Plant Substructuring and Real-time Simulation Using Model Reduction

Yili Zhao
Department of Computer Science
Viterbi School of Engineering
University of Southern California

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
August 2014
I would like to dedicate this thesis to my loving parents, Baochun Zhao and Tian Li.
Acknowledgements

Thank you, my advisor: Jernej Barbič. He is a genius, an expert in many fields. He gave me tremendous invaluable advices in academic research and life. He is a very gentle and nice mentor to work with, never ever yelling at his students (very important virtue for a PhD advisor!). It is my honor to have him be my PhD advisor. I will be nothing but grateful to him.

Many thanks are given to my PhD thesis committee: Jernej Barbič, Gaurav S. Sukhatme, Ulrich Neumann, Igor Kukavica, and Stefan Schaal. I really appreciated all their constructive comments and feedbacks which helped me think critically and deeply into the research problems and greatly polish my thesis.

Specially thanks Hao Li, a young, amazing, super talented, friendly, energetic, humorous professor. I received generous amount of help and advices from him! Sincerely thank you.

Thank you Computer Science Department, USC. First of all, you are creative and innovative. You keep hiring so many top-notch scholars as faculties doing cutting edge research in broad areas of computer science, teaching us the knowledge, and mentoring us to the wonderful academic world. Second, you are friendly and supportive. Staffs including Lizsl De Leon Spedding, Michael Archuleta, Kusum Shori, Steve Schrader, Macy Yu really helped me a lot! Thank you. Last but not least, you are rich. As a PhD student here, I never worried about funding, just concentrated on study and research.

Thank you USC, University of Southern California, (or rumor says University of Summer Construction) for your world class facilities and support. I love the Daland’s Swim Stadium (the main Aquatics venue at the 1984 Olympic Games). I swam a lot there, free style, 1 mile (nonstop) per day.

Many thanks go to Computer Graphics Lab at Peking University. Thank you my teachers,
Guoping Wang, Shihai Dong, Sheng Li. Without your immeasurable help, I could not fulfill my dream to pursue the PhD degree in a top university in the United States.

Thanks to the fact that I do not have a girl friend for more than two decades since I entered the elementary school. I know it is a little bit pathetic, but it really saves me a lot of time, energy, and money, so that I can focus on study and doing research.

Thanks my families: my parents, Baochun Zhao and Tian Li, my grandparents, aunts, uncles, and my cousins. No matter which path I choose, what adventure I take, you stand behind me. All of you are the source of my strength.

I owe debts of gratitude to Rose Liu, Daniel Lin, David Lin, and Henry Lin, together with your parents. You are my families in Los Angeles. You made me feel I still stay at home, not in a foreign country. Thank you!

Thank you, Lei Cheng, Chen Fan, Shengtao Zhou, Xin Huang. You are my brothers since childhood. I missed so much the old days when we had together.

Thank you, Yao Sun, Dikang Gu, Bo Wang, Fenye Bao, Jian Zhao, Yi Sun, Ying Xiong, Jingjing Gu, my best friends during the college years.

Thank you, Bin Lu, Chen Tang my best friends in Peking University. I missed the old time when we used to hang out for late-night snacks.

I am much obliged to Wei Guan. You helped and guided me tremendously for those years in USC. You are like my elder-brother.

Thank you, Xiaoyan Zhang. You are like my sister, always taking care of me.

Thank you, Qiming Li. You are my little star.

Thank you, Meihong Wang. I am always willing to listen to your opinions, exchange ideas with you. You are pretty smart and my best friend.

Thank you, Jing Huang, the coach of ACM-ICPC team at USC, and my good roommate. We discussed a lot of algorithms in the spare time. Without these algorithms, I cannot publish my SIGGRAPH papers.

Thank you, Yijing Li, Hongyi Xu, Guan Pang, Rongqi Qiu, Jiaping Zhao, Zhenzhen Gao, Lu Wang, Qianyi Zhou. I have a lot of good memories with all of you in the lab.

Thank you, all my friends! The list could be incredibly long if I put all your names down.
I cannot do that since it might be longer than my actual thesis. But please believe me you are always in my heart!
Abstract

This research is focusing on real-time, physically-based simulation of plants undergoing large deformations. To achieve this goal, we first propose a novel algorithm based on model reduction and domain decomposition. It extends 3D nonlinear elasticity model reduction to open-loop multi-level reduced deformable structures. We decompose the input mesh into several domains, build a reduced deformable model for every domain, simulate each one separately, and connect domains using proper inertia coupling. This makes model reduction deformable simulations much more versatile: localized deformations can be supported without prohibitive computational costs, parts can be re-used and precomputation time can be shortened. Our method does not use constraints, and can handle large domain rigid body motion in addition to large deformations, due to our derivation of the gradient and Hessian of the rotation matrix in polar decomposition. We show real-time examples with multi-level domain hierarchies and thousands of reduced degrees of freedom.

Then we design a pre-processor which takes a plant “polygon soup” triangle mesh as the only input and quickly pre-compute necessary data for the subsequent simulation. This tool breaks the ice for adoption of our multidomain dynamics simulator in practice. Our pre-processor is robust to non-manifold input geometry, gaps between branches or leaves, free-flying leaves not connected to any branch, small unimportant geometry (“debris”) left in the model, and plant self-collisions in the input configuration. Repeated copies (instances) of plant subparts such as leaves, petals or fruits can be automatically detected by our pre-processor.

We enhanced our multidomain dynamics simulator to provide plant fracture, and inverse kinematics to easily pose plants. It can simulate complex plants at interactive rates, subjected to user forces, gravity or randomized wind. We simulated over 100 plants from diverse cli-

mates and geographic regions, including broadleaf (deciduous) trees and conifers, bushes and flowers. Our largest simulations involve anatomically realistic adult trees with hundreds of branches and over 100,000 leaves.

Finally, we propose our future research in several directions including adding hierarchical instancing, collision detection and handling, etc.
## Contents

**Contents** vii

**List of Figures** x

**List of Tables** xii

### Chapter 1 Introduction 1

1.1 Plant Simulation Using Multi-domain Dynamics . . . . . . . . . . . . . . . . 2
1.2 Plant Substructuring and Preprocessing . . . . . . . . . . . . . . . . . . . . 2
1.3 Procedural Materials . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.4 Asynchronous Parallel Timestepping . . . . . . . . . . . . . . . . . . . . . . 3
1.5 Thesis Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

### Chapter 2 Related Work 5

2.1 Plant Modeling and Simulation . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 Finite Element Method and Model Reduction . . . . . . . . . . . . . . . . . 6
2.3 Domain Decomposition . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

### Chapter 3 Multidomain Dynamics in Reduced Space 10

3.1 Background: Model Reduction and Domain Decomposition . . . . . . . . 10
3.2 Kinematics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.3 Dynamics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.4 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.5 Polar Decomposition Rotation Gradient . . . . . . . . . . . . . . . . . . . . 23
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6</td>
<td>Appendix: System Force Integrals</td>
<td>24</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Plant Preprocessing and Simulation</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Plant Preprocessing</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Forests</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Interactive Plant Design</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Fracture</td>
<td>42</td>
</tr>
<tr>
<td>4.5</td>
<td>Motion of Leaves</td>
<td>43</td>
</tr>
<tr>
<td>4.6</td>
<td>Results</td>
<td>44</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Procedural Materials and Asynchronous Parallel Timestepping</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>Procedural Stiffness</td>
<td>52</td>
</tr>
<tr>
<td>5.2</td>
<td>Procedural Timestep</td>
<td>54</td>
</tr>
<tr>
<td>5.3</td>
<td>Procedural Damping</td>
<td>55</td>
</tr>
<tr>
<td>5.4</td>
<td>Asynchronous Parallel Timestepping</td>
<td>58</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Subdividing the Timestep</td>
<td>58</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Parallel Simulation</td>
<td>59</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Randomized Wind Force Field</td>
<td>63</td>
</tr>
<tr>
<td>6.1</td>
<td>Features of a Randomized Wind Force Field</td>
<td>63</td>
</tr>
<tr>
<td>6.2</td>
<td>Improved Perlin Noise Algorithm in 4D</td>
<td>64</td>
</tr>
<tr>
<td>6.3</td>
<td>Computing the Randomized Wind</td>
<td>65</td>
</tr>
<tr>
<td>6.4</td>
<td>Applying Wind Forces to the Plant</td>
<td>65</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>System Overview</td>
<td>69</td>
</tr>
<tr>
<td>7.1</td>
<td>Botanical Preprocessor</td>
<td>69</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Command Line</td>
<td>69</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Preprocessor</td>
<td>69</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Useful Tools</td>
<td>75</td>
</tr>
<tr>
<td>7.2</td>
<td>Botanical Simulator</td>
<td>76</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Simulation Control</td>
<td>76</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 Oregon white oak tree (Quercus Garryana) in strong randomized wind: 1
1.2 Anatomically realistic simulation of a peach tree (Prunus Persica) with fracture: 2

3.1 Domain decomposition: 12
3.2 Effect of system and interface forces: 12
3.3 Fitting the best interface transformation: 14
3.4 Model reduction with a large number of localized degrees of freedom: 19
3.5 Instancing (Space Station): 20
3.6 Our method supports localized deformations: 20
3.7 Similar trajectories, but different frequencies: 21
3.8 $C_0$ embedding is effective with nearly-rigid interfaces: 23

4.1 Preparing mechanical models ready for the simulation: 26
4.2 Branches (F), twigs (R1) and leaves (R2): 27
4.3 Instancing and anchors: 28
4.4 Loop resolution: 32
4.5 Voxel simulation meshes: 34
4.6 Hierarchy of a forest: 38
4.7 Fir forest in randomized wind field: 39
4.8 Editing tree shapes using point constraints: 40
4.9 Real-time Fracture: 42
4.10 Leaf Motion: 43
4.12 Representative subset of simulated models: 45
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.13</td>
<td>Space-time instantaneous (localized) external force followed by free vibration.</td>
<td>46</td>
</tr>
<tr>
<td>4.14</td>
<td>Palm tree (Cocos Nucifera) in strong randomized wind</td>
<td>47</td>
</tr>
<tr>
<td>4.15</td>
<td>Gravity space</td>
<td>47</td>
</tr>
<tr>
<td>4.11</td>
<td>Visualization of domains in the oak tree</td>
<td>48</td>
</tr>
<tr>
<td>4.16</td>
<td>Comparison to full simulation</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Avoid ξ’s minimum</td>
<td>57</td>
</tr>
<tr>
<td>5.2</td>
<td>Asynchronous Timestepping</td>
<td>60</td>
</tr>
<tr>
<td>5.3</td>
<td>Directed acyclic graph</td>
<td>61</td>
</tr>
<tr>
<td>6.1</td>
<td>Visualizing the Perlin wind field</td>
<td>64</td>
</tr>
<tr>
<td>6.2</td>
<td>Applying Perlin wind to the plant</td>
<td>66</td>
</tr>
<tr>
<td>6.3</td>
<td>Grand fir (Abies Grandis) in Perlin wind</td>
<td>68</td>
</tr>
<tr>
<td>7.1</td>
<td>Plant preprocessor</td>
<td>70</td>
</tr>
<tr>
<td>7.2</td>
<td>Select fronds and leaves</td>
<td>71</td>
</tr>
<tr>
<td>7.4</td>
<td>Connecting the domains</td>
<td>71</td>
</tr>
<tr>
<td>7.5</td>
<td>Resolving Loops</td>
<td>73</td>
</tr>
<tr>
<td>7.3</td>
<td>Select root domain</td>
<td>74</td>
</tr>
<tr>
<td>7.6</td>
<td>Botanical Simulator</td>
<td>77</td>
</tr>
<tr>
<td>7.7</td>
<td>Simulation control panel</td>
<td>78</td>
</tr>
<tr>
<td>7.8</td>
<td>Frequency tuning panel</td>
<td>80</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Simulation statistics (multidomain dynamics) . . . . . . . . . . . . . . . . . . 19
4.1 Timings for each pre-processing step . . . . . . . . . . . . . . . . . . . . . . 30
5.1 Simulation statistics (asynchronous parallel timestepping) . . . . . . . . . . . 62
Chapter 1

Introduction

A large fraction of our world is covered by vegetation. Botanical environments are both diverse and very common; therefore, they are crucial for special effects, games and virtual reality applications. The goal of this thesis is to design a real-time, physically-based simulator to compute large-deformation dynamics of plants.

Figure 1.1: Oregon white oak tree (Quercus Garryana) in strong randomized wind: the first fully mechanically simulated, botanically realistic and accurate, adult tree, anywhere in science, 871 branches, 120,000 leaves, 2,360,868 triangles, 9,520 reduced DOFs, simulation fps: 6 Hz
1.1 Plant Simulation Using Multi-domain Dynamics

We first propose a novel algorithm based on domain decomposition and model reduction which is capable of simulating complex, botanically accurate models with large deformation at interactive rates. Given a volumetric mesh of a plant, we decompose the mesh into several pieces called domains, build a reduced deformable model for each domain, simulate them separately, and connect them using inertia coupling. This makes model reduction deformable simulations much more versatile: localized deformations can be supported without prohibitive computational costs, parts e.g. twigs and leaves can be re-used and pre-computation time is shortened. Our method does not use constraints, and can handle large domain rigid body motion in addition to large deformations, due to our derivation of the gradient and Hessian of the rotation matrix in polar decomposition. We show real-time examples with multi-level domain hierarchies and thousands of reduced degrees of freedom (figure 1.1 and figure 1.2).

1.2 Plant Substructuring and Preprocessing

Existing domain decomposition methods do not address the problem of how to robustly and quickly pre-process many plants in the presence of imperfections in input geometry, which is a key challenge for the adoption of such methods in practice. We design a plant pre-processor
which is robust to non-manifold input geometry, gaps between branches or leaves, free-flying leaves not connected to any branch, small unimportant geometry (“debris”) left in the model, and plant self-collisions in the input configuration. We remove loops in the domain graph using a new user-assisted algorithm to select a minimum spanning tree in a general undirected graph. Our novel domain graph creation algorithm, instancing, and spanning tree selection procedure apply to any domain decomposition plant simulation method, including those that do not employ model reduction [96].

Our system supports plants represented using triangle meshes and alpha-masked billboards and exploits multi-core computation. Real-time fracture is supported, enabling our plants to shed leaves or drop fruits (Figure 1.2). We also present an approach to perform physically-based inverse kinematics, enabling the user to adjust the geometry of existing plants by dragging plant vertices. We simulated over 100 plants from diverse climates and geographic regions, including many broadleaf (deciduous) trees and conifers, bushes and flowers. Our system can simulate both simple plants and plants as complex as entire anatomically realistic adult trees with several hundreds of branches and over 100,000 leaves (see Figure 1.1).

