Physically Based Modeling of Interactive Plants

Undergraduate Honors Thesis

Bruno Li
Department of Computer Science
University of Texas at Austin
tsornin@cs.utexas.edu

Supervising Professor: Donald Fussell

May 4, 2012

Abstract

Producing realistic vegetation is an important challenge in interactive applications. Most modeling efforts have focused on realistic graphics, neglecting the distinctive soft-body motion of vegetation. The resulting trees approach photorealism, but stand still like statues, or at best exhibit an unconvincing swaying motion. This thesis studies an approach to tree animation using soft-body physics techniques, as well as a framework to automatically build such simulations from an L-system grammar. The resulting trees are interactive, exhibit realistic motion and are fast enough to be used in real-time. The physics simulation supports arbitrary external forces, and also allows for easy addition of game logic: as an example, rain and fire effects affecting trees were built on top of the simulation.

Contents

1 Introduction

Trees are a widespread landscape feature and a key element of immersion in interactive applications. The need for realistic trees is an outstanding problem in the development of real-time applications because of their geometric complexity and ubiquity. Conventionally, there have been two concerns in approaches to visually convincing trees: rendering and animation.

Many techniques have been developed to render realistic trees at real-time speeds [8]; however, these techniques focus mostly on rendering, neglecting animation. The resulting trees look better in screenshots than in interactive settings, where, lacking animation, they stand still like statues. Trees without animation are suitable for background scenery, but do not hold up well under close scrutiny.

One approach to procedural generation of tree animation is heuristic: assign each branch some sinusoidal motion of varying amplitude and frequency [1, 11]. This generally produces an unconvincing swaying motion. Instead of using heuristics, complex tree structures can be simulated with physical models: Habel [4] combines high-quality rendering with physically-based animation to produce visually convincing results.

Reproducing the distinctive soft-body motion of plants is the key to visually convincing trees, and applicable even in non-photorealistic settings. However, in many physically-based approaches [4, 13], the motion of the tree is motivated only by gravity and wind forces. Optimizing tree motion around these forces speeds up the simulation, but precludes external forces: interaction is limited to a few specific types of forces.

Even with rendering and animation, we only have trees that serve as scenery, and cannot be interacted with. A more difficult problem is allowing arbitrary interactions to affect the tree, while still retaining realistic motion. Yet this is what we ultimately want: tree objects which can bend, break, and respond realistically to a variety of external forces and stimuli, including (but not limited to) gravity, wind, rain, and fire. This thesis presents such a system for a full simulation approach: the resulting trees are first-class game objects with many capabilities suitable for interactive foreground objects. These tree objects are fully destructible and react to arbitrary external forces and collision. With the support of a physics system, it is also easy to add additional behavior and special effects.

There are two key ideas underlying our method. Our first observation is that all tree animation techniques are attempts to reproduce the distinctive swaying motion of trees, which are not entirely rigid, but pliable. Instead of using rigid-body physics, we adapt soft-body physics techniques to simulate our tree dynamics. Typically, soft-body physics are used to produce simulations of textiles such as string and cloth; however, the same techniques can also be used to simulate semi-rigid objects, such as trees. The second observation is that we can use the outputs of L-systems, which produce tree geometry, as inputs to our physical simulations. This gives two notable advantages: the descriptive power of L-systems allows us to quickly initialize simulations of a wide variety of tree shapes, and the fractal nature of the L-system output allows us to easily adjust the level-of-detail of our simulation.

There are three major steps to our approach:

- Geometry: In Section 2, we generate tree geometry using L-systems: this produces many different tree shapes to work with. L-systems as developed by Lindenmayer [9] only produce tree geometry: we augment the tree geometry generation process to produce additional data for the physical simulation.

- Dynamics: Section 3 describes our method to map tree geometry to soft-body physics simulations. To simulate the semi-rigid dynamics of a tree, we develop a novel physics object idiom, the "angular constraint", implemented using only the existing primitives of soft-body physics simulations.
- Effects: Finally, with tree dynamics in place, Section 4 discusses some possible special effects: fire, rain, and wind. We present a novel general-purpose fire propagation model, which is not limited to just modeling burning trees, but suitable for creating various visually convincing flammable objects.

To simplify development, our work was done in two dimensions: a transition to a three-dimensional simulation would be straightforward except for some considerations of rotation and orientation, discussed in Section 5, the conclusion.

2 Generating tree geometry

In this section, we examine L-systems, a type of formal grammar used to describe biological growth, and commonly used to generate plant-like graphics. We adapt L-systems to generate tree geometries suitable for input to a physical simulation.

2.1 L-systems

L-systems are a parallel rewriting system used by Lindenmayer [9] to model the grown of organisms, most notably plants. In a formal grammar system, production rules are applied one at a time. As a subset of formal grammars, parallel rewriting systems restrict the form of their output strings: production rules are applied in iterations, and as many rules as possible must be applied every iteration.

This parallel application of production rules models the simultaneity of plant growth, and the recursive nature of the production rules models the fractal nature of plants. The interpretation of the output strings as "turtle graphics"[1] commands produces plant-like graphics[2].

Figure 1: Outputs of a variety of L-systems.

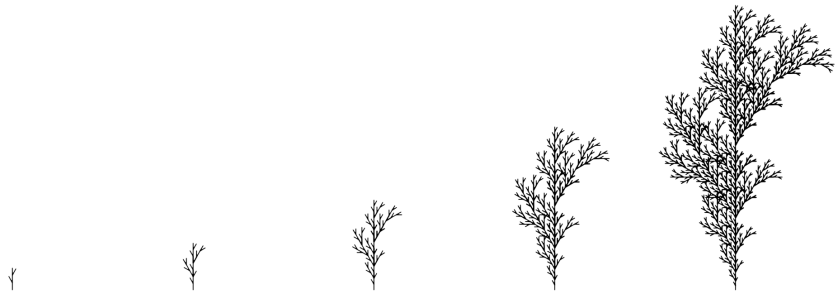


Figure 2: The output of successive iterations of an L-system.

We derive two benefits from using L-systems to generate our tree geometry:

- The descriptive power of L-systems allows us to quickly generate many different tree geometries. Figure 1 shows geometries produced by several different L-systems. This can be a thorough test of our dynamics method: we should be able to simulate a reasonably wide variety of tree shapes.
- The recursive nature of L-system rules produces fractal-like forms. This self-similarity provides for an interesting take on procedural multiresolution. Successive iterations of an L-system produce

[1] A method for producing graphics described as tracing actions of a cursor (the "turtle") with a location and an orientation.

[2] Figures 1 and 2 were rendered with http://nolandc.com/sandbox/fractals/.

similar but increasingly detailed outputs, as seen in Figure 2. Our simulation will be built according to these outputs, so we can scale down the dynamics to decrease the cost of the simulation and still retain a similar tree shape simply by using fewer iterations of the same L-system.

2.2 Adapting L-systems to physical simulation

The original interpretation of L-system strings as turtle graphics commands only produces plant-like graphics, which is necessary but not sufficient data for our physics simulation. In this section, we will modify each L-system symbol to generate data for our physical simulation (such as branch masses) in addition to tree geometry.

