r^{2.5}, \tag{22}$$

$$\mu \propto r^{3.5}. \tag{23}$$

This is what we have used in our simulations.

3.2 CUDA implementation

Parallelizing of the dynamics model is difficult because trees are hierarchical and have inter-dependencies between the parent and child segments. In the *dynamics calculation* phase, a branch's computations are based on the results of its children's computations. And in the *integration* phase, a branch's computations require the results of its parent's. We resolve these dependencies as follows.

Dynamics calculation.

The dependence on the child segments is only required for calculating the back-propagation forces as mentioned in Section 3.1.1. We eliminate this dependence by using the restoration forces of the child segments from the previous time step. This is based on the observation that the restoration forces vary slower than the other forces and that their effect on the final positions is small. Our experiments show that there is no observable difference in the resulting animation quality.

We now can employ as many threads as there are branches and leaves. Leaves are simulated as pseudo-branches as mentioned in Section 4 and we use a single kernel for both. All branch parameters, such as height, radius, original transformation matrix, etc. are initially passed to the GPU and stored in the global memory. Similarly, current orientation, current angular velocity, current transformation matrix, etc. are also stored in the global memory and are updated at each step.

Integration of movements.

The dynamics calculation phase provides the local transformation matrix of each branch. To calculate the position of a branch, we must calculate the cumulative transformation matrix of the branch (from the root), meaning the parent's cumulative transformation must be known first. Unlike restoring forces, one simply cannot use the motion of the parent in the last frame or the tree would become disjointed.

We process the hierarchy breadth first. We calculate the cumulative transformation matrix of all the branches/leaves at one level concurrently. To do this efficiently and with memory coalescing, we store the branches and leaves in the increasing order of depth, root first. We also maintain the starting index of the branches/leaves in each level along with the total number in each level.

Thus, if the depth of the forest is d, we call the integration kernel d times and each time we pass the starting index and the number of branches/leaves in the corresponding level as an argument to the kernel. The kernel implements the required matrix multiplication operations by using as many threads as there are branches/leaves in each level.

A note about matrix multiplications: both dynamics calculation and the integration kernels require (4x4) matrix multiplications where both input and output lie in the global memory. We speed this computation up by staging the matrices in the shared memory and performing the multiplication there. For this purpose, we set the block size to 64 threads. This leaves 256 bytes per thread as the total shared memory available is only 16KB.

4. LEAVES AS PSEUDO-BRANCHES

Capturing the jittery movement of leaves in an animation is difficult. Most heuristic approaches (even [15] resorts to heuristics) model their movement using a combination of high-frequency sinusoidals augmented with certain randomness. We instead transfer the branch deformation model to leaves as well thus employing a unified framework and kernel. This also leads to more parallelism as there are more blocks to cycle through (see Section 3.2).

We model a leaf as a flat polygon (rather than a cylinder). Therefore, we assume that only the velocity component along the normal of the face of the leaf applies to the dynamics of the leaf. This ensures that each leaf only has a single degree of freedom, i.e. it only rotates about the branch.

In essence, we try and model the leaves as very thin flat branches. However, to capture the jittery movement of leaves, we also make the following changes to the parameters previously used for branches.

- **Moment of Inertia** I **:** We set I to a value much lower than of even the thinnest branches to model the fact that they are very light and therefore would be affected by even light wind.
- **Axial damping constant** μ **:** We set μ to a low value to ensure that the damping of vibrations is minimal. This results in the jittery motion we desire for leaves.
- **Restoration constant** k **:** We set k to a value slightly higher than of the thinnest branches, to balance the effect of using very low values of I. This ensures that the leaves flutter significantly in strong winds but do not bend excessively.

5. RENDERING

In traditional tree simulation, physics is the bottleneck by far. Due to the speedups achieved by our algorithm, however, rendering quickly becomes the bottleneck. Thus efficient rendering becomes important.

We use OpenGL display lists: one for rendering a leaf and one for a branch (tapered cylinder). We make the assumption that all leaves have the same size. However, since branches have varying sizes we perform a scaling per branch. Similarly, the look of branches are changed by using the same texture but rotated from branch to branch.

We also employ *levels of detail* (LOD) [9], computing multiple LODs for the branches and the leaves. Depending on the branch thickness and tree distance from the viewer, each tree calls the appropriate sequence of display lists. As can be seen in the accompanying video, we have chosen to be conservative with the simplification so the image quality does not suffer.

We also pay close attention to data transfer between CUDA physics simulation and the OpenGL rendering engine. The cumulative transformations computed by CUDA (and stored in global memory) are bound to *dynamic* OpenGL texture.

The CPU host does not copy matrices nor performs matrix multiplications per branch – it only calls the appropriate display list for each tree based on the distance. The desired matrix per branch (and leaf) is available as a texture (constant memory is used) to be fetched by the vertex shader. The index into this texture needs to be passed per branch. This index and additional parameters such as scaling factors and texture translations are themselves passed in a second OpenGL texture, which is indexed by the instance ID.

Please note that we do not implement this direct transfer for the CPU implementation as there is no CUDA. However, for the CUDA implementations, dynamic textures and shaders lead to a significant speedup as shown in Section 6.