1.3 Procedural Materials

Tuning simulation parameters is quite critical in order to generate vivid and plausible results. For geometrically complex plants, such tuning process is quite challenging since it is too complicated to do it manually in practice. In this thesis, we propose a series of procedural methods to automatically tune simulation parameters including material stiffness, timestep, and damping coefficients for botanical objects with large structural and geometrical complexity. These tuning techniques are generic and can be applied to any modal-based FEM simulation.

1.4 Asynchronous Parallel Timestepping

We have designed an asynchronous parallel timestepping algorithm to timestep the domains more efficiently. Since natural frequency is different for each domain, it inspires us to integrate
each domain at its own pace as opposed to using a fixed timestep for the entire object. This avoids the shortcomings from a synchronous integrator, in which a small, locked time step is carefully chosen in order to prevent from numerical instabilities due to the high stiffness variations encountered at different domains. Our total running time to produce animations of the same length (same total physical time) is therefore shorter than in the synchronous method. By exploiting the fact that no loop exists in the tree topology, we extend simulation to multi-cores, introduce buffers to decrease serializing dependencies, and optimize the structure of our multi-domain dynamics algorithm. We demonstrate that such parallelization greatly enhances simulator’s performance (5x).

1.5 Thesis Overview

The rest part of this thesis is organized as follows. In Chapter 2, we provide a comprehensive survey of related work. This survey includes 3 parts: plant modeling and simulation, finite element method (FEM) and model reduction, and domain decomposition (substructuring). In Chapter 3, we present a novel method which makes model reduction adaptive in space. We call it multi-domain dynamics algorithm. In this algorithm, we first decompose the deformable object into several components (called domains), then pre-process the reduced dynamics of each domain separately, and finally couple the domains using inertia forces. In Chapter 4, we demonstrate a robust plant preprocessor, which takes a static plant triangle mesh as the only input and pre-computes the mechanical models ready for the simulation use. In Chapter 5, we propose a series of procedural methods to tune simulation parameters for complex botanical objects. These properties include material stiffness, timestep, damping coefficients, etc. We design an asynchronous parallel timestepping method based on the multi-domain dynamics. In Chapter 6, we describe how we generate a randomized wind field in our simulator. We adopt the improved Perlin noise [77] which is widely used in texture synthesis and create a 4D randomized wind field. In Chapter 7, we provide a brief overview of our botanical simulation pipeline which includes two main parts: botanical preprocessor and simulator. In Chapter 8, we conclude our research and list several directions for future work.
Chapter 2

Related Work

2.1 Plant Modeling and Simulation

Plant modeling has a long history in computer graphics and we refer the reader to good survey work [23]. There are many strategies to create and edit plant geometry [78, 79]. Lindenmayer systems (L-systems) are probably the most well-known system for plant modeling and simulation [57, 67, 79]. Image-based techniques produce 3d plant models from photographs [80], and can be augmented with sketch-based interfaces to guide the plant development [36]. Plant geometry can also be guided by specific environment conditions [78]. While the early plant modeling methods were purely procedural [14, 72], user-assisted plant modeling [58] has been demonstrated to scale and produce realistic models of complex plants, as seen in the commercial Xfrog system [101]. We use Xfrog models extensively in our work, as well as models from other sources, such as the SpeedTree [37] system, and other online 3d model stores [95]. Our method works with “triangle soup” input meshes and is agnostic of the specific modeling approach.

Perhaps the simplest approach to animating plants is to not use physics at all; but drive the deformations kinematically using a stochastic wind [100]. Animations can also be created using pre-recorded motion graphs [42, 105], optionally combined with stochastic motion [104]. Physical simulation, however, has the advantage that it can provide dynamics, large motion, secondary motion and easier runtime control. A common physical approach is to model
branches as rigid rods, and connect them with angular springs [83, 98, 103]. Such an approach is usually augmented with a proper randomized wind model [73], and can be accelerated using level of detail techniques [10]. In nature, branches are flexible and their bending has been modeled using one-dimensional flexible beams [32, 34], and using the three-dimensional solid Finite Element Method (FEM) [59, 96]. Three-dimensional FEM simulations have the advantage that they can work with “polygonal soup” input geometry, simply by converting them to volumetric meshes. They can easily support volume preservation and spatially-varying material properties. They also automatically incorporate branch thickness (thicker branches are harder to bend) and non-straight (crooked) undeformed branch geometry. Our simulator builds upon the 3D solid Finite Element Method (FEM). It augments it with frame-aware domain decomposition to support large deformations, modularity, interactive design and fracture, and employs model reduction to increase the speed of computation.

2.2 Finite Element Method and Model Reduction

Deformable object simulation is a well-studied problem in computer graphics. We review approaches for interactive FEM and model reduction; please see [68] for a general survey. FEM simulations with complex geometry do not run at interactive rates. Interactivity can be achieved using multi-resolution geometric constructions [17, 22, 31], employing co-rotational elasticity [19, 65], the multigrid method [27], or by coarsening of meshes and their material properties [46, 69]. In our work, we employ model reduction (see [87] for a survey), and demonstrate that whenever the object can be decomposed into natural components, this can provide deformation-rich real-time simulations. The common theme in model reduction is to project the original state-space onto a low-dimensional subspace to arrive at a (much) smaller system having properties similar to the original system [51]. The idea has been employed in several disciplines outside of computer graphics, such as control theory [28], electrical circuit simulation [55], computational electromagnetics [88], microelectromechanical systems [38], fluid simulation [33, 50, 60] and nonlinear elasticity [2, 51, 70]. Many powerful reduction techniques have been designed, both for linear (in many cases time-invariant)
2.2 Finite Element Method and Model Reduction

systems [5], such as balanced truncation [64] and Krylov subspaces [30], and nonlinear systems [53, 81]. Despite this progress, model reduction of large-scale, high-dimensional (thousands of degrees of freedom) nonlinear systems is still a relatively new field [11]. In computer graphics, model reduction of nonlinear systems has been used for fast simulation of deformable solids [3, 8, 44, 49, 62] and fluids [94], and for fast control of such systems [7]. For deformable FEM offline simulations, Kim and James [47] showed how to apply online model reduction to adaptively replace expensive full simulation steps with reduced steps. One drawback of these systems has been that the reduction basis is global in space. Wicke and colleagues [99] extended Treuille’s fluid reduction method to several inter-connected reduced domains. Their approach is similar in spirit to ours, but works for fluids and does not directly apply to nonlinear elasticity. Our open-loop domain structures are related to recursive algorithms for articulated rigid body structures [26]. Most of these algorithms were developed for rigid objects. Motivated by the applications in robotics, Featherstone’s algorithm has been extended to deformable objects [15, 18, 86], to small-deformation simulations [18, 86], and deformable rods [12]. Our algorithm, however, applies to 3D solid deformable objects with irregular interfaces undergoing large deformations.

Application of model reduction to linear systems are common [76], and offer several advantages such as rapid simulation rates, as well as easy runtime control over the simulated and rendered level of detail. In the context of plant simulation, Stam [91] used a modal basis to compute noise in the frequency domain, and Diener [24] used a wind projection basis to increase computation speed. Low-dimensional (three of less) linear modal simulations on individual, fully decoupled, branches modeled as Euler-Bernoulli beams were presented in [32, 34], as well as methods to tune the models to match recorded real tree motion. These approaches produce quality high-frequency motion of trees, e.g., leaves and branches rapidly fluttering in the wind, and are as such complementary to our nonlinear model reduction method. Because they employ linear models, they cannot, however, support large deformation plant dynamics, characteristic of non-wooden (herbaceous) plants or trees blowing in strong winds. Research on how to quickly simulate plants undergoing large deformations, especially the ones with large structural and geometrical complexity, appears very limited. In our research, we develop
a robust system which allows us to quickly pre-process many complex botanically accurate models and launch them in our interactive simulation program.

### 2.3 Domain Decomposition

The idea to decompose a deformable object into several interconnected components (domains), each of which can be simulated separately, is well-known in science. It is usually referred to as *domain decomposition* [93], or, especially in case of repetitive geometry, also *substructuring* [25, 71]. Many such methods do not employ reduction, but merely divide the object so that each domain can be assigned to a different processor node, or data for repetitive substructures can be reused [6, 82]. Perhaps the simplest form to add reduction to domain decomposition is to compute the static equilibrium of each domain, under $x, y, z$ perturbations of each interface vertex, and then restrict the domain deformations to a linear combination of those shapes (*static condensation* [92]). With complex geometry, however, interfaces themselves can be high-dimensional, leading to a large number of basis vectors and slow simulation times. Alternatively, one can compute the linear vibration modes of each structure, under the boundary condition that the interfaces are held fixed [21]. This method, called *component mode synthesis* [84], has been popular with simulations of deformations of superstructures, most notably in aerospace engineering (e.g., airplanes, space satellites) [75]. These previous methods, however, only simulated small, linearized deformations of each domain. It is not straightforward to extend them to large deformations because the resulting large interface rotations seemingly require modes to rotate, which invalidates precomputation. In our paper, we show how these obstacles can be avoided, yielding a component mode synthesis method supporting (1) large deformations within each domain, and (2) large (finite) rotations of the domain interfaces. This makes substructuring much more useful in computer graphics applications requiring large deformations. Domain decomposition for deformable models has also been previously applied in computer graphics, but only for small domain deformations and with running times dependent on the number of domain and interface vertices. For example, a linear quasi-static application using Green’s functions has been presented in [41], whereas Huang and colleagues [35]
exploited redundancy in stiffness matrix inverses to combine linear FEM with domain decomposition.

Because plants typically consist of well-defined subparts with only limited interaction, domain decomposition is a natural approach to tackling plant motion complexity. Twigg and Kačić-Alesić [96] simulated deformable objects consisting of many parts, including trees, by gluing them together using the Procrustes transform. Their method simulates each deformable part in the full space. Such simulations can be accelerated by orders of magnitude using model reduction, which makes it possible to timestep complex plants at interactive rates. Recently, the combination of domain decomposition and model reduction has received significant attention in the literature [9, 48, 102]. Our research [9], for example, employed gradients of rotation matrices computed in polar decomposition, for structures that do not have loops in the domain graph. Kim and James [48] efficiently simulated deformations of characters using inter-domain spring forces. Recent work of [102] uses modes obtained from unit displacements of interface vertices, and inertia modes, combined with modal warping.

Our method does not require a skeleton, pre-existing motion, or well-defined domain interfaces and it supports fracture, inverse kinematics, frequency tuning, and bottom-up meshing. Different from [48], a global volumetric mesh is not necessarily needed, as each domain is meshed individually, enabling component re-use and modularity.
Chapter 3

Multidomain Dynamics in Reduced Space

3.1 Background: Model Reduction and Domain Decomposition

Model reduction is a popular method for deformable model simulation, mainly because it can approximate complex physical systems at a low computational cost. The key idea of model reduction is to project the high-dimensional equations of motions to a suitably chosen low-dimensional space where the dynamics have properties similar to the original system, but can be timestepped much more quickly [51]. Real-time projection-based model reduction for deformable objects has, however, suffered from an important limitation: the reduction basis is global in space and time. Such bases require a large number of modal vectors to capture local deformations. More importantly, because nonlinear modal elasticity requires implicit integration for stability, and because all global basis vectors overlap in space, each timestep requires (at least) solving a \( \hat{r} \times \hat{r} \) dense linear system costing \( O(\hat{r}^3) \), where \( \hat{r} \) is the number of basis vectors. In practice, this has limited real-time nonlinear reduced simulations to less than (approximately) one hundred degrees of freedom [3].

In this Chapter, we present an approach to make model reduction adaptive in space, by decomposing the deformable object into several components (the domains, see Figure 3.1). We pre-process the reduced dynamics of each domain separately, and then couple the domains
using inertia forces. Assuming a decomposition free of loops, the resulting system supports large deformation dynamics both globally and locally within each domain. For the geometrically nonlinear FEM material model, the resulting nonlinear system can be timestepped at rates independent of the underlying geometric or material complexity. With exact reduced internal force evaluations on \(d\) domains with \(r\) degrees of freedom each, the running time of one timestep of our method is \(O(dr^4) \ll O(\hat{r}^4)\), for \(\hat{r} = dr\), and could be further decreased to \(O(dr^3)\) using approximate reduced forces [3].

The idea of decomposing a deformable object for efficient simulation has been previously extensively explored in the engineering community, usually under the names of domain decomposition and substructuring. However, previous methods either did not pursue reduction in each domain, or limited the domains to small deformations. Our method is related to the well-known Featherstone’s algorithm for linked rigid body systems, but differs from it by simulating large deformations involving large interface rotations, combined with model reduction. Like the Featherstone’s algorithm supports kinematic chains of arbitrary length, we make no assumptions on the depth of hierarchies. We approximate subtree inertia using mass lumping, which gives us fast and stable real-time large deformations rich in local detail. Our method supports instancing: a single object can be pre-processed once and replicated many times saving on precomputation and runtime costs. Rigid domains \((r = 0)\) are supported, and can be arbitrarily mixed with reduced-deformable domains. The method also supports unanchored objects undergoing free-flight motion.

### 3.2 Kinematics

Our method uses reduction to simulate geometrically nonlinear FEM deformations of a 3D volumetric mesh. We show examples with linear tetrahedral elements and trilinear voxel (cube) elements. Let a volumetric mesh be decomposed into \(d\) connected and mutually disjoint sets of elements \(\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_d\) (the domains, see Figure 3.1). The common surface where two domains \(i\) and \(j\) meet is called the interface, \(I_{ij}\). Although in principle an object could be decomposed arbitrarily, domain decomposition is most effective when the domains form a natural
3.2 Kinematics

development of the object, such as, for example, separating the space station modules, panels and antennas into separate domains (Figure 3.1). In such cases the interfaces are often small or deform mostly rigidly (see Figure 3.2, right), which we exploit to define our low-dimensional kinematic model. We first form the domain graph, where each domain is one node, and nodes are connected if they share a common interface (see Figure 3.1). We assume that there are no cycles in the domain graph (graph is a tree) but place no restriction on maximum node degree. We direct the graph by picking one domain as the root node, which then uniquely directs all edges by traversing the domains from the root to the leaves. We set the root domain to the domain rooted to the ground for fixed objects, or a central/largest domain for free-flying objects, which tends to minimize tree depth.

Figure 3.1: Domain decomposition: (a) domains, (b) domain graph, (c) 48 domains (space station), (d) domain graph (space station).

Figure 3.2: Effect of system and interface forces: Left: Pulling on the small stem causes large secondary motion (bloom, leaves). Right: relative deviation of the interface transformation $A_{ij}$ from rotation $R_{ij}$ (Frobenius norm, for all the 11 interfaces). The deviations are small.