L-systems describing branching trees[3] produce strings containing six different symbols: `F`, `-`, `+`, `X`, `[`, and `]`. Each of these symbols is meant to be interpreted as a turtle instruction. However, we wish to interpret these symbols as input to a physical simulation: to do this, we will tag each symbol with additional information.

2.2.1 "Forward"

The `F` symbol instructs the turtle to move forward and create a line, which represents a stem of a plant. This is sufficient for producing plant-like graphics: however, without any notion of "line thickness", i.e. mass, we do not have enough data to build a proper simulation. If we assigned each internode of a tree an equal weight, the simulated tree would become top-heavy and collapse. We have to modify the L-system to produce masses for each stem: trunk-stems are the heaviest, bough-stems average, and twig-stems the lightest.

How do we know what masses to assign to each stem? Eloy [2] offers a structural explanation for an old observation made by Leonardo da Vinci that "all the branches of a tree at every stage of its height when put together are equal in thickness to the trunk" [7]. A study of tree dynamics is not one of our goals: all we want is to reliably pick masses small enough to avoid top-heavy collapse and large enough to retain some swaying inertia.

The solution is to exponentially decrease mass going upwards, mirroring the fractal form of the tree. Accordingly, we will describe the mass of a branch relative to its parent branch: we give the `F` symbol a parameter, its *mass multiplier*.

In previous systems, tree growth is simulated with the rule `F` $\rightarrow$ `FF`. This produces long branches that actually consist of multiple `F` symbols. We avoid this since each `F` symbol corresponds to a constant number of physics objects (see Section 3.2). Instead, we will ask the physics engine to somehow increase the length of the physics object directly.

The growth rule also causes stems from any iteration to be twice as long as stems from the next iteration. Since we have removed this mechanic, we substitute by giving the `F` symbol another parameter, the *length multiplier*, which describes the length of a branch relative to its parent branch.

[3] L-systems containing the '`[`' and '`]`' symbols are known as bracketed L-systems [9].

2.2.2 "Turn"

The `+` and `-` symbols instruct the turtle to turn left or right; the angle turned is defined on a per-L-system basis. While this keeps the branching angle consistent, we choose to control this angle manually.

Instead of using two different symbols, our turning instruction will be the symbol `T`. We give it one parameter, its *angle*.

2.2.3 "Node"

The examples of node-rewriting L-systems in [9] only use a single node rewriting rule, such as `X` $\rightarrow$ `F[+X]F[-X]+X`. Since this recursive rule is the only rule (besides the growth rule), the resulting tree is very strongly self-similar. We will allow for different types of nodes: this reduces the mandatory self-similarity in favor of state-machine-like control over the growth of the L-system.

Our node instruction will be the symbol `N`, taking one parameter, the *name* of the node.

2.2.4 "Stack operations"

The bracket symbols `[` and `]` denote a branching event of the tree. They are implemented as a save and restore of the turtle state, respectively.

A turtle's state is its position and orientation. Since we added parameters to the `F` symbol which refer to a "parent branch", we will also have to store some physics object reference in the turtle state.

2.2.5 Modified L-system

Figure 3 compares the original L-system syntax with our modified L-system syntax, which appears more verbose because of added parameters. By convention, we name our initial rule "start".

$n = 7, \delta = 20°$
`X`
`X` $\rightarrow$ `F[+X]F[-X]+X`
`F` $\rightarrow$ `FF`

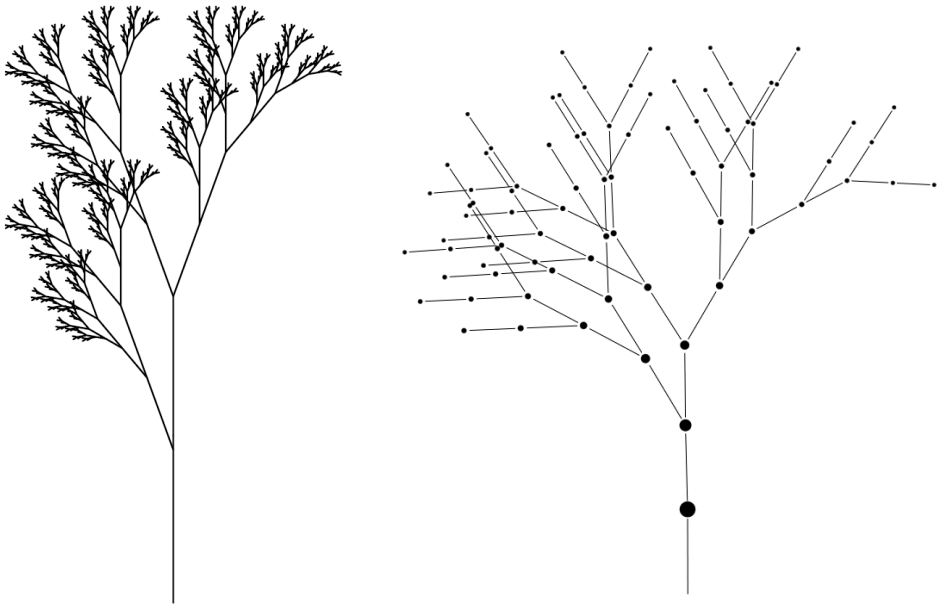
(a)

`start` $\rightarrow$ `F 0.9 0.5 N zero`
`zero` $\rightarrow$ `F 0.9 0.33 [ T 30 N zero ] F 0.9 0.33 [ T -30 N zero ] T 30 N zero`

(b)

Figure 3: Comparison of original and modified L-system syntax.

Figure 4 compares the original L-system output with our modified L-system output. Note that in the original output, each branch segment is half the length of its parent branch due to the growth rule `F` $\rightarrow$ `FF`. In our modified output, each branch segment is only a little shorter than its parent branch: we control the relative branch length with the length parameter of the `F` symbol. Branch masses are indicated by the size of the dots on ends of each branch segment: we can see the exponential decrease in mass caused by the mass parameter of the `F` symbol.

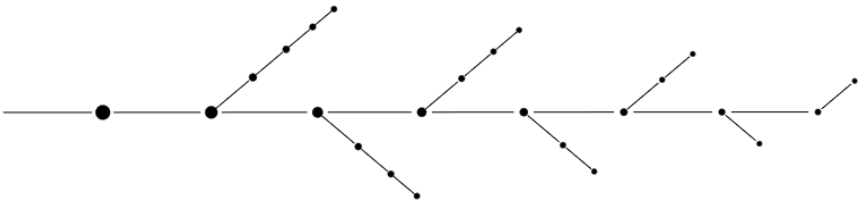


(a) (b)

Figure 4: Comparison of original and modified L-system output.

2.3 Bamboo

In this section, we design a second tree with a very different shape using our modified L-system. Figure 5 shows a bamboo tree output and the rules used to produce it. Despite the disparity between the shape of the bamboo plant and the branching plant in the previous section, we should be able to simulate both equally well.



```
start → F 1.0 0.5 N left
left → F 0.98 0.5 [ T 40 F 0.5 0.1 N one_a ] N right
right → F 0.98 0.5 [ T -40 F 0.5 0.1 N one_a ] N left
one_a → N one_b
one_b → F 0.95 0.5 N one_a
```

Figure 5: Modified L-system syntax producing a bamboo tree.