6. RESULTS AND ANALYSIS

To illustrate the benefits of parallelization, we built two implementations of each component of the entire physics engine - a serial implementation that runs completely on the CPU and a parallel CUDA implementation. We compared both of them on the following four platforms (all linux based):

Serial runs on a 3 GHz Intel Xeon E5450 quad core machine with 4 GB memory. This is the CPU used for all implementations.

9800 uses an affordable nVIDIA 9800 GTX graphics card, which performs both the CUDA-based physics computations and OpenGL rendering. It has 512 MB global memory and has 16 multiprocessors (128 cores) and has a clock rate of 1.7 GHz.

280 uses a high end gaming graphics card used for both CUDA and OpenGL. It has 1GB global memory and 30 multiprocessors (240 cores) with clock rate of 1.3 GHz.

Tesla uses professional-grade Tesla T10 for CUDA-based physics computations and a low-end graphics device, Quadro NVS 290, for rendering. Tesla T10 has 4 GB global memory and 30 multiprocessors (240 cores) with a clock rate of 1.3 GHz. NVS 290 has 256 MB with 2 multiprocessors (16 cores) and a clock rate of 0.92 GHz. On this platform rendering and simulation happen on different GPUs.

6.1 Wind-field generation

Total time taken for each implementation of the Navier-Stokes computations is listed in Table 1. We achieved over 11x speedup on the 9800 GTX platform and a 36x speedup on 280 (Tesla is similar).

Serial	**9800**	***speedup***	**280**	***speedup***
520	48	11x	14.1	36x

Table 1: Comparison of overall time taken (in ms) for fluid solver

6.2 Dynamics model

For dynamics testing, we use a tree with 1597 branches and 3603 leaves (5200 components). We compare each implementation for varying number of trees as shown in Table 2.

#Trees	**Serial** (ms)	**9800** *speedup*	**280** *speedup*	***Tesla*** *speedup*
1	3.1	3.56x	2.04x	5.74x
4	12.25	6.13x	3.71x	8.63x
9	25.4	7.13x	6.68x	9.27x
16	49	8.55x	8.91x	10.43x
25	77	8.92x	9.86x	11x
100	459	14.02x	17.06x	17.24x

Table 2: Speedup in dynamics model computations

Although there is a significant speedup due to the parallelization, because of level-dependence the overall speedup is lower than for the fluid solver.

However, notice the sub-linear increase in computation times as we increase the number of trees. The serial implementation is over 100 times slower for 100 trees than it is for a single tree. On the other hand, 9800 GTX is only 38 times slower and Tesla is 49 times slower. This is because we calculate the dynamics of all branches/leaves in parallel across the tress. For more trees, we exploit more parallelism, especially at the lower levels (near the root).

6.3 Rendering optimizations

Table 3 compares the frame rates (on 9800 GTX) for rendering varying number of trees (no physics simulation).

#Trees	**Vanilla**	**DL**	**LOD Range**	**DT Range**
1	152.3	258	258 - 299	782 - 1400
4	36.7	52	54 - 58.4	270 - 360
9	13.8	18.9	20 - 20.8	120 - 156
16	7.7	11.2	11.3 - 11.7	70.5 - 93
25	5.2	7	7.1 - 7.4	45.8 - 60
100	1.2	1.8	1.8 - 1.9	11.6 - 14.6

Table 3: Effect of Display Lists (DL), LOD and Dynamic texturing (DT) on framerates (fps).

The *Vanilla* column has the framerates obtained rendering the trees in the immediate mode. For LOD and Dynamic Texturing we have shown the range of frame rates observed. There is an improvement from Vanilla to display-lists to LOD to dynamic texturing. Note that LODs (organized as display lists) do not yield interactive framerates for more than 16 trees. However, with the use of Dynamic Textures and shaders, we get a 5-8x improvement.

6.4 Overall results

The comparison of the framerates of the entire tree animation, including the physics computations and the rendering, and the performance benefits of parallelization using CUDA are shown in Table 4.

The CPU implementation barely runs 2 frames per second even for a single tree. Physics computations are the bottleneck. 280 GTX achieves rates of 63 fps on the same machine. This highlights the benefit of parallelization. Further, the GPU implementations scale better.

Figures 4 and 5, show the breakups of the times taken by the fluid solver, the dynamics model and the rendering as we vary the number of trees, keeping the grid size fixed. Thus for a large number of trees, rendering them becomes a bottleneck.

Among the physics based methods, ours is the first interactive system. Other GPU based methods have been reported but they are all heuristic based and we are still competitive. For example, [22] is able to render nearly 10 frame a second animating 100 trees on 9800. In contrast, the quality of [6] appears better but they report 32 fps using an nVIDIA 8800 GTS with only 4 trees (it was hard to replicate their result in our environment). [3] manages to perform heuristic-based CPU simulation of a simple tree at 30 fps.