We simulate each domain as a nonlinear reduced-deformable object, with its own specific
reduced basis $U_i$ of size $r_i \geq 0$. All our reduced models are precomputed under the boundary condition that the domain is held fixed at the interface to the parent, which is consistent with the interface rigidity assumption. The deformation of each domain in its local frame of reference, is given as a linear combination of the modes of domain $i$, $u_i = U_i q_i$, where $u_i$ contains the 3D deformations of all the vertices of domain $i$, for the reduced-coordinate vector $q_i \in \mathbb{R}^{r_i}$. Deformations $u_i$ are expressed in a local frame of reference $F_i$ of each domain, which we define below. Frames are necessary because parts of the mesh may undergo large rotations, whereas modal models are poorly suited to represent large rotations. At first, it may seem possible to avoid frames by including all affine transformations into each basis $U_i$. However, the remaining vectors in $U_i$ (the non-rigid deformations) would then need to be rotated synchronously with the domain rotation at runtime, leading to a time-dependent basis and a significant additional computational cost.

To define the frames, we first collect all individual vectors $q_i$ into a global vector $q \in \mathbb{R}^{r_1 + \ldots + r_d}$. The frame computation then proceeds from the tree root to the leaves. Frame $F_0$ is the world coordinate frame for fixed objects, and the global rigid body motion frame for free-flying objects. For each child domain $j$ of domain $i$, we define frame $F_j$ as the best fitting frame to the interface $I_{ij}$. We do so by specifying its position $x_{ij} \in \mathbb{R}^3$ and rotation $R_{ij} \in \mathbb{R}^{3 \times 3}$, relative to frame $F_i$, and expressed in the coordinate axes of frame $F_i$, by fitting the best rigid transformation that transforms vertices of interface $I_{ij}$ from their rest positions to current positions given by $q_i$ (see Figure 3.3). Let $v_{ij}^1, \ldots, v_{ij}^m$ be the vertices of domain $i$ that are on the interface to child domain $j$. We can weight the vertices according to the surface area (or mesh volume) locally belonging to each vertex, arriving at weights $w_{ij}^1, \ldots, w_{ij}^m$. In domain $i$, each vertex $k$ deforms according to a $3 \times r_i$ submatrix of $U_i$, denoted by $U_{ij}^k$. We make $x_{ij}$ track the centroid of $I_{ij}$:

$$x_{ij} = \frac{1}{\sum_{k=1}^m w_{ij}^k} \sum_{k=1}^m w_{ij}^k (X_{ij}^k + U_{ij}^k q_i) \equiv \hat{x}_{ij} + \hat{a}_{ij} q_i,$$

(3.1)

where $X_{ij}^k$ is the rest position (in $F_i$) of vertex $v_{ij}^k$, and the matrix $\hat{a}_{ij} \in \mathbb{R}^{3 \times r_i}$ can be precomputed. In order to fit the rotation, we need to align the interface vertices to their deformed
positions as best as possible using a rigid transformation (after subtracting centers). We do so by first computing the covariance matrix \[ A_{ij}(q_i) = B_{ij}(q_i)C_{ij}^{-1}\hat{R}_{ij} \equiv \hat{R}_{ij} + \sum_{\ell=1}^{r_i} A_{ij}^\ell q_i, \] where \[ B_{ij}(q_i) = \sum_{k=1}^{m} w_{ij}^k ((X_{ij}^k - \hat{x}_{ij}) + (U_{ij}^k - \hat{a}_{ij})q_i) \left( X_{ij}^k - \hat{x}_{ij} \right)^T, \] \[ C_{ij} = \sum_{k=1}^{m} w_{ij}^k (X_{ij}^k - \hat{x}_{ij}) (X_{ij}^k - \hat{x}_{ij})^T, \] and the fixed matrices \( A_{ij}^\ell \in \mathbb{R}^{3\times3} \) can be precomputed. Here, \( \hat{R}_{ij} \) is the rotation of interface \( I_{ij} \) in the rest configuration, relative to rest frame \( \mathcal{F}_i \). We then perform polar decomposition to extract the best-fitting rotation \( R_{ij} \):

\[ A_{ij}(q_i) = R_{ij}(q_i)S_{ij}(q_i), \]

where \( S_{ij} \) is a symmetric \( 3 \times 3 \) matrix. We note that rigid transformations cannot be made to fit deformable interfaces in general, leading to discrepancies in the domain meshes at the interface. In practice, however, the interface transformation are often very close to rigid and the discrepancies are small (Figures 3.2). If interface vertices lie in the same plane in the rest configuration, matrix \( C_{ij} \) becomes degenerate. Let \( 0 \leq \lambda_1 \leq \lambda_2 \leq \lambda_3 \) be the eigenvalues of \( C_{ij} \). If \( \lambda_1 < \varepsilon \lambda_3 \) (we use \( \varepsilon = 0.001 \)), we modify the precomputation of \( A_{ij}^\ell \) by adding an extra point at \( \hat{x}_{ij} + \eta N \), where \( N \) is the eigenvector for \( \lambda_1 \) (interface normal). The value \( \eta = \sqrt{\varepsilon \lambda_3 - \lambda_1} \) increases the smallest eigenvalue of \( C_{ij} \) to \( \varepsilon \lambda_3 \), and gave stable planar interfaces in our examples.

Once the root frame \( \mathcal{F}_0 \) and transformations \( (x_{ij}, R_{ij}) \) are known for all interfaces \( I_{ij} \), we can easily compute world-coordinate expressions for all frames \( \mathcal{F}_i, i = 0, \ldots, d - 1 \). We note that the frames \( \mathcal{F}_i \) are completely determined by \( q \) and \( \mathcal{F}_0 \); i.e., the frames are not separate inde-
pendent simulation parameters. An alternative would be to keep them separate and simulate
the combined frame-deformable system, but doing so would require constraints to keep the
domains connected, turning the system into a differential-algebraic equation and leading to
standard problems of constraint drift. Our construction, in turn, avoids constraints. In the next
section, we shall derive the dynamics that govern $q$, using a mass-lumped formulation running
in time linear in the number of domains. This is similar to Featherstone’s algorithm, but for
flexible objects undergoing large deformations.

For dynamics, it is necessary to compute linear velocity $v_{ij}$, linear acceleration $a_{ij}$, angular
velocity $\omega_{ij}$, and angular acceleration $\alpha_{ij}$ of each frame $F_j$, relative to $F_i$ and expressed in
$F_i$, based on $q_i, \dot{q}_i, \ddot{q}_i$. Linear quantities are simply $v_{ij} = \dot{a}_{ij} \dot{q}_i$ and $a_{ij} = \ddot{a}_{ij} \ddot{q}_i$.

To compute the angular quantities, first differentiate Equation 3.2:

$$\dot{A}_{ij} = \sum_{\ell=1}^{r_i} A_{ij}^{\ell} \dot{q}_i^{\ell}, \quad \ddot{A}_{ij} = \sum_{\ell=1}^{r_i} A_{ij}^{\ell} \ddot{q}_i^{\ell}. \quad (3.6)$$

Next, we compute the first and second derivatives of the polar decomposition rotation matrix
$R_{ij}$ with respect to $q_i$. We derived a general formula for $\dot{R}(t)$ and $\ddot{R}(t)$, where $R(t)$ is the rotation
in polar decomposition of a $3 \times 3$ matrix $A(t) = R(t)\hat{S}(t)$ that depends on a scalar parameter
$t \in \mathbb{R}$ (usually time, but can be any parameter):

$$G = (\text{tr}(\hat{S})I - \hat{S})R^T \in \mathbb{R}^{3 \times 3}, \quad \omega = G^{-1} \left( 2 \text{skew}(R^T \dot{A}) \right) \in \mathbb{R}^3, \quad (3.7)$$

$$\dot{R} = \tilde{\omega}R, \quad \dot{\hat{S}} = R^T (\dot{A} - \dot{\hat{S}}), \quad \ddot{R} = \tilde{\omega} \dot{R} + \omega^2 R, \quad (3.8)$$

$$\dot{\omega} = G^{-1} \left( 2 \text{skew}(R^T (\dot{A} - \dot{\hat{A}})) - (\text{tr}(\hat{S})I - \dot{\hat{S}})R^T \omega \right). \quad (3.9)$$

Here, $\tilde{\omega}$ denotes the $3 \times 3$ skew-symmetric matrix corresponding to a vector $\omega \in \mathbb{R}^3$, i.e.,
$\tilde{\omega}x = \omega \times x$ for all $x \in \mathbb{R}^3$. Similarly, skew($A$) denotes the unique skew-vector $\omega \in \mathbb{R}^3$ so that
$\tilde{\omega} = (A - A^T)/2$. The derivation of Equations 3.7 - 3.9 is given in Section 3.5. Evaluation
of $\omega, \dot{R}, \dot{\hat{S}}, \ddot{R}$ requires solving the $3 \times 3$ nonsymmetric linear system given by matrix $G$. Because this system is nonsingular whenever $A$ is nonsingular, which is the case for interfaces
that deform mostly rigidly, computing $G^{-1}$ is stable. We can now apply Equations 3.7-3.9
to $A_{ij}(t) = A_{ij}(q_i(t))$ and its time derivatives as given by Equations 3.2 and 3.6, yielding $\omega_{ij}$ and $\alpha_{ij}$. We note that such matrix decomposition gradients have been previously explored for singular value decomposition [61, 96]. With SVD, singularities in the decomposition gradient occur whenever two singular values are equal, e.g., even if $A$ is identity, and would, unlike polar decomposition, require additional treatment.

### 3.3 Dynamics

We now give the equations of motion for $q$, under the kinematic model of Section 3.2. These equations simulate the coupled motion of all domains. Each domain follows the equation

$$M_i \ddot{q}_i + D_i \dot{q}_i + f^\text{int}_i(q_i) = f^\text{ext}_i + f^\text{sys}_i + \sum_{j \text{ is child of } i} f^\text{itf}_{ij}, \quad (3.10)$$

where $M_i$ is the reduced mass matrix (constant matrix), $D_i = D_i(q_i)$ is the reduced damping matrix, and $f^\text{int}_i(q_i)$ are the reduced nonlinear internal elastic forces of domain $i$. The terms $f^\text{sys}_i, f^\text{itf}_i, f^\text{ext}_i$ contain the reduced system forces due to motion of ancestor domains, interface forces of child domains of $i$ arising due to subtree inertia, and external forces, respectively.

Equation 3.10 is standard in model reduction; it is obtained by projecting a full (geometrically nonlinear) FEM deformable model to a chosen basis $U_i$. We use the modal derivative basis [8] because the computation is automatic and does not require any presimulation. We address non-inertiality in by computing non-inertial frame of reference system forces $f^\text{sys}_i$ due to the motion of frame $i$ as imposed by its parent domain.

**System Forces:** Equation 3.10 is expressed in frame $\mathcal{F}_i$ which is non-inertial (accelerates through time). An observer rigidly attached to $\mathcal{F}_i$ can correctly simulate deformations if she adds the resulting system forces $f^\text{sys}_i$ to the equations of motion of her domain (see, e.g. [40]). Let $X$ be a material point in frame $\mathcal{F}_i$. The world-coordinate velocity and acceleration of $X$, ...
expressed in the coordinate frame \( \mathcal{F}_i \), equal [85]

\[
\begin{align*}
\dot{v}(X) &= v_i + \omega_i \times X + \dot{X} \\
\ddot{a}(X) &= a_i + \alpha_i \times X + \omega_i \times (\omega_i \times X) + 2 \omega_i \times \dot{X} + \ddot{X},
\end{align*}
\]

where \( v_i, \omega_i, a_i, \alpha_i \) are the world-coordinate velocity, angular velocity, acceleration and angular acceleration of frame \( \mathcal{F}_i \), respectively, expressed in the frame \( \mathcal{F}_i \). The system forces are distributed volumetrically throughout the domain. When projected to the low-dimensional space of each domain, they are

\[
f^\text{sys}_i = -\int_{\Omega_i} \rho_i(X) U_i^T(X) a(X) dV,
\]

where \( \rho_i(X) \) is mass density at \( X \), \( U_i \in \mathbb{R}^{3 \times r_i} \) are the spatially-varying modes, and \( a(X) \) is acceleration at \( X \). The \( \ell \)-th component of \( f^\text{sys}_i \in \mathbb{R}^{r_i} \), for \( \ell = 1, \ldots, r_i \), can be expanded to

\[
f^\text{sys}_i \ell = W^1_{i\ell} a_i - W^2_{i\ell} \alpha_i + (W^3_{i\ell} + q^T_{i\ell} W^4_{i\ell}) \| \omega_i \|^2 -
\]

\[
-(\omega_i^T W^5_{i\ell} + 2 q^T_{i\ell} W^6_{i\ell}) \omega_i - q^T_{i\ell} W^6_{i\ell} \alpha_i - (\omega_i^T \omega_i)^T : \sum_{p=1}^{r_i} W^7_{i\ell} q^P_{i\ell},
\]

for constant precomputable coefficients \( W^j_{i\ell} \) (Section 3.6). Notation \( A : B \) denotes component-wise matrix dot product. The evaluation of \( f^\text{sys}_i \) requires \( O(r^2_i) \) flops, and is fast in practice.

**Interface Forces:** We model reduced interface forces as

\[
f^\text{if}_{ij} = -M_{ij} \dot{q}_i + f^0_{ij}, \quad \text{for}
\]

\[
M_{ij} = \bar{m}_j a^T_{ij} \hat{a}_{ij}, \quad f^0_{ij} = \hat{a}_{ij} \left( R_{ij} \bar{f}^\text{ext}_j -
\]

\[
-\bar{m}_j (a_i + \alpha_i \times x_{ij} + \omega_i \times (\omega_i \times x_{ij}) + 2 \omega_i \times v_{ij}) \right),
\]

where \( \bar{m}_j \) and \( \bar{f}^\text{ext}_j \) are the total mass and net sum of external forces (expressed in \( \mathcal{F}_j \)) in the subtree rooted at \( j \) (call it \( \overline{D}_j \)). Note that \( \bar{m}_j \) is constant and precomputable, but could vary.
at runtime, e.g., if domains fracture, or are replaced (modularity). Equation 3.15 is similar to Featherstone’s recursive terms. The terms $M_{ij}$ and $f_{ij}^0$ model the mass inertia of $D_j$, by assuming that the mass is lumped at $I_{ij}$. The term $f_{ij}^0$ also models the effect (to domain $i$) of external forces applied in $D_j$. This approximation gives stable motion and keeps the system matrix symmetric, and we found it reasonable for examples with limited domain graph depth.

**Integration:** Equation 3.15 transforms Equation 3.10 into

$$
\left( M_i + \sum_j M_{ij} \right) \ddot{q}_i + D_i \dot{q}_i + f_{i}^{\text{int}}(q_i) = f_i^{\text{ext}} + f_i^{\text{sys}} + \sum_j f_{ij}^0.
$$

(3.17)

We timestep Equation 3.17 in time linear in the number of domains. The algorithm first constructs the frames for all domains, then timesteps each domain using semi-implicit Newmark integration for stability [8], and finally updates relative frame kinematics (see Algorithm 1). The frame of the root domain can be the world-coordinate frame if the root domain is fixed to the ground, or its frame can be floating and affected by the forces and torques of the subdomains for free-flying objects.