Without the auxiliary shoots, the bamboo is essentially just a stick. This is reflected in the portions of the `left` and `left` productions outside the brackets. We could produce a stick with just a single rule (`stick` → `F 0.98 0.5 N stick`), but we need separate `left` and `left` rules to start off the alternating pattern of auxiliary shoots. These small branches are given a small mass multiplier (0.1) so they do not significantly affect the with the weight of the tree. The `one_a` and `one_b` rules are an abuse of our named nodes as a state machine, causing the auxiliary shoots to only grow every other iteration.

3 Simulating tree dynamics

We developed a small variety of tree geometries in Section 2. Now, we describe a method to build a physics simulation with a structure corresponding to a given tree geometry.

3.1 Soft-body physics using Verlet integration

Jakobsen [5] presents an approach to soft-body simulation oriented toward interactive use, emphasizing the two goals of speed (real-time computation) and "believability" (a realistic feel as well as stability) at the expense of true physical accuracy. The method is based on two components, particles and distance constraints. Particles act as point masses, and constraints act as stiff springs between pairs of particles. The stability of the system comes from using Verlet integration to step particles. Consider a particle with position $\mathbf{x}$ and velocity $\mathbf{v}$. The Euler integration of this particle is:

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a} \cdot \Delta t$$
$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{v} \cdot \Delta t$$

The Verlet integration of this particle is:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + (\mathbf{x}_i - \mathbf{x}_{i-1}) + \mathbf{a} \cdot \Delta t^2$$

For Euler integration, we store the position and velocity of a particle. In Verlet integration, we instead store the position and previous position of a particle: the velocity is implicit as the difference between the two positions. In implicitly computing the velocity, we assume that the timestep is fixed.

During each simulation step, the particles move under the effects of external forces (such as gravity, wind, or user input), which may violate their constraints; the system of constraints is then solved by satisfying each constraint independently (as if it were a very stiff spring) for a number of iterations. By always fully satifying each constraint (and no more), we avoid overshooting, which is the main cause of "blowing up" in simulations of high-frequency springs with Euler integration. The constraints we use are position constraints, so the solver only modifies the position of the particles. However, we step the particles using Verlet integration (in which velocity is implicit) so that position updates implicitly change velocity as well: this represents the effects of the applied spring forces.

Through strategic placement of these two basic physics objects, larger structures can be created to model various soft-body objects, such as string and cloth (Figures 6 and 7, respectively), and even "rigid bodies" or an entire ragdoll. In general, object structures are represented by a collection of particles held together by distance constraints.

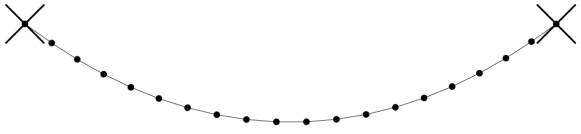


Figure 6: Steady-state of a string simulation with gravity.

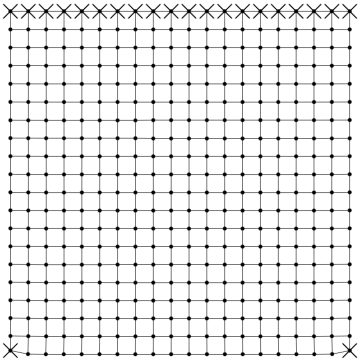


Figure 7: Steady-state of a cloth-simulation with gravity.

3.2 Basic model

Soft-body physics modeling encapsulates the shape of an object with masses and constraints. We now explain how to use this same strategy to model tree objects.

Observe that the plant geometries produced from L-systems, being composed of many connected line segments, lend themselves very well to this modeling scheme: line segment endpoints and line segments can correspond to particles and constraints, respectively. In plant terminology, particles correspond to nodes, and distance constraints correspond to internodes (stems). Note that particles themselves have no orientation, and the orientation of a stem is implicit in the position of its two endpoints.

This is a major departure from the traditional model of a tree object. Usually, trees are modeled as branches connected by joints, where branches are rigid meshes, and joints are relative transforms from a parent to a child branch. In such a hierarchical system, the stack of relative transforms is the mechanic by which the motion of a parent branch causes its children to move. However, the simulation of the tree entirely as physics objects, which all share the same coordinate system, precludes hierarchical modeling. Propagation of the motion of a parent branch to its children (and vice versa) is caused by constraints and computed by the physics engine; this is what produces the distinctive wavelike swaying of a tree object.

So far, we've placed particles at every node and a distance constraint at every internode. However, this isn't enough to keep the tree stiff: Figure 8 shows the results of the simulation after a few seconds. Without something to constrain the angles between stems, our initial attempt actually has the same particle-constraint structure as the string simulation (Figure 6).

3.3 Angular constraints

In the previous section, we accomplished the basic mapping from geometrical L-system output to soft-body physics simulation primitives. However, we need to build additional physics object structures to maintain the plant's shape against external forces. This can be accomplished by developing an "angular constraint", a physics object idiom which will constrain the angle between two distance constraints.

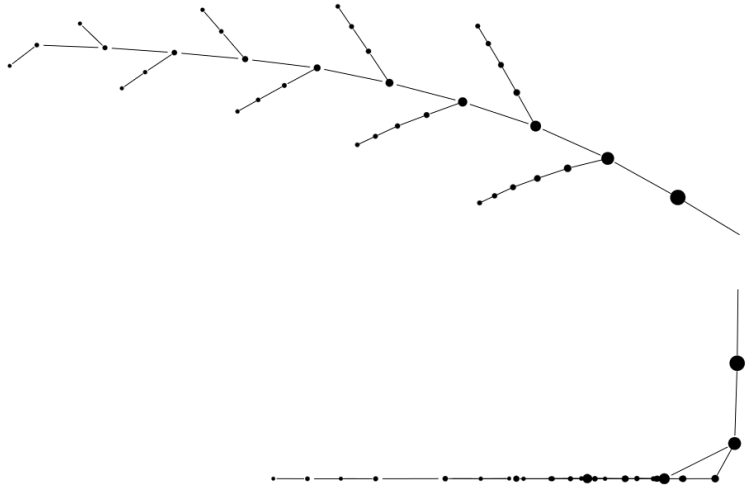


Figure 8: Starting state and steady-state of the basic model with gravity.

3.3.1 Angular constraint testbench

To hold soft bodies in place (for example, to model a cloth pinned to a wall, or a tree anchored in the ground), we often fix the position of select particles. Fixed particles have zero velocity and are unaffected by distance constraints. (In the figures, fixed particles are marked by a cross.)

The locus of a moving particle attached to a single fixed particle by a distance constraint is a circle. If we attach a moving particle to two fixed particles, the locus is now reduced to the intersection of two circles. The two loci must intersect for the constraints to be satisfiable, and if they do, the position of the particle should be fixed. However, we shall see that the way in which the loci intersect—tangentially or perpendicularly—affects the convergence of the system.