7. CONCLUSIONS

#Trees	Serial (fps)	9800 *speedup*	280 *speedup*	Tesla *speedup*
1	1.9	10.7x	33.2x	27.4x
4	1.8	10.7x	28.3x	23.6x
9	1.6	11.1x	26.6x	20.5x
16	1.5	10.9x	23.4x	16.7x
25	1.3	11.3x	22.3x	14.6x
100	0.7	11.4x	16.3x	9.3x

Table 4: Comparison of framerates of the overall tree animation (physics + rendering), for varying number of trees

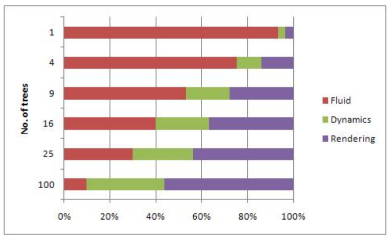

Figure 4: Percentage breakup of the time taken for the serial benchmark, for varying no. of trees.

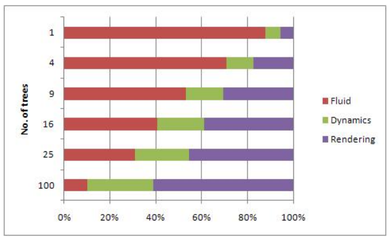

Figure 5: Percentage breakup of the time taken for the 9800 GTX benchmark, for varying no. of trees.

We have successfully demonstrated that physics-based approaches can be used for animating trees without resorting to heuristics. We achieve this by moving the entire physics computations - the Navier-Stokes fluid solver as well as the dynamics model - onto the GPU, using CUDA. We have shown that we obtained considerable speedups in the process, enabling us to achieve interactive animations. Furthermore, the CPU load goes down leaving it to focus on other parts of the application like AI or user interaction.

We have shown that numerical solution to Navier-Stokes equations parallelizes well. We have also shown that the dynamic part can parallelize well by breaking the hierarchical dependencies. Further rendering improvements are necessary to interactively animate even larger forests. Incorporating levels-of-detail in dynamics computation should further reduce the physics time as well.

Finally, our results should encourage attempts to transfer other physics-intensive simulations to GPUs.

8. REFERENCES

[1] Y. Akagi and K. Kitajima. Computer animation of swaying trees based on physical simulation. *Computer and Graphics 30*, pages 529–539, 2006.

[2] K. Crane, I. Llamas, and S. Tariq. Real-time simulation and rendering of 3d fluids. *GPU Gems 3*, 2007.

[3] L. Ding, C. Chongcheng, T. Liyu, and W. Qinmin. Geometrical shapes and swaying movements of realistic tree: design and implementation. In *VRCAI '09: Proceedings of the 8th International Conference on Virtual Reality Continuum and its Applications in Industry*, pages 295–302, New York, NY, USA, 2009. ACM.

[4] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(C):948–960, 1971.

[5] T. Giacomo, S. Capo, and F. Faure. An interactive forest. *Eurographics Workshop on Computer Animation and Simulation*, pages 65–74, 2001.

[6] R. Habel, A. Kusternig, and M. Wimmer. Physically guided animation of trees. *Eurographics*, 28(2), 2009.

[7] M. Harris. Fast fluid dynamics simulation on the gpu. *GPU Gems*, 2004.

[8] X. Hu, W.Tao, and Y. Guo. Using fem to predict tree motion in a wind field. *Journal of Zhejiang University*, 9(7):907–915, 2008.

[9] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, , and R. Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann.

[10] nVIDIA CUDA best practices guide, version 3.0.

[11] nVIDIA CUDA programming guide, version 3.0.

[12] P. Prusinkiewicz and A. Lindenmayer. The algorithmic beauty of plants. *Springer-Verlag*, 1990.

[13] nVIDIA CUDA reference manual, version 3.0.

[14] J. Roy R. Craig and A. J. Kurdila. *Fundamentals of Structural Dynamics*. Aeronautical Engineering Books, 2e edition.

[15] T. Sakaguchi and J. Ohya. Modeling and animation of botanical trees for interactive virtual environments. In *VRST '99: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 139–146, New York, NY, USA, 1999. ACM.

[16] O. Shin, T. Fujimoto, M. Tamura, K. Muraoka, and N. Chiba. A hybrid method for real-time animation of trees swaying in wind field. *The Visual Computer*, pages 613–623, 2004.

[17] J. Stam. Stable fluids. *SIGGRAPH*, 1999.

[18] P. Visser, L. Marcelis, G. Heijden, J. Vos, P. Struik, and J. Evers. 3d modelling of plants: a review. *Plant Research International*, pages 7–8, 2002.

[19] X. Wei, Y. Zhao, Z. Fan, W. Li, S. Yoakum-Stover, and K. A. Blowing in the wind. *Eurographics*, 2003.

[20] H. Xiaoyi. Dynamic finite element based animation of tree swing in the wind. *JOURNAL OF COMPUTER AIDED DESIGN AND COMPUTER GRAPHICS*, 19(9):1166–1171, 2007.

[21] L. Zhang, C. Song, Q. Tan, W. Chen, and Q. Peng. Quasi-physical simulation of large-scale dynamical forest scenes. *Computer Graphics International*, pages 735–742, 2006.

[22] R. Zioma. Gpu-generated procedural wind animations for trees. *GPU Gems 3*, 2007.