---

**Algorithm 1: Multidomain reduced dynamics**

1. Procedure **Simulation timestep**
2. **Input:** values of $\hat{q}^{(k)} = (q, \dot{q}, \ddot{q})$ and frames $\mathcal{F}^{(k)}$ at timestep $k$, external forces $f_i^{\text{ext},(k+1)}$ at timestep $k + 1$, timestep size $\Delta t$
3. **Output:** values of $\hat{q}^{(k+1)}$ and $\mathcal{F}^{(k+1)}$ at timestep $k + 1$
4. \textbf{begin}
5.   \textbf{for} all domains $i$, leaves to root \textbf{do} Assemble $\overline{f}_i^{\text{ext}}, \overline{m}_i$
6.   \textbf{for} all domains $i$, root to leaves \textbf{do}
7.     \textbf{for} all child domains $j$ of $i$ \textbf{do}
8.       Compute $\omega_i, a_i, \alpha_i, \mathcal{F}_i^{(k+1)}$ (Eq. 3.11, 3.12)
9.       Compute $f_i^{\text{sys}}$ (Eq. 3.14)
10.      \textbf{for} all child domains $j$ of $i$ \textbf{do}
11.         Compute $M_{ij}, f_{ij}^0$ (Eq. 3.16)
12.      \textbf{end for}
13.   \textbf{end for}
14. \textbf{end for}
15. \textbf{for} all children $j$ of $i$ \textbf{do}
16.   Use $\hat{q}_i^{(k+1)}$ to compute $x_{ij}, R_{ij}, v_{ij}, \omega_{ij}, a_{ij}, \alpha_{ij}$
17. \textbf{end for}
18. \textbf{end}

---
3.4 Results

<table>
<thead>
<tr>
<th>Example</th>
<th>vol-vtx</th>
<th>vol-el</th>
<th>rend-vtx</th>
<th>rend-tri</th>
<th>d</th>
<th>r</th>
<th>D</th>
<th>pre</th>
<th>rt:StVK</th>
<th>rt:total</th>
<th>S</th>
<th>$u_t = U_i q_i$</th>
<th>fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>flower (tet)</td>
<td>2,713</td>
<td>7,602</td>
<td>6,675</td>
<td>12,868</td>
<td>12</td>
<td>240</td>
<td>3</td>
<td>0.6 min</td>
<td>22 %</td>
<td>5.0 msec</td>
<td>1</td>
<td>0.43 msec</td>
<td>69 Hz</td>
</tr>
<tr>
<td>SIGGRAPH (tet)</td>
<td>18,945</td>
<td>83,753</td>
<td>10,463</td>
<td>20,934</td>
<td>15</td>
<td>160</td>
<td>5</td>
<td>3.1 min</td>
<td>45 %</td>
<td>1.7 msec</td>
<td>1</td>
<td>2.5 msec</td>
<td>83 Hz</td>
</tr>
<tr>
<td>dragon (tet)</td>
<td>46,736</td>
<td>160,553</td>
<td>38,625</td>
<td>77,250</td>
<td>40</td>
<td>454</td>
<td>5</td>
<td>5.1 min</td>
<td>55 %</td>
<td>1.6 msec</td>
<td>1</td>
<td>7.0 msec</td>
<td>40 Hz</td>
</tr>
<tr>
<td>space station</td>
<td>219,058</td>
<td>107,556</td>
<td>177,691</td>
<td>248,521</td>
<td>48</td>
<td>921</td>
<td>4</td>
<td>0.8 min</td>
<td>75 %</td>
<td>4.3 msec</td>
<td>1</td>
<td>25.5 msec</td>
<td>14 Hz</td>
</tr>
<tr>
<td>oak tree (tet)</td>
<td>262,363</td>
<td>626,734</td>
<td>578,801</td>
<td>838,704</td>
<td>1435</td>
<td>11,972</td>
<td>5</td>
<td>1.0 min</td>
<td>52 %</td>
<td>29 msec</td>
<td>1</td>
<td>12.2 msec</td>
<td>5 Hz</td>
</tr>
</tbody>
</table>

Table 3.1: Simulation statistics for #volumetric and rendering mesh vertices, elements and triangles (vol-vtx, vol-el, rend-vtx, rend-tri), #domains (d), total # of reduced DOFs (r), tree depth (D), precomputation time (pre), #simulation steps per graphics frame (S), constructing deformations for rendering ($u_t = U_i q_i$, once per frame), frame rate (fps), simulation step time (rt:total), and its breakdown in terms of time stepping reduced dynamics of individual domains (rt:StVK) and the rest (rt:core, including polar decomposition, gradient and Hessian, system and interface forces, frames). Machine specs: Intel Core i7-980X, 6-Core, 3.33 GHz, 10 GB memory. Only a single core was used.

Figure 3.4: Model reduction with a large number of localized degrees of freedom: Left: non-linear reduced simulation of an oak tree, all leaves are deformable domains (41 branches ($r = 20$), 1394 leaves ($r = 8$), $d = 1435$ domains, $r = 11,972$ total DOFs) running at 5 fps. Right: simulation detail.

In our first example we show a complex oak tree (41 branches, 1394 leaves) deformed in the wind and undergoing large deformations (see Figure 3.4). This example greatly uses substructuring, as the leaves all use a single mesh (rotated and translated into the proper place). The leaf mesh is pre-processed and reduced only once. The copies use pointers to the single datastructure, leading to short precomputation times (Table 3.1). Similarly, the 41 branches only belong to 5 distinct classes, translated, rotated and scaled (uniformly) into their place.
3.4 Results

Figure 3.5: Instancing (Space Station): The repeated panel copies are instanced, and can all bend independently, as can all major structure parts.

Scalings can be handled efficiently by observing that for a uniform scaling factor $s$, the basis matrix does not change, whereas the frequency spectrum scales by $1/s$. We also pursued instancing in the space station example (Figure 3.5).

Figure 3.6: Our method supports localized deformations. End-effector domains often deform most due to largest system forces.

The number of modes in each domain can be chosen automatically, by setting a total number of modes $\hat{r}$, and assigning to each domain the number of modes proportional to its number of elements. One good choice is a logarithmic distribution: $r_i \propto \log(\#\text{elements}(i))$ (dragon example, Figure 3.6). The SIGGRAPH example (Figure 3.6) demonstrates that our method sup-
ports free-flying motion. This is achieved by pre-processing a “free-fly” reduced deformable model [8] for the root domain, and then using the external forces to integrate the rigid body motion for the entire object. Our method supports rigid domains \( r_i = 0 \), so it can combine rigid and deformable objects: the 8 letters of “SIGGRAPH” are connected by 7 rigid links.

**Figure 3.7: Similar trajectories, but different frequencies:** Top and middle: Bird’s-eye view on the trajectories of two flower vertices: top of primary (A) and secondary bloom (B). Same scale used for A and B. Rest and extreme poses are indicated. Ground truth (G)=solid black, interface lumping (L1)=dashed green, center of mass lumping (L2)=dashed red. Bottom: x-dof of A versus time.

**Accuracy:** Figure 3.7 compares our method to an unreduced single-domain geometrically nonlinear FEM simulation (ground truth). We provide a comparison to two variants of our method: (L1) interface lumping (Equation 3.16), and (L2) center-of-mass lumping where the mass of the subtree is lumped not at the interface, but at the center of mass of the subtree. Method L2 can be implemented by adding additional terms to Equation 3.16, guided by Equation 3.12. The ground truth (G) was computed offline and is 55x slower than L1 and L2. All three simulations use the same material parameters, timestep, fixed vertices, and the initial condition: a velocity aligned with the first eigenmode of the entire mesh, sufficient to bend the flower stem by about 45 degrees. We found that the three methods give similar trajectories, but differ in oscillation frequencies: L1 and L2 gave 2x and 1.5x higher lowest natural frequency than G, respectively. In general, reduced simulations of solids lack detailed DOFs, resulting
in small increases in natural frequency (“artificial stiffening”). With multidomain reduced dynamics, however, the increase is largely due to mass lumping, similar to how a pendulum with mass lumped close to the pivot oscillates at a higher frequency than if the mass was distributed further away. L2 matches G more closely than L1 because lumping at the subtree center better approximates the actual mass distribution.

There is a pyramid of methods that model the subtree inertia progressively better, at the cost of additional implementation complexity. All these methods take the form of Equation 3.15, but differ in how $M_{ij}$ and $f_{ij}^0$ are computed. All examples in Table 3.1 use method L1, which we found to be the simplest approach producing stable, deformation-rich results. In L1, each domain feels the total weight of the attached subtree, but not the rotational or deformable inertia. Variant L2 improves the accuracy for physically long domains. In some examples, however, we observed that L2 suffers from instabilities; we attribute this to the quickly time-varying matrices $M_{ij}$ in method L2. At a significant additional implementation complexity, one could compute correct, non-lumped subtree inertia for kinematic chains with arbitrary tree depth (cf. [85]). Such $M_{ij}$ and $f_{ij}^0$ would fully parallel Featherstone’s algorithm, but may require modifications to the integrator to maintain stability.

With uniform material parameters, small mesh parts (e.g., protrusions) vibrate at higher frequencies than the rest of the mesh, which manifests as little or no deformations in those regions even with the ground truth (unless poked explicitly). The frequency content of each domain can be linearly scaled at runtime without redoing the precomputation, simply by scaling the domain’s precomputed reduced internal forces. In the flower example (Figure 3.2), we adjusted the frequencies of the leaves so that the leaves gave interesting large deformations when pulling on the stem. Similarly, we made the horns, tail, spike and mouth of the dragon softer than the rest of the mesh, to cause larger deformations in those regions (Figure 3.6).

**Rendering:** We render triangle meshes embedded into volumetric meshes. Although the volumetric mesh for the entire object is a manifold mesh in the rest configuration, the domains slightly separate at the interfaces under deformation. At first, we anticipated this to be problematic – however, we found that it can be very easily handled with techniques similar to those
employed in discontinuous Galerkin FEM [45]. We can establish $C^0$ continuity simply by averaging the two copies of each interface vertex (see Figure 3.8), whereas $C^1$ continuity could be achieved via moving least square (MLS) embeddings [45]. Once a consistent volumetric mesh deformation is computed, it is transferred to the embedded triangle mesh using barycentric interpolation. Because our bases are local in space, computing the vertex deformations via equation $u_i = U_i q_i$ only involves matrices $U_i$ with a small number of columns, leading to small memory footprints and fast computation.

![Figure 3.8: $C_0$ embedding is effective with nearly-rigid interfaces:](image)

(a), (c): individual domain meshes, for two representative deformations. Domain gaps (black) are generally small; most are sub-pixel size. (b), (d): Mesh deformations with $C_0$ continuity.

### 3.5 Polar Decomposition Rotation Gradient

Let $A = A(t)$ be a $3 \times 3$ matrix that depends on a scalar parameter $t \in \mathbb{R}$. For any $t$, one can perform polar decomposition, $A(t) = R(t)S(t)$, where $R$ is orthogonal and $S$ is symmetric positive semi-definite. Fix $t_0 \in \mathbb{R}$. We first shift the problem by defining

$$B(t) = R^T(t_0)A(t) = \left(R^T(t_0)R(t)\right)S(t).$$

(3.18)

Because $R^T(t_0)R(t)$ is identity for $t = t_0$, its derivative at $t = t_0$ must be a skew-symmetric matrix $\tilde{\omega}$ for some $\omega \in \mathbb{R}^3$. By differentiating Equation 3.18 by $t$, and setting $t = t_0$, one obtains

$$\dot{B}(t_0) = R^T(t_0)\dot{A}(t_0) = \tilde{\omega}S(t_0) + \dot{S}(t_0).$$

(3.19)
We now replace $t_0$ with $t$, and apply the skew operator (Section 3.2) to both sides of Equation 3.19. This causes the symmetric term $\dot{S}$ to drop and yields 3 linear equations for the 3 components of $\omega$ (Equation 3.7). In order to derive $\dot{\omega}$ and $\dddot{R}$, differentiate both sides of Equation 3.7 with respect to $t$ (note that $G$ and $\omega$ depend on $t$). After rearranging, one obtains a $3 \times 3$ linear system for $\dot{\omega}$ (Equation 3.9).

### 3.6 Appendix: System Force Integrals

The constants of Equation 3.14 are integrals over each domain $D_i$:

\begin{align*}
W_{it}^1 &= -\int_{D_i} \rho U_{i\ell}^T dV \in \mathbb{R}^3, \\
W_{it}^2 &= \int_{D_i} \rho \dot{X} U_{i\ell}^T dV \in \mathbb{R}^3, \\
W_{it}^3 &= \int_{D_i} \rho U_{i\ell}^T X dV \in \mathbb{R}, \\
W_{it}^4 &= \int_{D_i} \rho U_{i\ell}^T U_{i\ell}^T dV \in \mathbb{R}^{n_i}, \\
W_{it}^5 &= \int_{D_i} \rho U_{i\ell}^T X dV \in \mathbb{R}^{3 \times 3}, \\
W_{it}^6 &= \int_{D_i} \rho U_{i\ell}^T \dddot{U}_{i\ell}^\ell dV \in \mathbb{R}^{n_i \times 3}, \\
W_{it}^7 &= \int_{D_i} \rho U_{i\ell}^T U_{i\ell}^T dV \in \mathbb{R}^{3 \times 3},
\end{align*}

where $U_{i\ell} \in \mathbb{R}^3$ is the $\ell$-th column of mode $U_i(X) \in \mathbb{R}^{3 \times n_i}$. 
Chapter 4

Plant Preprocessing and Simulation

Previous computer graphics research on plants has focused on plant geometry creation and appearance (rendering), as well as efficient simulation. Simulation of complex plants is challenging, however, because plant meshes are typically designed for rendering, not simulation. In this chapter, we demonstrate how to robustly and quickly pre-process complex plants in the presence of imperfections in the input geometry, for subsequent fast physically based simulation. Because plants naturally decompose into their constituent parts (branches, twigs, leaves, etc.), this makes botanical objects fit for our multi-domain dynamics simulator described in the previous chapter. Such authoring of simulation-ready plants augments and completes simulation, by making it easy to apply our model-reduction based simulator to general, complex plant models.