Consider a system of three particles—A, B, and C—with two distance constraints (A, B) and (B, C); let the positions of particles A and B be fixed. This is our testbench for the development of the angular constraint: our goal is to keep particle C collinear with A and B. Devoid of any other constraints, particle C is free to spin in a circle around particle B (Figure 9a), as mentioned previously.

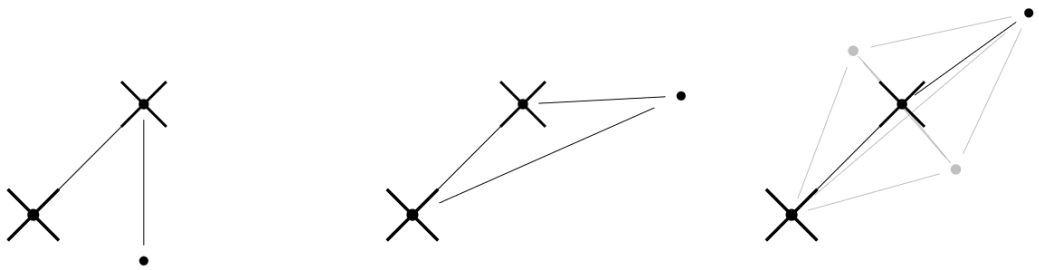


Figure 9: Steady-state of different angular constraints with gravity.

3.3.2 Weak angular constraint

To form an angular constraint, we must add some number of extra particles and constraints to keep particle C in place relative to A and B. Our first attempt is modest: a single extra constraint between A and C. The steady-state of this system under gravity is shown in Figure 9b. Even in this simple test case, the angular constraint does not hold up well under external forces (in this case, gravity). It turns out that this configuration of constraints is not strong enough to hold up a plant structure. However, analyzing its convergence is instructive.

Recall that in our physics simulation, each constraint is solved independently, and we iterate on all constraints to attempt to approach the true solution. While the correct solution for this system is indeed the collinear configuration we're looking for, solving for this solution using the two present constraints produces a very slow convergence.

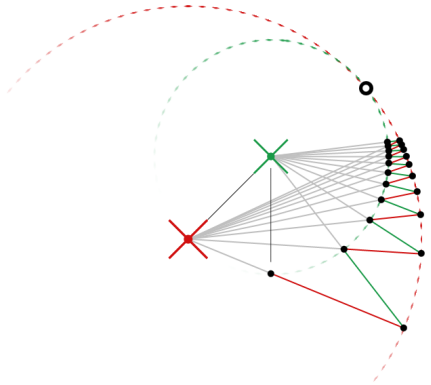


Figure 10: Action of two constraints with tangential loci on a particle.

Figure 10 shows the motion of particle C under the action of constraints (A, C) and (B, C) over several solver iterations. The locus of C under constraint (A, C) is a circle centered at A with a radius equal to the resting-length of (A, C); the locus of C under constraint (B, C) is a smaller circle centered at B with a radius equal to the resting-length of (B, C). During successive iterations of the solver, the position of particle C "bounces" from one locus to another. Moving the position of a particle back and forth is correct solver procedure; the problem here is that these two circles are tangent at their intersection. As we approach the solution, the effects of the two constraints become increasingly antagonistic.

From this example, it appears that convergence along two constraints is slow when their loci are tangent at their intersection. We might suspect that, conversely, two constraints with perpendicularly intersecting loci produce fast convergence.

Constraints only push or pull on particles, so a particle moves towards its true solution only along the direction of its constraints. In the space close to the solution, when the motion of the particle is small compared to the length of the constraints, we can view the directions of the two constraints as providing a "basis" for the solving motion of the particle. Tangentially interescting loci indicate that the direction of the two constraints is almost parallel: the basis is ill-conditioned in the space near the solution, so the particle moves back and forth and never makes progress. Conversely, perpendicularly intersecting loci indicate that the direction of the two constraints is perpendicular, providing a well-conditioned basis of motion near the solution—which quickly determines the correct position of the particle.

3.3.3 Strong angular constraint

Knowing the underlying cause (tangential loci) of the undesirable properties of this system (slow convergence), we can design something better. We still want to add constraints to cause the position of C in the true solution to be collinear with A and B. Last time, we made the mistake of choosing constraints with tangent loci. Instead, we should choose two constraints that are satisfied by the same true solution, but have perpendicular loci: this suggests two constraints with endpoints on opposite sides of C preventing movement either left or right, illustrated in Figure 11.

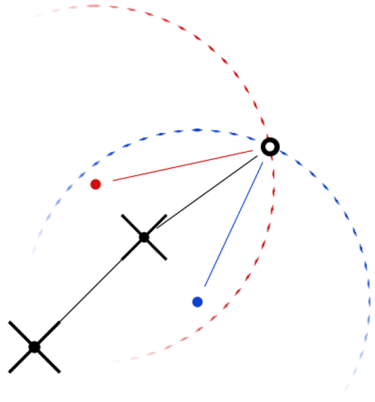


Figure 11: Loci of two "perpendicular" constraints.

We create two supporting particles L and R, placed to the left and right of the center particle B, and then add the constraints (C, L) and (C, R). To keep these supporting particles in place, we need to add four more constraints: (B, L), (B, R), (A, L), and (A, R). Finally, to keep the structure from collapsing, we should create constraints (L, R) and (A, C). The final result is a strong angular constraint, locking the two distance constraints in place. Its steady-state, holding up well under gravity, is shown in Figure 9c.

Figure 9 shows the steady-states of (a) the original system, (b) our first attempt, and (c) our second attempt. The drooping of the systems under gravity was exaggerated by lowering constraint power[4] to 10%.

3.3.4 Semi-rigid bar

This testbench helped us analyze the convergence behavior of our angular constraint designs. However, we can't be assured that our angular constraint design is satisfactory yet. Recall that we want motion to propagate up and down plant structures. Unlike the testbench, most angular constraints in actual usage will not have any fixed particles.

As another basic test of our angular constraint, we can try to build a semi-rigid bar: a straight line of particles and distance constraints (the structure of a string object), but with an angular constraint between every pair of adjacent distance constraints. When simulated, the semi-rigid bar bends slightly under gravity (Figure 12).

[4]Constraints are normally applied fully, but we can "soften" constraints (and systems of constraints) by only applying a percentage of the correction per iteration.

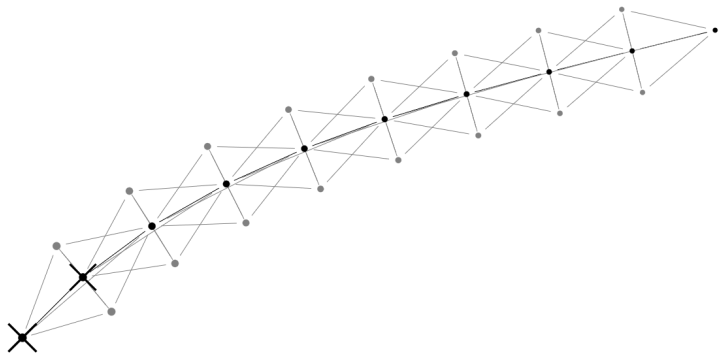
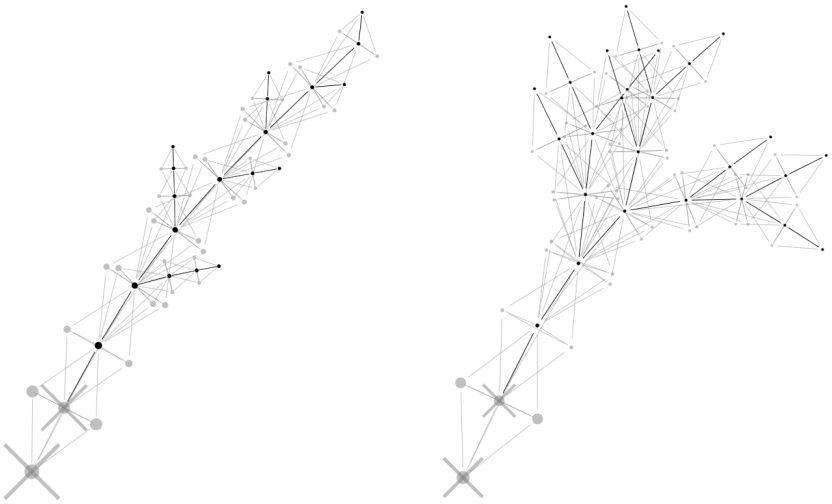


Figure 12: Steady-state of semi-rigid bar with gravity.

3.3.5 Semi-rigid trees

We now have all the physics structures we need to simulate tree dynamics. In the initial attempt to map L-systems to physics objects in Section 3.2, every line segment produced by the L-system corresponded to one particle and one distance constraint. To keep the tree stiff, every line segment produced by the L-system should also correspond to an angular constraint between the parent segment and the new segment. Figure 13 shows the steady-state of our tree dynamics structure: the trees flex under the effect of gravity, but do not collapse.



(a) (b)

Figure 13: Steady-state of two semi-rigid trees with gravity.

3.4 Graphs and deletion

A common feature of soft-body physics engines is the simulation of object "destruction", such as the cutting of a rope or tearing of a cloth. Within the particle-constraint model of soft bodies, breakage is represented by the deletion of constraints, or, less commonly, by the deletion of a particle. To support

interactive destruction of trees, we have to implement physics object deletion routines correctly. In particular, we must deal with the complications that angular constraints add to these routines.

Particles and constraints form a graph where particles are vertices and constraints are edges. A constraint's existence is dependent on its two endpoints: when a particle is removed, all associated constraints must be removed as well. This requires a graph data structure. The particle-constraint graph may be implemented as an edge list (of constraints) in each vertex (a particle); the consequent physics object deletion functions are shown in Algorithm 1. Besides physics object removal, the graph data structure can be used for other operations (see Section 3.5 and Appendix A).

Algorithm 1 Soft-body physics object destruction functions

```
function DESTROY-DISTANCE(DC)
    A ← the tail vertex of DC                          // Graph edge deletion
    remove DC from the edge list of A
    B ← the head vertex of DC
    remove DC from the edge list of B

    delete DC from memory
end function

function DESTROY-PARTICLE(P)
    E ← edge list of P                                 // Graph vertex deletion
    for all DC ∈ E do
        DESTROY-DISTANCE(DC)
    end for

    delete P from memory
end function
```

Our new angular constraints complicate the deletion process. An angular constraint is dependent on the two distance constraints it is created on, in the same way a distance constraint is dependent on its two endpoint particles. Therefore, when a distance constraint is removed, all associated angular constraints must be removed; in other words, distance and angular constraints form another graph (Figure 13).

We will refer to each graph according to the type of its edge objects. The original graph, with particles for vertices and distance constraints for edges, is the "DC (distance constraint) graph". The second graph, with distance constraints for vertices and angular constraints for edges, is the "AC (angular constraint) graph". In graph theory terms, the AC graph is a spanning subgraph[5] of the line graph[6] of the DC graph.

Algorithm 2 shows the new necessary steps to delete each type of physics object. Since distance constraints are edges of the DC graph and vertices of the AC graph, we can expect the deletion of distance constraints to be especially involved. Note that even though angular constraints are implemented with extra particles and constraints, we don't think of them as such—in fact, it would be an error to try to destroy "implementation" physics objects of an angular constraint.

During vertex deletion operations in Destroy-Particle and Destroy-Distance, we rely on the corresponding edge deletion function (Destroy-Distance and Destroy-Angular, respectively) to delete all edges. However, we can't iterate the edges directly, since we call the edge deletion function (which modifies the edge list)

[5] A spanning subgraph of G is a subgraph of G containing all the vertices of G (but a subset of the edges).
[6] A line graph represents adjacencies between the edges of a graph.

Algorithm 2 Soft-body physics object destruction functions with angular constraints

```
function DESTROY-ANGULAR(AC)
    for all DC ∈ eight "implementation" constraints of AC do
        DESTROY-DISTANCE(DC)
    end for
    for all P ∈ two "implementation" particles of AC do
        DESTROY-PARTICLE(P)
    end for

    M ← the first vertex of AC                          // AC graph edge deletion
    remove AC from the edge list of M
    N ← the second vertex of AC
    remove AC from the edge list of N

    delete AC from memory
end function

function DESTROY-DISTANCE(DC)
    E ← edge list of DC                                 // AC graph vertex deletion
    for all AC ∈ E do
        DESTROY-ANGULAR(AC)
    end for

    A ← the tail vertex of DC                           // DC graph edge deletion
    remove DC from the edge list of A
    B ← the head vertex of DC
    remove DC from the edge list of B

    delete DC from memory
end function

function DESTROY-PARTICLE(P)
    E ← edge list of P                                  // DC graph vertex deletion
    for all DC ∈ E do
        DESTROY-DISTANCE(DC)
    end for

    delete P from memory
end function
```

on each edge. The workaround is to either make a local copy of the edge list, or call the edge deletion function on the head of the edge list until it is empty.

3.5 Tree splitting

With physics object deletion functions properly implemented, we now develop the algorithm to split tree objects. This algorithm is the basis for all tree-structure-destroying interactions, such as cutting trees or burning trees.

A tree object is expected to know about its constituent particles and constraints. For the purposes of rendering and tree object deletion, we need no particular organization of the constituent physics objects; merely a list of particles and a list of constraints will suffice. This invariant is easy to maintain during generation: all trees start with two particles and a distance constraint, and every "forward" L-system instruction generates one particle, one constraint, and one angular constraint.

Recall that with our physics model, breakage is modeled by the removal of distance constraints. We can model the breakage of tree objects from external forces as stem removal: to start, we will call the Destroy-Distance routine from the previous section on a constraint we want to break. However, when tree objects are cut, they should produce a new tree object. To maintain the invariant of each tree keeping a list of its physics objects, we will have to find out which physics objects belong to the new tree, then move them from the old tree to the new tree.