4.1 Plant Preprocessing

Given an input triangle mesh of a plant, we present an efficient and easy to use user-assisted pipeline to organize the plant into domains, create the domain hierarchy, create the simulation meshes, and pre-process reduced models (Figure 4.1). The end result is a plant mechanical model that can be rapidly simulated using our model reduction simulator (previous chapter). All of the steps of our pipeline except the specific model reduction pre-processing in Step 10 apply generally to plant domain decomposition, and could be used with other domain decom-
Figure 4.1: Preparing mechanical models ready for the simulation: (a) Triangle mesh of an European larch (Larix Decidua). This mesh was designed for rendering and is not suitable for simulation. (b) Simulation meshes: the tree is divided into different domains (16,549 domains, shown in different colors). We compute a mechanical model for each domain separately. The total number of DOFs is 9,825. (c) Real-time simulation (using wind), using the preprocessed mechanical models. Simulation is running at 30 frames per second. Rendering is at 5 frames per second.
4.1 Plant Preprocessing

position methods [48, 96]. Our procedure starts with a triangle mesh of a plant. We make no assumptions on mesh topology or connectivity and support arbitrary “triangle soups” which may contain cracks, T-vertices or duplicated triangles. Such plant models are very common in practice. The various plant parts need not be properly connected to each other in terms of sharing vertices; for example, it is sufficient if adjacent branches simply collide with each other slightly at their common intersection (see Figure 4.2, right). Our system supports “billboards”, i.e., texture-mapped (usually simple) triangle meshes with transparency, commonly used to model leaves, fronds and smaller branches (twigs). We now explain our pre-processing pipeline. Steps performed by the user are marked as (U), whereas fully automated steps are marked as (A). Table 4.1 analyzes the time needed for each of the steps.

![Figure 4.2: Branches (F), twigs (R1) and leaves (R2) Branches can simply “sink” into each other in the input geometry (right).](image)

**Step 1: (U) Organize input mesh into domains:** Given a “polygon soup” mesh, the polygons must be grouped into domains. Each of our domains is characterized by the user as one of three types: (i) F, (ii) R1 or (iii) R2 (see Figure 4.2). Domains F are flexible, and typically incorporate meshed, non-billboard 3D geometry, e.g., the trunk, branches, or flowers.Domains of type Rx are rigid, and are in practice often instanced. They are typically used for fruits, billboard twigs, fronds, leaves, small decorative geometry or even unwanted geometry left in the model by artists (“debris”; e.g., small unconnected triangles). We use two levels R1 and R2 because in some plants, the artist intended billboard domains to be parented to other billboard domains, e.g., conifer billboard needles attached to billboard twigs. If a single level R1 was
4.1 Plant Preprocessing

employed, some domains may be parented to flexible domains that are too far away, which can cause neighboring domains to separate at runtime. A typical example of the decomposition is to assign all branches into F, twigs into R1, and leaves into R2 (Figure 4.2). The user is free to deviate from such guidelines, however. For example, with some models, leaves are modeled as detailed triangle meshes, in which case they may be considered of type F. Every triangle must be assigned into exactly one domain, and each domain is assigned one of the F, R1, R2 types. We employ a user interface similar to that in, say, Maya, where the user can select triangles or domains, and show/hide/add/subtract/delete/merge them. Domains can be selected with the mouse and then tagged as either F, R1, or R2. In some models, polygons are pre-grouped by the artists into individual logical parts, e.g., each leaf is a separate domain already in the input, in which case the domains must only be selected, and their type identified. Often, however, the parts of the same kind are grouped together, e.g., all leaves are initially in one domain, and must therefore be separated into individual leaf domains. Therefore, another operation that we support is to break an existing domain into connected components, where two triangles are considered connected if they share a vertex. We compute the connected components using the union-find datastructure [20]. When breaking domains into components, the user can choose to impose a rule that all triangles in the domain may use at most one texture image; otherwise, the domain is broken further, into one domain for each texture image. Such a rule is useful with instancing, but is rarely needed because most models already satisfy this requirement as is.

Figure 4.3: Instancing and anchors: (a) One-to-one texture map with transparency. (b) User-selected anchor points (in purple).
Step 2: (U) Instancing: Many plants consist of repeated parts. The instances are translated, rotated and sometimes scaled copies of each other, e.g., replicated leaves or flower petals. We automatically identify instances as follows. We assume that the triangle mesh of each instanceable domain is texture-mapped with a single image, with a one-to-one texture map (Figure 4.3, a). In particular, vertices cannot have different $(u,v)$ values if they appear in more than one triangle, and triangles cannot “fold over” or cross each other in the $(u,v)$ space. In our model databases, we did not encounter texture maps that would not be one-to-one. We identify instances as follows. We first inspect the texture image and the number of vertices; domains that do not match in both, are not instanced copies. For each vertex $i$ of the first domain, we then find the vertex $j = P(i)$ of the second domain whose texture coordinates are closest to those of vertex $i$, by performing a nearest neighbor search in the $(u,v)$ space. If the map $P$ is not one-to-one and onto (a permutation), or the distance to the nearest neighbor is greater than $\varepsilon = 10^{-3}$ for any vertex, we deem the domains not instanced. Finally, we check that the triangle mesh topology is same for both domains, i.e., if vertices $i, j, k$ form a triangle in the first domain, so must vertices $P(i), P(j), P(k)$ in the second domain, and vice-versa. If topology is the same, domains are deemed instanced copies of each other. At first, we attempted to use instancing where we did not seek the permutation $P$, but simply checked the $(u,v)$ distance between vertex $i$ of first domain and vertex $i$ of second domain. This approach did not work well because modeling packages sometimes arbitrarily re-order triangle mesh vertices before exporting them. Note that our definition of instancing relies on the $(u,v)$ space and therefore permits arbitrary translation, rotation and scaling of the object in world-space. All the domains that are instanced copies of one another are placed into an instance set. A single domain is chosen as a representative instance; we choose the one that appears first in the input mesh. For each instance set, we also ask the user to select an “anchor point” on the representative instance (see Figure 4.3, b). The user selects the anchor point in world coordinates by clicking with the mouse on the model, upon which we determine and store the corresponding $(u,v)$ coordinate. The anchor point will be used to determine the instance transformation, and also in Step 7 to assign a parent to each Rx domain. The anchor point should typically be selected at the end of the botanical piece, e.g., on leaf’s stem (petiole) next to the attachment point to branch (see...
4.1 Plant Preprocessing

<table>
<thead>
<tr>
<th>Example</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>10</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>dahlia</td>
<td>16</td>
<td>40</td>
<td>0.2</td>
<td>0</td>
<td>3</td>
<td>27</td>
<td>0.01</td>
<td>0.45</td>
<td>141</td>
<td>228</td>
</tr>
<tr>
<td>jasmine</td>
<td>29</td>
<td>14</td>
<td>0.6</td>
<td>20</td>
<td>3</td>
<td>63</td>
<td>0.09</td>
<td>1.13</td>
<td>124</td>
<td>255</td>
</tr>
<tr>
<td>white pine</td>
<td>36</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td>3</td>
<td>60</td>
<td>3.50</td>
<td>17</td>
<td>170</td>
<td>311</td>
</tr>
<tr>
<td>broadleaf</td>
<td>7</td>
<td>10</td>
<td>1.4</td>
<td>0</td>
<td>3</td>
<td>269</td>
<td>1.14</td>
<td>3.3</td>
<td>136</td>
<td>431</td>
</tr>
</tbody>
</table>

Table 4.1: Timings for each pre-processing step, in seconds, for four representative models (shown in Figure 4.12), including user and computer time. Four users were involved in the experiments. Timings for steps 9 and 11 are not shown to save space, but are included in totals; they are less than 0.7 sec in all examples.

Figure 4.3, b). Sometimes, billboard domains are not designed to attach to anything, but are free-floating in great numbers, e.g., clusters of needles on a conifer tree (Figure 4.3, b1). In such cases, the user should select the center of the billboard polygon. Next, for each instance from an instance set, we determine the linear transformation that best aligns the vertices of the representative instance to this instance, using shape matching [66]. Optionally, the user can force the linear transformation into a rotation, computed using polar decomposition. It is often preferable to use linear transformations, however, because the leaves are often scaled in size to add variety. The instance transformation is stored to disk, and used at runtime to properly transform the domain. Only a single mesh must be stored for each instance set.

**Step 3: (A) Computing the F domain graph:** Our system then automatically builds the domain graph for the F domains. The nodes of the graph are the F domains, and two nodes are connected if the two triangle meshes intersect in the undeformed configuration. We determine the graph edges using collision detection, accelerated by bounding volume hierarchies and spatial hash tables [56]. Because we only need to perform collision detection once (on static shapes), the domain graph construction only takes a few seconds at most, even for our most complex examples. Initially, we attempted to use collision detection on volumetric meshes (computed using voxelization) to create our hierarchy. Although such an approach avoided gaps between domains, it created many spurious graph edges, which greatly complicated spanning tree selection in Step 6.
Step 4: (U) Connecting the F domain graph: The graph computed in step 3 may not be connected. In practice, we encountered such situations in about 25% of all models. Most often, this occurs for one of two reasons: (1) There is a small gap between two domains, often visually (nearly) invisible and unimportant for rendering, or (2) the domains are “debris”: small pieces inadvertently left in the model by the artist, often invisible inside branches. We let the user connect the graph as follows. We compute the connected components (using union-find). The user can then scroll among the components, select arbitrary two domains in arbitrary components, and connect them. At any time of this process, she can recompute the graph and the connected components. In most cases, when the graph was initially not connected, the total number of components was less than 10. We encountered a few cases of “debris” with approximately 50 components, which we deleted one by one. When visualizing the connected components, we sort them base on their size. At any moment, the user can opt to simply delete the remaining components, if their size is deemed insignificant.

Step 5: (U) User selects the root domain: Plants in nature are rooted. Although most models come pre-oriented so that the Y-axis is up, this is not guaranteed, so the user has to specify the root domain. Typically, this is the stem or the trunk. We provide the domain with the largest diameter as the initial suggestion.

Step 6: (U) User-assisted resolution of loops: Although in principle the output of Step 4 should have no cycles (a tree), this is not always the case in practice. For example, an artist may have accidentally left a branch colliding with another branch, forming a cycle (see Figure 4.4, top). There are three very common causes of cycles: collisions of non-neighboring branches in the input (very common with complex trees), “Y-bifurcations” where a branch splits into two branches and all three meshes collide with each other, forming a 3-cycle, and petals on flowers (Figure 4.4, D). We found cycles to be common in commercial plant model libraries. They must be removed so that a tree hierarchy can be computed. Because the computer lacks context, it is difficult to remove cycles automatically, e.g., the small transverse branch in Figure 4.4 (magnified in A), may as well be connected to either of the main branches. A human, however, can look at the branch and recognize its intended direction. Therefore, we designed an
4.1 Plant Preprocessing

Figure 4.4: Loop resolution: **A**: a 7-cycle. **B**: an example of a 3-cycle where a computer may make a mistake and that requires user intervention. Because both smaller branches collide with the bigger branch, computer initially suggests an incorrect loop-breaking graph edge (center), whereas the correct edge is shown on the right. **C**: the minimum spanning tree selection algorithm. (1): Initial minimum spanning tree (green) and redundant edges (red). The edge to be resolved next is marked by “add”. (2): The loop. User decides to cut the edge marked with “X”. (3): The minimum spanning tree after deleting the edge. (4): The hierarchy after the remaining two redundant edges were processed. **D**: Examples of loops in input geometry: “Y”-bifurcation and flower petals.
algorithm for user-assisted removal of loops. Our algorithm applies generally to any problem where a quality minimum spanning tree must be selected out of the many minimum spanning trees of an undirected graph with cycles. We first compute an initial candidate minimum spanning tree, using a breadth-first traversal starting from the root domain. If the graph has no cycles, we are done; otherwise, the edges which are not in the minimum spanning tree are added to the set of redundant edges, prioritized by their breadth-first order (Figure 4.4, C1). We then let the user resolve these edges, one by one. For each redundant edge, we add it to the minimum spanning tree, and therefore exactly one cycle appears in the resulting subgraph $S$ (Figure 4.4, C2). The cycle is computed by traversing the minimum spanning tree from each of the two vertices of the redundant edge towards the root of the tree, until a common ancestor is detected, thereby detecting a cycle. The cycle is then visualized to the user, by coloring the cycle domains in a golden color (as in Figure 4.4, top). The user is then asked to break the cycle by removing exactly one edge. The user does so by scrolling through the cycle (as in Figure 4.4, top; the two node domains that are the endpoints of the edge are shown in red and pink), and selecting the edge to be removed. After the edge removal, the working graph $S$ is a tree again, and the redundant edge set has shrunk by one (Figure 4.4, C3). This process is repeated until the redundant set becomes empty, i.e., the working graph $S$ is a tree and all the redundant edges were resolved. We then orient $S$ to form a tree hierarchy, starting from the root and proceeding to the leaves (Figure 4.4, C4). Because redundant edges are prioritized by their breadth-first traversal order, the user resolves cycles closer to the root first. In our tree database, virtually all plants initially had a non-tree domain graph, and required user intervention. Most loops, however, occur due to “Y” bifurcations, and can be resolved very quickly. For small / moderate examples, these were often the only loops. The vast majority of redundant edge sets that we have encountered in our examples had less than 100 edges. A few large examples, such as large trees, had approximately 500 redundant edges. Even for the most complex trees, the redundant edge set removal was manageable and was completed within minutes of user time. We were able to significantly shorten the user time by implementing an auto-focus feature where the camera automatically focuses on each loop as the user is scrolling through the loops.
Step 7: (A) Add domains \textbf{Rx} to the domain tree: We now assign parents to domains \textbf{Rx}. Domains \textbf{R1} are always parented to \textbf{F} domains, whereas domains \textbf{R2} can be parented by \textbf{R1} or \textbf{F}. We use anchor points to determine the parent. The anchor position is computed using the inverse of the texture map, by finding the world-coordinate location on the domain mesh whose texture coordinates are \((u, v)\), where \((u, v)\) are the texture coordinates selected on the representative instance by the user in Step 2. Such anchor position computation was very robust in practice. We then perform a nearest neighbor search, seeking the nearest triangle in all domains in \textbf{F} (and also in \textbf{R1} for \textbf{R2}) to the anchor position; the closest domain becomes the parent. Such an assignment is robust to cracks and can accommodate (intentionally) “floating” domains that are common with billboarding. For example, leaves on trees in many models are simply accumulated in close proximity to give a space-filling perception, and are far from any branch. The nearest neighbor query is accelerated using a bounding volume hierarchy (we use axis-aligned bounding boxes), and only takes a few seconds, even for complex models (Table 4.1).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.5}
\caption{Voxel simulation meshes. Left: root domain. Middle, Right: two representative domains. Note that the meshes for the different domains are completely independent and do not need to meet in a common interface. Adjacent petals can thus be meshed independently without any difficulty, even though they collide in the input triangle mesh (middle, right). Fixed vertices are shown in red, and need not be vertices of the parent volumetric mesh; some are even outside the parent mesh.}
\end{figure}

Step 8: (A) Build simulation meshes: We build the volumetric mesh for each domain by voxelizing the domain’s triangle mesh [39] (Figure 4.5). We chose this approach because it is completely automatic and supports arbitrary (potentially ill-formed) input “polygon soup”
4.1 Plant Preprocessing

geometry. Alternatively, one could employ automatic tetrahedral meshes such as [52]. The
user specifies the maximum voxelization resolution (it defaults to 100 in our system) for the
expanded bounding box of the input triangle mesh (we use expansion factor of 1.2 ×). The
voxelization resolution of each domain is then set automatically, by assigning to each domain
a resolution proportional to the longest edge of its bounding box. Using such an adaptive reso-
lution, the meshes of different domains grade approximately uniformly across the entire plant
(see Figure 4.5). We note that the voxel meshes for the individual domains are separate meshes;
they need not topologically connect to meshes of other domains using any well-defined inter-
face. Such meshing flexibility is possible in domain decomposition simulators that do not need
the connectivity requirement, such as [9, 96]. Our “bottom-up” mesh generation approach has
the advantage that an entire simulation mesh never needs to be constructed. It is also very
modular. Input triangle mesh collisions of neighboring pieces (e.g., petals on a flower) are not
a concern, because each piece is simply meshed separately (see Figure 4.5). If a new piece
needs to be added, it can be pre-processed in isolation and added to the domain graph. In [9],
domain decompositions were created by forming a global tetrahedral mesh and then manually
subdividing it into domains. Although such a ”top-down” approach results in well-defined in-
terfaces, it requires careful manual work to produce quality interfaces. For example, the choice
of whether to place a tet into the left or the right domain at the interface can affect the bend-
ability of short branches. Also, top-down approaches require special handling in the presence
of collision in the input triangle mesh; otherwise, the colliding branches or petals will simply
be welded in the global volumetric mesh.