Algorithm 3 Tree splitting procedure

```
function TREE-SPLIT(OldTree, DC)
    create a new tree object NewTree

    A ← the tail particle of DC
    B ← the head particle of DC
    DESTROY-DISTANCE(DC)

    NewTree.particles ← CONNECTED(B)
    for all P ∈ NewTree.particles do
        remove P from OldTree.particles
    end for
end function
```

The tree splitting procedure (Algorithm 3) starts with a target distance constraint. First, we store the endpoints of the constraint; call them A and B. Then, we delete the target distance constraint. In graph theory, a "tree graph" is a connected graph without cycles, and the removal of any edge results in two connected components. The trees generated by branching L-systems are such tree graphs; therefore, we can be sure that A and B are now disconnected[7]. Furthermore, if we implement the particle-constraint graph as a directed graph, we can set, as a convention, the direction of all edges away from the root of the tree during generation. This way, we know that A is the root-side particle and B is the branch-side particle: by finding the connected component of B, we find all the particles in the new tree.

[7]The presence of angular constraints implemented as extra particles and constraints doesn't affect the properties of the graph of "real" distance constraints, becasue the deletion of the target distance constraint also removes all associated angular constraints (Section 3.4).

4 Special effects

Since we simulate trees as physics objects, special effects can easily be added with the support of the physics engine. This section describes the development of three typical special effects which affect trees: fire, rain, and wind.

Both fire and rain effects are implemented as particles spawned via an emitter system, described in Appendix B. In general, emitters spawn particles with collision detection against tree structures, and resulting behaviors are performed during collision response. (Note that particles from emitters are not the same particles as the Verlet-integrated particles used for tree nodes.)

Figure 14: Screenshots of fire and rain special effects in action.

4.1 Fire

An emitter object can be configured to produce flame particles[8] in a small area, as shown in Figure 15.

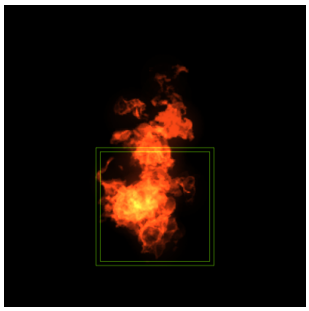


Figure 15: An active fire emitter.

We now develop a model for flammable trees. The goal is a visually impressive display of a tree catching on fire, and not a detailed thermodynamic simulation. As such, we define a few visual criteria for the behavior of burning trees. Starting with some external source of flames, trees should be able to catch on fire. Once some part of a tree is burning, the combustion should spread locally across a single tree, eventually engulfing the entire tree, as well as globally, moving from one tree to another. Additionally, fire should be biased towards spreading upwards and possibly also be affected by wind (Section 4.3). Finally, trees should not burn indefinitely, but should eventually be consumed by the fire and collapse.

Our model for flammable objects is based on a "fuel cell" primitive, a game-logic object consisting of a hit-detection region, a fire emitter, and a temperature record. The hit-detection region listens for fire particles. The temperature of the fuel cell increases when fire particles are seen, and decays over time to room temperature. The fire emitter begins to emit fire particles when the temperature reaches a threshold. The upshot of these three components is that fuel cells catch on fire after sufficient exposure to flames—and after they are ignited, the flames they produce can set other fuel cells on fire. In general, we can model an arbitrarily-shaped flammable object by attaching smaller fuel cells all over the object. For trees, we attach several fuel cells to each distance constraint, making sure that the emitting radius of each fuel cell overlaps the hit-detection region of adjacent fuel cells.

Figure 16 shows a time-lapse of our fire model in action. this scheme satisfies all the visual criteria discussed above. Flammable areas of an object are indicated by fuel cells. A single tree is always eventually engulfed by flames if fuel cells overlap, and since the hit-detection of flame particles happens globally, any burning object can set other flammable objects on fire. Fire is propagated via flame particles, which move upwards and can be affected by wind, so fire spreading is biased upwards and affected by wind as well. To collapse burning trees, we can add an additional field to fuel cells which describes the remaining amount of fuel. Fuel cells should stop emitting flames when this field is depleted, and the tree object may break branches which are (almost) entirely burnt.

This model is by no means the only way to implement flammable trees. A tree-specific solution might consist of flame objects traveling along tree branches, destroying branches when they reach endpoint nodes[9]. However, this "fuel cell" model is quite general-purpose, consistently applicable to all flammable objects in a game.

[8] The flame textures shown in Figures 15 and 14 are from Street Fighter IV.
[9] This is actually the fire model used in World of Goo (http://www.worldofgoo.com/)

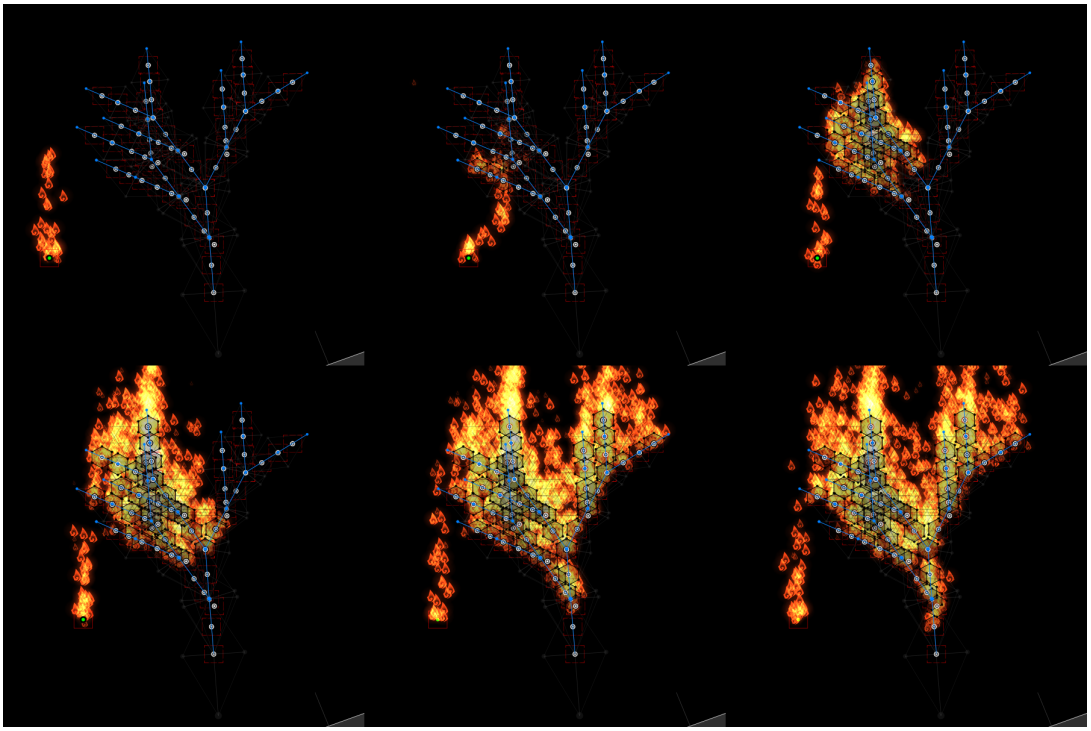


Figure 16: Time-lapse of fire propagation on a tree.

4.2 Rain

An emitter object can be configured to produce rain particles over a large area, as shown in Figure 17. Note the collision of rain particles with walls.