Step 9: (A) Assign fixed vertices: Fixed vertices serve as connection points of a domain
to its parent. The root domain is rooted to the ground, either by fixing all vertices that are
below a user-provided height, or by manually selecting its fixed vertices. Fixed vertices for the
remaining domains are set automatically as follows. For each parent-child pair in the hierarchy,
we constrain the child vertices that are located close to the parent. Specifically, we detect child
voxels that intersect parent voxels, and constrain all the vertices in those child voxels. Because
we always constrain all eight vertices of a voxel, we avoid degenerate planar sets of fixed
vertices. The intersection test is accelerated using a bounding volume hierarchy. Such an assignment handles both the case where the child voxels are smaller than parent voxels and may be completely inside a parent voxel, and the case where child voxels are large and may subsume parent voxels. It can happen that all the vertices of a domain are deemed fixed; e.g., with small, in-grown, branches close to a bigger branch; in such cases, we declare the domain to be rigid. When no fixed vertices are detected, the domain is also made rigid. Such cases are rare in our data, but sometimes occur with domains not properly connected to the rest of the plant, e.g., small branches added for decoration in the tree crown. Note that for our domain decomposition approach, it is not necessary to establish a well-defined interface between the two domains; the child fixed vertices may even be outside of the parent mesh (Figure 4.5, middle, right). Also note that in botanical systems, the interfaces between branches and/or leaves are often small in surface area, and their bending can be neglected. Namely, the state-of-the-art papers in plant simulation go even further and assume that all the parts (the branches) are completely decoupled [32, 34]. In our work, the domains are coupled, with the assumption that the interface deformation is small.

**Step 10: (A) Compute low-dimensional simulation basis and pre-process reduced dynamics:** Given the fixed vertices, we compute linear modes, their large-deformation correction (modal derivatives [8]), and the simulation basis. By default, we use the first 10 linear modes, and a 20-dimensional basis. We use a default (and tunable), spatially uniform, mass density $\rho = 1000 \text{kg/m}^3$, Young’s modulus $E = 10^6 \text{N/m}^2$ and Poisson’s ratio of $\nu = 0.45$, corresponding to a fairly incompressible material. We then precompute a geometrically nonlinear FEM reduced model for each domain [8]. Each of these models is a compact, low-dimensional representation of the FEM dynamics of each domain. It supports large deformations, and can be timestepped rapidly, in microseconds. The domains are coupled to each other as described in Section 3.2. The entire process is automatic. In order to save space and increase speed, we use an adaptive number of modes per domain, by pre-processing smaller bases for smaller domains. The user sets the maximum size of a domain simulation basis $r_{\text{max}}$ (we use $r_{\text{max}} = 20$). For a domain with $e$ elements, the number of modes is then computed as
4.2 Forests

\[ r = \max(5, \lfloor r_{\text{max}} \log(e) / \log(e_{\text{max}}) \rfloor) \], where \( e_{\text{max}} \) is the maximum number of elements in a domain (see also [9]).

**Step 11: (A) Tune Young's modulus:** After the reduced models are pre-processed, they have correct stiffness levels relative to each other. However, the global stiffness scale is arbitrary; one can globally scale \( \rho \) and \( E \) with an arbitrary constant without affecting the basis, and the precomputed reduced polynomials only scale by a constant [8]. Because we are using a FEM method, it would be possible to set the material properties for wood, stems, leaves, etc. Although progress on measuring elastic material properties has been made [13], the parameters for plants are typically not available. Therefore, we opt to assign default material values, pre-process the reduced models, then exploit the fact that natural vibration frequencies scale linearly under a linear global scale of Young's modulus. We scale Young's modulus so that the lowest natural vibration frequency of the root domain becomes 1 Hz, i.e., \( E' = E / f_0^2 \), where \( E \) is the default Young's modulus, and \( f_0 \) is the lowest natural frequency of vibration of the root domain, determined using a sparse eigenvalue solver [54]. The user can then further scale the spectrum as needed, to make the plant stiffer or softer. Tuning is very fast at runtime; it takes less than one second even for complex models.

4.2 Forests

It is easy to expand our simulator to simulate a forest (Figure 4.7). The key idea is to form a virtual root to accommodate the forest. It becomes the root of our hierarchy (Figure 4.6). This virtual root is simply a box, large enough so all the botanical objects can be rooted. It is rigid and fixed in the world coordinate frame, thus, no simulation is needed. All the botanical objects on the top are pre-processed separately. Replicated copies of the same plant form an instancing group and only the representative of this group needs to be pre-processed. We continue to spell-check how to handle transformation of an instanced copy.

**Scaling** If the mesh is scaled simultaneously in \( x, y, z \) by factor \( \alpha \), the reduced deformation basis must be scaled by \( \alpha^{-3/2} \). Note that, in this case, the tangent stiffness matrix \( K \) is scaled
4.3 Interactive Plant Design

Figure 4.6: Hierarchy of a forest: (a) (b) Hierarchies for two plants. (c) A virtual root (shown in red) is created to accommodate the forest. It becomes the root of the hierarchy. Replicated plants (plant 1 shown in yellow) form an instancing group.

by $\alpha$, the mass matrix is scaled by $\alpha^3$, and the frequency is scaled by $1/\alpha$. These scalings happen automatically in our system. User may have to tune the stiffness properties in the simulator to meet the application needs.

Rotation The reduced deformation basis must be rotated accordingly. Often with plants, however, rotation is not needed.

Translation Translate the plant to the correct position. No special operation is needed for the pre-processed data.

4.3 Interactive Plant Design

Because our method is fast and modular, it is possible to use it as an interactive shape editor for plants. The user can easily delete or add new parts at runtime, either by duplicating existing plant parts, or importing parts from a pre-processed library of plant parts. The editing process is local and interactive. The mass, stiffness, position and orientation of each branch, twig or leaf, as well as a linear scale (geometric size), can all be adjusted interactively at runtime, with immediate physically based simulation feedback to the user, without any additional pre-
Figure 4.7: Fir forest in randomized wind field: It consists of 20 firs, each of which has 81,228 triangles, 9,077 domains, 4,858 DOFs. Entire scene is simulated at 2 frames per second, including $u = Uq$ construction of vertex deformation. Rendering is at 1 frames per second.

processing. It is possible to randomize these choices, creating an arbitrary number of variations of the same plant. In our interactive editor, we can select an arbitrary domain, and linearly scale the stiffness (Young’s modulus) of all of its elements by any factor $\alpha > 0$. It can be shown that under such a scaling, the reduction basis does not change [8], whereas the reduced forces scale by $\alpha$; therefore, the scaling is instant, and there is no need to maintain an explicit volumetric mesh or its element material properties. Because stiffness is proportional to the square of the lowest natural vibration frequency, such scaling makes it possible to cause a branch to oscillate faster or slower. For cinematic effect (to show more secondary motion), we sometimes found it useful to make smaller branches oscillate more slowly than dictated directly by physics. We achieve this by scaling every domain with a factor $\alpha = \beta^d$, where $\beta > 0$ is a constant and $d$ is the depth of domain in the hierarchy. Because our method is fast, the user can tune $\beta$ interactively with immediate feedback; typically, values close to $\beta = 0.1$ were producing good results.

In addition, in our system the user can also manipulate the plant using inverse kinematics-like
Figure 4.8: Editing tree shapes using point constraints: The tree shape was adjusted to avoid collision with the house. The tree manipulated points are shown in red. Bottom row shows the tree without the leaves. 45 branches, 125 twig billboards, 1154 leaf billboards, 8 msec of simulation for one graphical frame.
handles. Using such a tool, it is possible to constrain and drag plant vertices, while the rest of the plant automatically re-adjusts using physics to a good-looking, minimal strain energy configuration. For example, a tree can be made to lean in a certain direction or avoid external objects (Figure 4.8). The user can also use it to resolve any unwanted collisions in the rest configuration, or simply re-adjust the plant shape to increase scene geometric variety. In our IK tool, the user can select or deselect an arbitrary number of vertices (IK handles). As the user drags the mouse, a three-dimensional force is applied to the active IK vertex, whereas the remaining IK vertices are kept fixed to their positions using linear springs. The mouse force is applied in the “screen plane”, i.e., plane orthogonal to the view direction and cutting through the manipulated vertex. The force magnitude is proportional to the number of pixels traveled by the mouse since the beginning of the drag. We use our standard real-time dynamics solver for such IK manipulation; no special code is required. Because the model normally undergoes dynamic motion, we must employ some mechanism to quickly stabilize the motion to a limit equilibrium configuration. One approach is to perform a Newton-Raphson iteration to seek the model equilibrium under the IK linear spring forces (a static solver [7, 63]). However, such a solver often suffers from a high linear system condition number, which has lead to instabilities in our experiments. We found much better results by simply using a high level of stiffness-proportional Rayleigh damping, in a dynamic simulation. Although the model does not reach the equilibrium configuration instantly, with proper gains and damping the convergence is rapid and stable. A single stiffness gain for all the linear springs was sufficient in our examples. Because we already injected sufficient damping into our simulator, linear springs did not need any additional damping.

We note that we first attempted to perform inverse kinematics by enforcing exact user control over the handle positions. This was performed by solving, at every timestep, an optimization problem that minimized the total strain energy of the plant subject to the exact IK constraints. This approach did not work very well in practice. Because the handles must be specified in the three-dimensional space which can be difficult to visualize on a 2D screen, it was easy for the user to accidentally command unreasonable handle positions, inconsistent with plant dimensions. As the solver was trying to meet the constraint exactly, the plant would stretch
unnaturally, which led to vibrations and instabilities. Instead, when positions are enforced via
springs, the solver has the ability to selectively “yield” on each constraint as needed. This tends
to produce much smoother, natural-looking shapes. We were able to tune the IK stiffness gains
to reach both good output shapes and minimal deviation of the constrained vertices from their
prescribed positions (Figure 4.8). After the user is satisfied with the shape, she can save it to
disk, and re-process the reduced models with respect to the new rest shape. For small/moderate
edits where the old reduced basis is still sufficient, a re-process is not necessary; in this case,
one can simply compute the reduced forces for the new shape for each domain, and then offset
reduced forces so that the new shape is the rest configuration.

4.4 Fracture

![Image of a plant]

**Figure 4.9: Real-time Fracture:** The tea bush (Camellia Sinensis) leaves are shaken from the
bush by the user-applied force in real time. Left: Before fracture, with applied user force indi-
cated. Middle: during fracture. Right: after fracture. Note that the leaves close to the user force
location fractured in greater numbers than elsewhere on the model because they underwent higher
accelerations. 6 msec of simulation per graphical frame.

In nature, plants often fracture at the joints between its parts. For example, leaves or fruits
detach from branches (see Figure 4.9), or branches crack away from the main stem. Such
fracture with pre-specified patterns is useful in interactive applications because it is control-
lable and artist-directable [74]. We support such fracture by monitoring the (reduced) interface
forces between adjacent domains. If the $L_2$-norm squared of reduced interface force vector ex-
ceeds a user-adjustable threshold, we fracture the entire subtree from the main structure. The subtree (in many applications a single domain) then undergoes a ballistic trajectory under gravity, e.g., peach tree fruits land on the ground (Figure 1.2). Such fracture is computationally extremely inexpensive as the $L_2$-norm test can be performed in nanoseconds. Because each part has its own rendering mesh independent of all the other parts, the objects are automatically free of holes after separation; there is no need for any re-meshing. The fracture events could also be scripted / keyframed, e.g., to simulate trees undergoing an explosion. Our domains can only fracture at the interfaces to other domains. For more detailed fracture, domains can be divided during the pre-process, e.g., the trunk can be pre-cut into two pieces.

4.5 Motion of Leaves

![Figure 4.10: Leaf Motion: A leaf is swinging in a randomized wind force field.](image)

Leaves are usually (but not always) considered as rigid domains in our method, especially in the tree simulations (Figure 1.1, Figure 1.2). It is possible to make leaves flexible by treating them as every other domain. (Figure 4.11, Figure 3.4) A rigid leaf has no reduced degrees of freedom so they can do only rigid transformation along with the branch it attaches to. This makes it possible to produce very plausible simulation results. However, to make the animation even more realistic, we can add motion to the leaves not by actually simulating them but by kinematically deforming them before the rendering stage. Note that this is an alternative approach to treating leaves as flexible domains. (Figure 4.10) To do so, we first compute a 3-dimensional linear reduced basis $U \in \mathbb{R}^{3n \times 3}$ for a leaf. Here, $n$ is the number of vertices of the leaf mesh. Since the leaves are instanced, the reduced basis is only computed for every representative leaf. Then we introduce a space-time randomized wind field whose strength is adjustable by the user. At each timestep, we query the wind field with the current time instant.
and the position of a sample point (the anchor point of the leaf), yielding a 3-vector $f$ which is the wind force vector. We assemble the displacement of the leaf using:

$$u(f) = U f$$

(4.1)

Sometimes the deformation of a leaf could be large, causing visible artifact since we are using linear modes. We fix this problem by introducing the large deformation correction using modal derivatives [8]. Equation 4.1 essentially gives a kinematic term that corrects linearized deformations to large deformations.

$$u(f) = U f + \sum_{i=1}^{3} \sum_{j=1}^{3} \Phi_{ij} f_i f_j$$

(4.2)

where $\Phi_{ij}$ is the modal derivative and $\Phi_{ij} = \Phi_{ji}$

### 4.6 Results

We pre-processed over 100 plants; we provide a selected subset of 32 pre-processed and simulated models in Figure 4.12. It can be seen that the performance is interactive even for very complex plants. Plants of small to moderate complexity are fast enough to be easily used in real-time systems such as computer games. Inverse kinematics and fracture were demonstrated in Figures 4.8 and 4.9, respectively. Plant motion resulting from user (mouse) forces, followed by free vibration, is demonstrated in Figure 4.13.