As with fire, we are more interested in visuals than an accurate simulation. Again, we consider some visual criteria for trees in the rain. Rain should wet and put out burning trees; raindrops should also cause the smaller branches of trees to shake on impact. Finally, raindrops should splatter against anything they collide with.

We modeled burning trees using the "fuel cell" model (Section 4.1); the fire-fighting effects of raindrops is implemented by extending the existing model. Let the hit-detection region of fuel cells listen for raindrops as well as flame particles, and lower the temperature of the fuel cell in response; with enough incoming raindrops, ignited fuel cells will be extinguished.

To implement shaking of tree branches from raindrops, we ask the physics engine to provide collision detection and collision response between emitter particles and distance constraints. However, tree branches are modeled as immaterial constraints: to "apply" an impulse to a distance constraint, we have to apply it to the endpoints instead.

Splattering is almost trivial to implement: raindrop objects turn into splattering animations upon receiving collision events from the physics engine. Since the physics engine handles tree-raindrop collisions, raindrops will splatter against trees as well as walls, as seen in Figure 14.

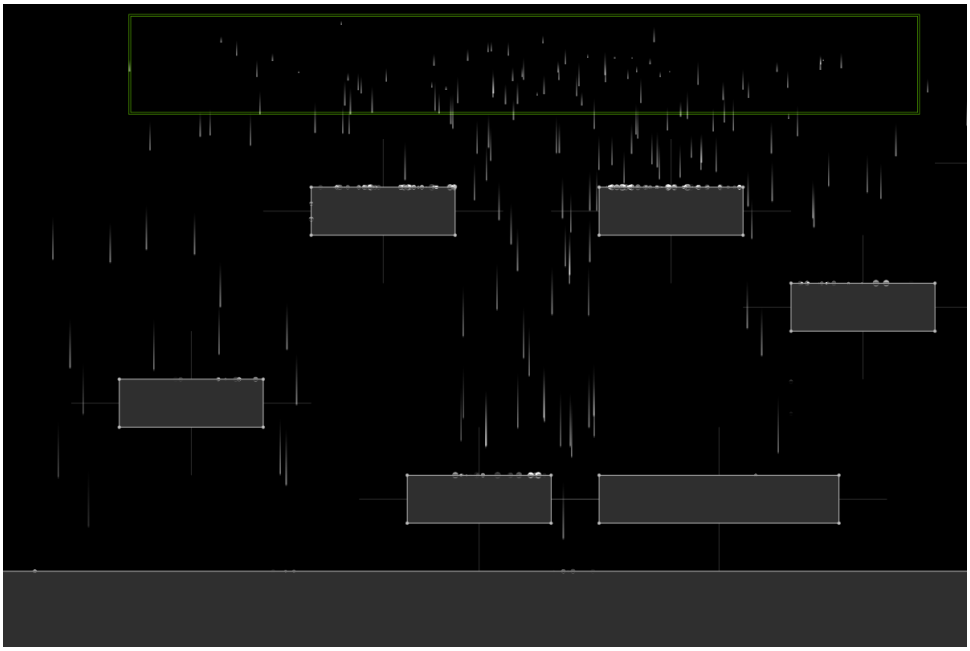


Figure 17: An active rain emitter.

4.3 Wind

In many physically-based approaches to tree animation, wind forces are the principal (and sometimes only) motivators of tree motion [4, 13]. In contrast, our approach subsumes the application of wind forces as arbitrary external forces.

We assume that we can query for wind speed at any position in world-space. The development of a wind model is outside the scope of this thesis; here are two examples of previous wind models:

- Lee [6] describes a coarse-grained fluid simulation centered on the player (to give the illusion of an infinite vector field) to provide wind speeds. A live fluid simulation allows "blowing" interactions in which tree objects respond to wind forces.
- Habel [4] notes that while slow-moving wind fields are dominated by turbulence, strong winds are well described by a single global wind vector. Although a simple model, this allows us to easily control wind speeds, perhaps as scripted events.

Recall that external forces are applied to particles (Section 3.1). However, this isn't the best way to apply wind forces to trees. Zioma [13] observes that instead of applying drag forces, we should consider (and at least approximate) the aerodynamic properties of tree branches. In our case, we can apply wind forces to branches proportional to the projection of wind speed onto the normal of a branch. This applies the full wind force to branches orthogonal to the wind direction, and no wind force to branches aligned with the wind direction.

In Section 4.1, we noted that fire should propagate in the direction of wind. Appendix B discusses the application of drag forces to particles.

5 Conclusion

We have produced a fully simulated tree, capable of responding with realistic motion to arbitrary external forces and stimuli. In the process, we also developed a soft-body physics structure "idiom", the angular constraint, to build stiff trees. The presented method of tree generation—the combination of L-systems and soft-body physics—quickly creates simulations of many different types of trees. The tree model, being supported by the physics engine, is easily extended with special effects. As some examples of special effects, we have designed fire and rain emitters and implemented the interaction of flames and raindrops with trees. While developing the fire special effect, we also designed a fire propagation model that can be used for many different flammable objects.

5.1 Future work

The potential level-of-detail uses of L-systems remains to be developed. Recall that we do not store any tree structure in the tree object (Section 3.5) beyond a list of constituent physics objects. However, if we somehow retain information about the self-similar structure of the L-system output inside the tree, we can take shortcuts to simulate and render clumps of physics objects at once.

Another speed-up technique left unconsidered is a "sleeping" policy for the physics engine, where groups of physics objects which have come to rest are no longer simulated until further disturbed. This can be implemented in conjunction with the connected components scheme discussed in Appendix A.

For simplicity, the geometry and dynamics in this thesis were developed in 2D. All the particulars of the 2D work described, including the fire propagation model, would carry over to a corresponding 3D simulation; however, working in 3D adds a few problems.

- Orientation: There are more degrees of freedom for orientation in 3D than in 2D. In particular, controlling the L-system T action ("turn") is more difficult.
- Twisting: In two dimensions, the angular constraint (Section 3.3) needs two support points and eight additional distance constraints. In three dimensions, three or even four support points may be needed. With four support points and constraints between every pair of particles, an angular constraint requires nineteen additional distance constraints. Moreover, this isn't enough to secure the tree structure: angular constraints only prevent bending, not twisting.
- Self-collision: The 2D string and cloth simulations (Section 3.1) are technically incorrect: as there is no self-collision of the soft body, the soft body is free to move through itself. In 2D, this may actually be a boon, as the self-penetration creates a pseudo-3D effect. This is not the case in 3D, where self-penetration of string and cloth is a major issue. However, strings and cloth are foldably soft, while the many constraints of a tree keep it stiff. Despite the physical implausibility, self-penetration may not be very noticeable in a stiff 3D tree.

Acknowledgements

This thesis would not have been possible without the guidance of my advisor, Donald Fussell, and his seemingly endless knowledge about computer graphics. Special thanks to Calvin Lin for being my second reader and always having time to provide additional help with writing and revising. Thanks to Warren Hunt, my external committee member, for advice on how to put together an engaging presentation.