Several of our plants are animated deforming in the wind. We refer the reader to Chapter 6 for more details about the wind. Our wind consists of two components: a wind with a (tunable) constant direction and magnitude, and a randomized wind, implemented as a 4D space-time Perlin noise [77], with standard parameters: number of frequencies and persistence (how quickly high frequencies decay). All the wind parameters are easily adjustable at runtime without any precomputation. Our method can model the entire spectrum of winds from gentle breezes, moderate winds (Figure 1.2) to hurricanes (Figure 4.14). Stochastic noise is
4.6 Results

Figure 4.12: Representative subset of simulated models: including flowers, bushes, broadleaf and conifer trees. Input meshes are from Xfrog, SpeedTree and 3dmolier (Turbosquid) model libraries. Models were deformed either by pulling on vertices or using a randomized wind. Each plant reports #triangles, #domains, #flexible domains, total # of DOFs, the simulation frame rate and memory. The simulation frame rate includes all computation to produce the next graphical frame, except the rendering itself. The pre-processing times range from a minute for simple models to 20 minutes for the most complex trees. Intel Xeon 2x8 cores 2.9 GHz CPU, 32GB RAM. GeForce GTX 680, 2GB RAM. Models from Table 4.1 are shown in (1-indexed) (rows, columns)
often used to animate trees by directly driving the deformations [73]. We employ Perlin noise to create spatially varying and controllably turbulent wind forces, causing large deformations and secondary motion of branches due to the motion of parent branches. We use Perlin noise directly, but other noise generators such as the $1/f^\beta$ noise [73] or even Navier-Stokes equations [1] could be used instead. In order to apply the wind, we must sample it at properly selected locations on the model. One could evaluate the wind at every plant vertex, but doing so would require many wind evaluations and subspace force projections. Another alternative that we considered but did not pursue due to prohibitive cost was to sample the wind on a regular grid, and then seek the nearest vertex to each grid point. Instead, we select a representative set of volumetric mesh vertices on each branch domain, and then sample the wind at those locations (see Figure 4.14, top-left). Each wind sample is scaled with the volume of each branch.

Our system also supports plants loaded by gravity (Figure 4.15). Because gravity acts in a constant direction, it needs to be rotated into the frame of reference of each domain, $f_i^{ext} = U_i^T R_i f$, where $U_i$ and $R_i$ are the matrix specifying the low-dimensional space, and world-
4.6 Results

Figure 4.14: Palm tree (Cocos Nucifera) in strong randomized wind: Top-Left: the locations (in red) where the wind is sampled. Other images: selected animation frames. Simulation time: 4 msec per graphical frame. In this demo, palm leaves are not simulated, but are skinned to the palm branches. Each leaf is skinned entirely to one branch, with each vertex copying the displacement (in local branch frame) of the closest branch vertex in the undeformed configuration. Such skinning can greatly enrich the plant visual appearance at a minimal computational cost.

Figure 4.15: Gravity space: Left: undeformed. Right: under gravity.
coordinate rotation of domain $i$, respectively, and $f = [0, -g, 0, 0, -g, 0, \ldots 0, -g, 0]^T$ is the gravity vector. Because $f$ is constant, $f_i^{\text{ext}}$ can be efficiently precomputed using the sandwich transform [48], by evaluating $f_i^{\text{ext}}$ for $R_i = e_k e_k^T$, where $e_k$ is the $k$-th standard basis vector in $\mathbb{R}^3$, for all $k, \ell = 1, 2, 3$. At runtime, for each non-rigid domain, one then merely has to multiply a constant pre-computed $r_i \times 9$ matrix ($r_i = \# \text{reduced DOFs of domain } i$) with the 9-vector of the entries of $R_i$, imposing a negligible overhead. It would be possible to pre-load plants so that the input configuration is also the rest configuration under gravity [97].

Comparison to full simulation: In practice, vegetation is often simulated by building a simulation mesh for the entire model, then employing a deformable object simulator to timestep the plant dynamics forward in time [4, 59]. In Figure 4.16, we compare our method to a geometrically nonlinear full simulation [16]. For this experiment, we generated a global tetrahedral mesh for the shefflera plant (Shefflera Actinophylla), and then manually subdivided it into domains. This process took 8 hours of work, whereas our pre-processing pipeline takes less than 10 minutes. In order to make the motion more natural, we made the stem 2x stiffer than the rest of the mesh. We then simulated the shefflera under an identical force load, material and simulation properties. In this example, our method (including time for $u = Uq$ displacement computation) is 23x faster than the full simulation. Visually, the two motions differ slightly in frequency, but appear qualitatively similar. If a close frequency match is desired, it is possible to automatically scale the frequency spectrum of each domain so that the lowest frequency matches some externally prescribed frequency, such as frequency from full simulation, or real measurements of plants [43]. We illustrate this

Figure 4.11: Visualization of domains in the oak tree: In this example, all leaves are flexible domains, each of which has 8 reduced DOFs.
concept with a 1D harmonic oscillator, $m\ddot{x} + d\dot{x} + kx = f$, whose natural angular frequency (without damping) is $\omega = \sqrt{k/m}$. Suppose a different frequency $\omega' \neq \omega$ is desired. The modified mass and stiffness must therefore satisfy $\alpha^2 km' = k'm$, where $\alpha = \omega'/\omega$. If we impose an additional condition that the new oscillator attains the same maximum amplitude after a fixed initial impulse, we obtain $k' = \alpha k$ and $m' = m/\alpha$. With plants, we perform the same scaling, to match the lowest vibration frequency, separately for each domain. We obtain the desired frequency for each branch by creating a volumetric mesh for the entire subtree rooted at that branch, and computing (unreduced) mass and stiffness matrices $M$ and $K$. The desired angular velocity is then the square root of the lowest eigenvalue of $Mx = \lambda Kx$, which we find using a sparse eigensolver [54]. We scale the reduced mass matrix by $1/\alpha$ and internal elastic forces by $\alpha$. The result is shown in Figure 4.16. We note that, instead of matching the lowest frequencies to full FEM simulation, one could instead match them to real-world plant observations.

Figure 4.16: Comparison to full simulation. 91 flexible domains, 1183 modes, 11,396 triangles. All three simulations visually look similar. The unscaled reduced simulation has a higher natural frequency, for two reasons: (1) reduced systems lack degrees of freedom and are typically somewhat stiffer than unreduced systems with same parameters, and (2) interface lumping [Barbič and Zhao 2011] increases frequency, much like a pendulum with a shorter length oscillates faster. After scaling, the frequency and amplitude match full simulation closely.
**Rendering:** We render the plants interactively using OpenGL. Billboards are in practice often partially transparent, with alpha values continuously ranging from zero to one. Therefore, one-pass alpha-testing results in noticeable aliasing, for example, at the leaf edges. We use two-pass rendering where we first render all the geometry with alpha-testing enabled, i.e., fragments where the alpha value is strictly less than 1.0 are discarded. We then make the depth buffer read-only, disable alpha-testing, enable alpha blending, and re-render all domains that use transparent texture maps. Note that in many models transparent textures represent a large fraction of the geometry, e.g., leaves, conifer needles, twigs. Our simulation is substantially faster than rendering; more optimized rendering pipelines could be designed [90]. Before our models can be rendered, the displacements (in local domain frame of reference) of all the mesh vertices must be computed via the modal equation $u_i = U_i q_i$ [40] and then their world coordinate positions constructed using the domain’s current position vector and rotation matrix. All our reported simulation timings include the time necessary to compute $u_i$ and world coordinate triangle mesh vertex positions. We compute $u_i$ on the CPU, but GPU implementations [40] would be readily possible. We note that matrix $U_i$ here contains the modes that were already interpolated (during pre-process) from the volumetric mesh to the plant triangular geometry, using barycentric interpolation. We found that such a strategy is usually faster than interpolating volumetric mesh displacements to the triangle mesh at runtime; the tradeoff depends on how finely the triangle meshes are tessellated. Offline renderings were performed using the Yafaray ray tracer.
Chapter 5

Procedural Materials and Asynchronous Parallel Timestepping

Given a large plant, how to determine material properties of each branch, twig or leaf? In real trees, branches undergo lignification, which greatly affects mechanical behavior of trees. In the first part of the chapter, we present a novel procedural method to automatically set simulation parameters for complex plants. These parameters include material stiffness, damping coefficients, and timestep. Such tuning is quite challenging due to the complexity of realistic tree models. In practice, it is often too complicated to do it manually in order to generate vivid and plausible plant motion. The simulation results, achieved by applying automatically tuned parameters to our reduced multi-domain simulator, demonstrate the quality and efficiency of our tuning algorithm. Note that this algorithm could be applied not only to our architecture, but also to any other FEM simulation.

In the second part of this chapter, we introduce an asynchronous, parallel timestepping algorithm to improve our reduced multi-domain dynamics simulator presented in previous chapter. Since the natural frequency varies in domains, we integrate each domain at its own pace, as opposed to using a fixed timestep for the entire object. This avoids the limitation of a synchronous integrator, in which a small, fixed time step needs to be carefully chosen in order to avoid numerical instabilities due to the high stiffness variations encountered at the domains. The total running time to produce animations of a given length (same total physical time) was
therefore shorter than the synchronous method. By exploiting the fact that no loop exists in
the tree topology, the simulation is further accelerated by timestepping all domains in parallel.

5.1 Procedural Stiffness

We now present our approach to set the material property of each domain so that the result
motion looks real, and natural. In nature, objects vibrate at different frequencies when their
size is changed. Longer and thicker objects tend to vibrate slower. If we keep the material
properties constant and uniform everywhere on the plant, smaller branches would then have
much higher natural frequencies than the bigger ones. This will cause, for example, such
small branches to be severely overdamped and behave mostly rigidly. However, this does
not match the phenomenon observed in the real world where smaller branches usually tend
to behave more lively. For example, twigs are swinging heavily in the wind while the trunk
may not shake too much. The reason is that branches have different material stiffness. Small
branches are usually younger and therefore feels softer than the big ones which have been
largely lignified. Therefore small branches in nature have lower vibration rates.

To create realistic results, one has to set the stiffness of all branches correctly. A possible way
to do this is to adjust the stiffness of each branch manually. The more dense the object is, the
slower it will vibrate, and the lower its frequency will be. Such a technique, however, requires
a lot of manual work and is by no means intuitive to the user. In this section we design a
model called “aging mode” which automatically sets the stiffness of each domain based on the
geometry of a domain and the topological structure of the entire plant.
We observe that the age of a branch often relates to its length. In most cases, the older the
branch is, the longer it will be. Our idea is to first estimate the age based on the geometry and
then set the stiffness according to the age. To do so, we design the following scalar function to
model the stiffness as a function of age.

\[ E(a) = c_1 \cdot a^D \]  

(5.1)
where $a$ is the age of the branch and $c_1$ ($c_1 \geq 0$) and $D$ ($D \geq 0$) are two real coefficients. The exponent $D$ is input by the user from UI. It controls how aggressive the aging mode is. Higher values will make small branches younger and therefore more lively. Stiffness is proportional to the square of the lowest natural vibration frequency. Therefore we model the final branch frequency $\nu$ as

$$\nu = \nu_{\text{geom}} \cdot \sqrt{E(a)}$$

(5.2)

where $\nu_{\text{geom}}$ is the initial frequency under default, uniform materials, precomputed in the pre-processor using linear model analysis. By inserting 5.1 into 5.2, we obtain

$$\nu = c \cdot \nu_{\text{geom}} \cdot a^{D/2},$$

(5.3)

where $c = \sqrt{c_1}$. In the following, we explain how to model the age of the branch. Let $\ell$ denote the length of the branch. We postulate that the age is linearly proportional to length $\ell$. Since $\nu_{\text{geom}}$ is inversely proportional to $\ell$, we have:

$$a = \text{const} \cdot \ell = \frac{c_2}{\nu_{\text{geom}}},$$

(5.4)

where $c_2 \geq 0$ is a constant.

Sometimes a branch is shorter than its descendents. That means it has a higher initial frequency than some branches in its sub-tree. This often happens, for example, in “shrub models”. In this case, Equation 5.4 implies that this branch is “younger” than some of its descendents which is a contradiction. To fix this problem, we enforce the condition that a branch should be always order than all of its descendent. We extend 5.4 to

$$a = \max\{\frac{c_2}{\nu_{\text{geom}}}, a_{\text{oldest descendent}}\}$$

(5.5)

where $a_{\text{oldest descendent}}$ is the age of the oldest descendent of that branch.

Our material adjusting procedure consists of two steps. In the first step, we traverse hierarchy from leaf nodes to the root (bottom-up) and compute the age property for every domain based on Equation 5.5. In the second step, we set the stiffness by assigning vibration frequency of
5.2 Procedural Timestep

Each branch based on Equation 5.3. We set the coefficient $c$ constant within each hierarchy level. User sets $c$ for each level on the UI. This is based on the assumption that domains with the same depth belong to the same generation and have roughly the same stiffness property. We assign decreasing $c$ values to increasingly deep levels from root to leaves. A good way to tune a tree is to set the first few levels stiff, i.e., 3.0, 5.0 or even 10.0. Then, make deeper levels softer, i.e., 1.0. For the aging exponent $D$, a value of 2.0 is a reasonable choice; it causes all branches to have equal frequency. A value of 0.0 is equivalent to a constant stiffness for all the branches, i.e. purely geometric models. From the simulation results, for $D = 2$ (or similar) we clearly observed much richer and more lively secondary motion in smaller branches due to the motion of parent branches. And the main body of the plant, like the trunk, looks very stiff preventing it from rubber-like deformations even under large external forces. The overall motion looks much more natural and real then when using a constant and uniform material property everywhere in the tree.

5.2 Procedural Timestep

Here our goal is to compute a separate timestep for each domain so the simulation could be advanced in each local frame without any numerical instabilities. To achieve this, we first solve a mass-scaled generalized eigen problem [8] to compute the natural frequencies of the domain. The generalized eigenproblem has the form, $K\psi = \lambda M\psi$ where the $K$ is the reduced tangent stiffness matrix, evaluated at the origin. Here, $M$ is the reduced mass matrix (Equation 3.17). Note that both $K$ and $M$ are $r \times r$ symmetric, positive-definite matrices, where $r$ is the number of degrees of freedom of the reduced space. If the $r$ eigenvalues are sorted in ascending order, the smallest eigenvalue $\lambda_0 > 0$ corresponds to the lowest natural frequency, also known as the main frequency, of the object. We pick this $\lambda_0$ to compute the corresponding oscillatory period as $\lambda = \omega^2$ where $\omega$ denotes the natural frequency in radians. The oscillatory period is then given by

$$T = \frac{2\pi}{\sqrt{\lambda_0}}.$$  (5.6)
At this point, our system requires the user to provide a positive integer, called *quality control coefficient*, denoted as $Q$. The simulation timestep is set as $\frac{\Delta t}{2^n}$, where $n$ satisfies that

$$\frac{\Delta t}{2^n} < \frac{T}{Q} < \frac{\Delta t}{2^{n-1}},$$

and $\Delta t$ is the graphical time step. We usually set $\Delta t = 1/30 \text{ sec}$ or $\Delta t = 1/24 \text{ sec}$, which conforms to the standard frame rate in modern TV and digital cinema. The user is able to adjust this value on the UI. We use power of 2 ($2^N$) so that we can perform parallel timestepping efficiently (next section).