I am grateful to Katheryn Shi for always being the first to look for bugs in my programs and flaws in my reasoning. And finally, thanks to my parents, Qixin Zhang and Richard Li, who taught me that hard work overcomes all obstacles.

Appendix A Adaptive iterations

A.1 Connected components

The constraint solver uses multiple iterations to converge to the "true" solution. Increasing the number of iterations improves convergence, causing a stiffer (higher frequency) simulation. So far, the solver has used an arbitrary set number of iterations, which isn't quite ideal; [5] suggests an adaptive number of iterations based on tracking an error margin.

Whenever there are multiple unconnected soft-body objects, the particle-constraint graph has several connected components. Since they are not connected by constraints, we are free to solve each component separately—and we should: some of these components may be much smaller than others, and in the worst case, we could be wasting many iterations on components consisting of a single constraint (which, as a special case, reaches full convergence in a single iteration). Generally, a system with more constraints will need more iterations for satisfactory convergence. This observation suggests an improved relaxation strategy where we take some time to find out the connected components of the graph, segregating the constraints, and then solve each system with an adaptive iteration count.

The connected components algorithm runs during the physics update before the relaxation step (if any soft-body physics object creation or deletion requests have been made by the user since the last update). When solving each system of constraints, we can figure out the iteration count by tracking error (stopping at an acceptable error or a maximum number of iterations) or by guessing (based on empirical results). In our implementation, the guess was $\lceil \sqrt{n} \rceil$, where n is the number of constraints in the system.

While adaptive, this isn't a very sophisticated heuristic. A more complicated heuristic, for example, might try to differentiate between "string-shaped" systems, with graphs resembling cycle graphs, and "rigid-body-shaped" systems, with graphs closer to complete graphs. Let the graph of a system be $G = (V, E)$; the ratio of $|E|$ to the number of edges in the complete graph $K_{|V|}$ would be a measure of the "completeness" of G.

Running the connected components algorithm during the physics update (potentially every frame) seems slightly heavy-handed. The alternative would be some sort of dynamic graph data structure to keep track of the connected components of the graph. Incremental dynamic graph structures (which only support vertex creation and edge creation) perform very well: the "disjoint-sets data structure with path compression and union-by-rank", described by Galler and Fischer [3], has an almost linear time per operation[10]. However, decremental dynamic graph structures are less promising [10].

This new relaxation scheme is not just a performance boost; it also changes the behavior of the simulation. A large system of constraints converges slower than a small system of constraints. Conversely, for the same number of iterations, a small system of constraints is stiffer than a large system of constraints. The original constant iteration count causes small systems to be stiffer than large systems; with an adaptive iteration count, each connected-component system of constraints now exhibits a similar stiffness. The upshot is that small trees wiggle just as much as large trees.

[10] The amortized time per operation was proven by Tarjan [12] to be $O(\alpha(n))$, where $\alpha(n)$ is the inverse of the extremely fast-growing Ackermann function.

A.2 Wall contacts

Besides having their motion restricted by distance constraints, particles also collide against walls. For particles that have moved inside walls, the collision response is simply to move the particle out of the wall. This response, a "wall contact", can be thought of as an inequality position constraint.

It's not enough to fix wall contacts once per physics update: these constraints need to be stored and relaxed along with distance constraints. Otherwise, the solver ignores wall constraints and particles colliding with walls will be shoved back into walls (and possible straight through thin walls).

If we separate constraints into connected components, as shown above, we also have to associate wall contacts with the correct component. To do this, we should run the connected-components algorithm before wall-particle collision detection and tag particles with a component ID. Then, during wall-particle collision detection, we can tag wall contacts with component IDs from particles.

Appendix B Particle systems

Particle systems are a method to graphically model fuzzy phenomena as groups of particles. The production of particles is controlled by emitters, which emit particles and initialize their position and velocity. By modifying the spawn parameters of emitter objects, many different effects can be produced. For example, an explosion effect can be created by spawning particles with a starting position close to the emitter and a starting velocity directed away from the emitter; a rain effect can be created by spawning particles in a large rectangular area around the emitter and allowing them to fall under gravity.

We use the physics engine to simulate particles. Since the physics engine is in charge of the physical aspect of all particles, it is able to provide collision detection and response between particles and other physics objects.

B.1 Applying Stokes' drag to particles

While developing fire special effects in Section 4.1, we noted that we should apply wind forces from Section 4.3 to flame particles to effect fire propagation along the direction of the wind.

We can model the action of wind on a particle (with velocity $\mathbf{v}$) as Stokes' drag:

$$\mathbf{F} = -b\mathbf{v}$$
$$m\mathbf{a} = -b\mathbf{v}$$
$$\mathbf{v}' = -\frac{b}{m}\mathbf{v}$$

Under the effects of Stokes' drag, the velocity of a particle decays exponentially. To achieve this effect in a discrete simulation, we multiply the velocity of a particle by a certain percentage, k, every frame. For example, if we want to retain 80% velocity after one second at a rate of 60 frames per second:

$$k^{60} = 80\%$$
$$k = \exp\left(\frac{\ln 80\%}{60}\right)$$

Finally, we add the effects of wind speed, $\mathbf{w}$, by applying the velocity-reducing multiplier in the wind's frame of reference:

$$\mathbf{v}_{i+1} = k \cdot (\mathbf{v}_i - \mathbf{w}) + \mathbf{w}$$

References

[1] Candussi, A., Candussi, N., and Hllerer, T. Rendering realistic trees and forests in real time, 2005.

[2] Eloy, C. Leonardo's rule, self-similarity, and wind-induced stresses in trees. *Phys. Rev. Lett. 107* (Dec 2011), 258101.

[3] Galler, B. A., and Fisher, M. J. An improved equivalence algorithm. *Commun. ACM 7*, 5 (May 1964), 301–303.

[4] Habel, R., Kusternig, A., and Wimmer, M. Physically guided animation of trees. *Computer Graphics Forum 28*, 2 (2009), 523–532.

[5] Jakobsen, T. Advanced character physics. In *Game Developers Conference Proceedings* (2001), CMP Media, Inc., pp. 383–401.

[6] Lee, E. Realistic real-time grass rendering. Master's thesis, Southern Methodist University, Dallas, Texas, December 2010.

[7] Leonardo da Vinci. The notebooks of Leonardo da Vinci, translated by Jean Paul Richter. Seattle, , circa 1500.

[8] Mantler, S., Tobler, R. F., and Fuhrmann, A. L. The state of the art in realtime rendering of vegetation, 2003.

[9] Prusinkiewicz, P., and Lindenmayer, A. *The Algorithmic Beauty of Plants (The Virtual Laboratory)*. Springer, Oct. 1991.

[10] Shiloach, Y., and Even, S. An on-line edge-deletion problem. *J. ACM 28*, 1 (Jan. 1981), 1–4.

[11] Sousa, T. Vegetation procedural animation and shading in crysis. In *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, 2008, pp. 373–385.

[12] Tarjan, R. E. Efficiency of a good but not linear set union algorithm. *J. ACM 22*, 2 (Apr. 1975), 215–225.

[13] Zioma, R. Gpu-generated procedural wind animations for trees. In *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, 2008, pp. 105–121.