The quality control coefficient $Q$ is the only parameter that needs to be set by the user. By default, we set $Q = 100$ which is, in our experience, a very good value for most simulations. User can easily adjust it to favor either the simulation speed (lower $Q$) or accuracy (higher $Q$). Our simulator is quite stable even for small values of $Q$ where a lot of artificial damping is introduced into the system which helps to quickly dissipate the energy. We only observed explosions in some very rare and extreme cases. By increasing $Q$, these numerical instabilities were soon fixed and disappeared.

### 5.3 Procedural Damping

We adopt proportional (Rayleigh) damping in our system which has the form $C = \alpha M + \beta K$ where $M$ is the reduced mass matrix, $K$ is the reduced stiffness matrix, $\alpha$ and $\beta$ are damping coefficients. Since domains have different stiffness and are simulated separately, each of them should be assigned different damping coefficients $\alpha$ and $\beta$ in order to achieve better simulation results. To do so, we have designed a procedural method to tune these damping coefficients automatically. We first employ the *modal damping factor* [40]:

$$\xi = \frac{1}{2} \left( \frac{\alpha}{\omega} + \beta \omega \right)$$

(5.8)
where $\omega$ is the undamped, lowest natural frequency of vibration (in radians) computed using:

$$\omega = 2\pi \nu,$$  \hfill (5.9)

where $\nu$ is the lowest natural frequency of the domain. We model $\xi$ as a function of the frequency of each domain. This is done by modeling $\xi$ as a constant function of $\nu$ for $\nu \leq \nu_{\text{low}}$ and $\nu \geq \nu_{\text{high}}$ and a straight line in between. Formally, this piecewise function is as follows:

$$\xi(\nu) = \begin{cases} 
\nu_{\text{low}} & : \nu \leq \nu_{\text{low}} \\
\frac{\xi_{\text{high}} - \xi_{\text{low}}}{\nu_{\text{high}} - \nu_{\text{low}}} (\nu - \nu_{\text{low}}) + \xi_{\text{low}} & : \nu_{\text{low}} < \nu < \nu_{\text{high}} \\
\nu_{\text{high}} & : \nu \geq \nu_{\text{high}}
\end{cases}$$  \hfill (5.10)

where $\xi_{\text{high}}, \xi_{\text{low}}, \nu_{\text{high}},$ and $\nu_{\text{low}}$ are set by the user on the UI. Given the frequency of a domain, $\xi$ is then computed using equation 5.10. However, knowing $\xi$ does not uniquely define $\alpha$ and $\beta$ since we have two unknowns but only one equation 5.8 so far.

At this point we introduce another dimensionless factor $\gamma$ which models the liveliness of the high frequency components. When the natural vibration frequency becomes twice as large as the lowest one, we let the modal damping factor $\xi$ be scaled by $1/\gamma$, that is,

$$\frac{1}{2} \left( \frac{\alpha}{2\omega} + 2\beta \omega \right) = \frac{\xi}{\gamma}.$$  \hfill (5.11)

The damping coefficients thus can be computed using Equations 5.11 and 5.8:

$$\alpha = \frac{4\xi \omega}{3} \left( 2 - \frac{1}{r} \right),$$  \hfill (5.12)

$$\beta = \frac{2\xi}{3\omega} \left( \frac{2}{r} - 1 \right).$$  \hfill (5.13)

Generally we wish the damping forces to penalize higher frequency motions more, which means $\xi$ should be a monotonically increasing function for frequencies of each domain. It can be derived from Equation 5.8 that $\xi$ reaches a global minimum when $\omega^* = \sqrt{\frac{\alpha}{\beta}}$. We impose a condition that when $\xi$ reaches the minimum, the frequency $\omega^*$ is no greater than the lowest
frequency of the domain (Figure 5.1), that is,

$$\omega^* = \sqrt{\frac{\alpha}{\beta}} \leq \omega$$  \hspace{1cm} (5.14)

By inserting Equation 5.12 and 5.13 into 5.14, we obtain the condition \(\gamma \leq 0.8\). Since \(\alpha\) and \(\beta\) are non-negative values, we get \(\gamma \in [0.5, 2]\). So eventually we have \(\gamma \in [0.5, 0.8]\). On the UI, the user inputs a value \(\gamma_{\text{user}}\) within the range \([0.0, 1.0]\) which is mapped to \(\gamma\) by \(\gamma = 0.5 + 0.3\gamma_{\text{user}}\). By default, \(\gamma_{\text{user}}\) is set to 0.0 which leads to \(\alpha = 0.0\). This means that only stiffness damping \(\beta\) is injected into the system which is typically sufficient to stabilize the simulation. Increasing \(\gamma_{\text{user}}\) will add more high frequency motions to the simulation.

**Figure 5.1: Avoid \(\xi\)’s minimum** The minimum point of \(\xi\) is marked in yellow. The lowest natural frequency (in radians) of the object is represented by a red dash line. We impose that \(\omega^* \leq \omega\).
5.4 Asynchronous Parallel Timestepping

Synchronous time integration evolves the entire plant forward in a fixed step from one time instant to the next. It is simple and straightforward. However, the achievable simulation speed is dictated by the stiffest domain which often requires a very small timestep. Such timestep, however, is unnecessary for the domains whose vibration frequency is much smaller. These domains could be time-evolved more efficiently using a much bigger timestep. Therefore, we design an asynchronous integrator which advances different domains at different rates.

5.4.1 Subdividing the Timestep

As presented in Section 5.2, the user is required to set the graphics timestep $\Delta t$, and the quality control coefficient $Q$. The physics timestep will be divided into $2^N$ sub-timesteps per domain according to Equation 5.7. This sub-timestep is the actual simulation step. Our first attempt was to modify Algorithm 1 so that we would do semi-implicit Newmark integrations $N$ times. It was performed by incorporating Line 9 in Algorithm 1 into a for loop from 0 to $2^N-1$. Note that we did not even need to use powers of 2 at this point. We only needed a factor $N$ to satisfy $\frac{\Delta t}{N} \leq \frac{T}{Q}$. This approach, however, did not work very well in practice. Because the local frame (polar decomposition), relative and absolute kinematics were updated only once before the integrations. Vibrations and instabilities could occur for large deformations and small values of $Q$, as the integrator was trying to move forward with too many steps using obsolete kinematic and dynamic information.

To fix this problem, we set the subdivision factor to be a power of 2 which satisfies Equation 5.7. This guarantees that we can update domain’s kinematics and dynamics uniformly in time. Note that if domain is rigid, we set $N = 0$ if it is also the root, otherwise, we set $N$ equals to its parent’s $N$. If a sub-tree is found rigid, namely, all domains within are rigid, we set $N = 0$ for all domains in the subtree. This always happens when a large number of leaves recognize a twig as their parent. In such cases, this saves a significant amount of computation time.
We update the absolute and relative kinematics once every \( \ell \) timesteps, where \( \ell \) is also a power of 2 and varies with each domain. We will explain how to compute \( \ell \) later in this section. Assume the domain being simulated is domain \( i \). When the integrator is at the beginning of each \( \ell \) timestep sequence, it will first update (i) absolute kinematics, (ii) system forces, and interface forces, then (iii) do \( \ell \) semi-implicit Newmark integrations forward, (iv) compute local frame and relative kinematics for every child domain, and finally (v) save updated absolute kinematics \( v_i, a_i, \omega_i, \alpha_i \) to domain \( i \)'s buffer, and relative kinematics \( x_{ij}, A_{ij}, R_{ij}, v_{ij}, a_{ij}, \omega_{ij}, \alpha_{ij} \) (see Section 3.2) to the buffer of its child domain \( j \). The buffer size for absolute kinematics of domain \( i \) is \( 2^{N_i}/\ell_i \) while the buffer size for relative kinematics of domain \( j \) is \( 2^{N_j}/\ell_j \). Using the buffer prevents the parent from waiting for the child in the parallel algorithm presented later in this chapter since the child needs kinematic information from the parent. If domain is rigid, steps (ii),(iii) are skipped. Now we describe how to determine \( \ell \). We assume a general case where domain \( j \) has a parent domain \( i \) and \( n \) child domains, \( k_0, k_1, \ldots, k_n-1 \). Then \( \ell_j \) is computed as:

\[
\ell_j = \min\{ \left[ 2^{N_j-N_i} \right], \left[ 2^{N_j-N_{k_0}} \right], \left[ 2^{N_j-N_{k_1}} \right], \ldots, \left[ 2^{N_j-N_{k_n-1}} \right] \} 
\] (5.15)

where \( N_i, N_j, N_{k_0}, N_{k_1}, \ldots, N_{k_n-1} \) are the exponents for these domains. If domain \( j \) is the root (no parent) or a leaf (no children), we simply remove the related terms in Equation 5.15. Figure 5.2 illustrates a concrete example of our timestepping algorithm working on a tree.

### 5.4.2 Parallel Simulation

Since no loop exists in the tree structure, it is possible to simulate domains in parallel. The fundamental idea is as follows. We setup simulation tasks and push them in a work queue. The task starts when the scheduler pops it from the queue. It happens when a CPU thread (core) becomes available. Note that the work queue is a global, critical resource and the “push/pop” operations must be serialized. Once created, all CPU threads are kept alive, executing or waiting the tasks until all the work is done. Each domain has \( 2^N \) simulation steps. Based on the content of the tasks, we have designed and implemented two scheduling schemes:
5.4 Asynchronous Parallel Timestepping

Figure 5.2: Asynchronous Timestepping: Each independent rectangle represents a simulation step of one domain. Rectangles have the same width indicating the graphics timestep (e.g., 1/30 sec) is fixed for every domain. Every rectangle is subdivided into $2^N$ grids where $2^N$ is the subdivision factor of the domain. Each grid represents a physics timestep for a domain. An arrow pointing from domain $i$ at time instant $t_0$ to domain $j$ at time instant $t_1$ means that the domain $j$ requires domain $i$'s kinematic information at time instant $t_0$ to update its own kinematics at time instant $t_1$. The update frequency $\ell$ is shown beside each rectangle.

Scheme 1: Atomic task (fundamental amount of work by each thread) is to simulate a domain $2^N$ physical steps ($N$ varies with each domain, as discussed in previous section). This means simulating a single domain occurs exclusively on one thread. Data dependency happens when a domain needs its parent’s kinematic information. This will not be an issue since we adopt breadth first search for the tree traversal. Domain is fully simulated prior to any of its children. In this process, domain assumes the parent has provided the necessary data to the proper internal storage location, and then reads from it. When the task is completed, all the child domains, which can be timestepped in parallel, will be pushed into the work queue. Algorithm stops after all domains are simulated.

Scheme 2: Atomic task is to simulate a domain $\ell$ physical steps ($\ell \leq 2^N$, see Section 5.4.1). The collection of tasks is represented by a directed acyclic graph (DAG) (Figure 5.3) with a node for each task and an edge for each constraint (data dependency). The DAG gives an order when certain tasks must be performed earlier than others. Compared to the first scheme, there is one more data dependency here, that is, the task to integrate domain from step $k$ ($k \geq 1$) cannot start if step $k - 1$ has not been computed. In order to generate a valid task sequence, we
5.4 Asynchronous Parallel Timestepping

**Figure 5.3: Directed acyclic graph:** Each node is a simulation task while each directed edge is a constraint representing the data dependency.

employ the standard topological sort algorithm. Traversal starts from simulating root domain at timestep 0. Once the job is done, program writes all intermediate results that may be needed by the following tasks internally to the buffer. A new task is triggered immediately when all its constraints (there are at most two) are satisfied.

**Comparison** The second scheme provides a more compact schedule plan than the first one. Theoretically, it means the simulation is expected to be faster. However in practice, we found it is slower than the first scheme in most cases. The reason is buffers storing the kinematic information may cause read-write conflict between child and parent domains. To prevent this conflict, additional locking is needed which imposes additional complexity and overhead to the algorithm. When the number of tasks is large, the *lock/unlock* operation happens frequently, causing significant decrease in performance. As opposed to the second scheme, the first scheme does not require locking. By default, our simulator uses the first scheme. User is allowed to switch to the second scheme dynamically during the run time. The system performance has been compared and illustrated in Table 5.1.
Table 5.1: Simulation statistics for #triangles, #total domains, # flexible domains, total # of reduced DOFs, frame rate (fps) from our SIGGRAPH 2013 paper [106], quality control coefficient, frame rate (fps) for Scheme 1, domain-based multicore algorithm, frame rate (fps), for Scheme 2, DAG-based multicore algorithm Machine specs: Intel Xeon 2×8 cores 2.9 GHz, 32 GB RAM, Nvidia GeForce GTX 680, 2GB RAM.
Chapter 6

Randomized Wind Force Field

Wind plays a very important role for adding natural-looking variety to plant animations. This chapter describes how we create a randomized wind force field used in our simulator. Our implementation is based on the improved Perlin noise algorithm presented by [77]. We also refer the reader to [89] as good supplementary materials. We begin by describing the requirements for a correct and robust randomized wind field. Next, we provide implementation details on the 4D Perlin noise algorithm. Then we extend technique to construct a randomized wind. Finally, we demonstrate how to apply the wind to the plants in our system.

6.1 Features of a Randomized Wind Force Field

Wind is an important building block in plant simulation. A good wind force field incorporates (but is not limited to) the following features [89]:

- It is not spatially or temporally uniform, namely, it has a controlled way of adding randomness to the output.

- It produces a repeatable pseudorandom force vector for every input pair, position and time instant.

- It is smooth which means it has band-limited spatial and temporal frequency.
6.2 Improved Perlin Noise Algorithm in 4D

- It does not show obvious repeating patterns.
- Its spatial frequency is invariant under translation.

The improved Perlin noise [77] satisfies these requirements, and we adopt it in our work.

6.2 Improved Perlin Noise Algorithm in 4D

The standard improved Perlin noise algorithm takes a 3D position as input. We extend it to incorporate time as the fourth dimension. In general, user inputs a real quadruple \((x, y, z, t)\) where \((x, y, z)\) is the 3D position and \(t\) is the time instant. The noise at this 4D position is obtained by first computing a pseudorandom gradient at each of the sixteen nearest quadruple nodes (the “corners”) on the integer 4D lattice, and then performing spline interpolation (see [77]).

Let \((x', y', z', t')\) denote the coordinate of one corner where \(x' = \lfloor x \rfloor\) or \(\lfloor x \rfloor + 1\), \(y' = \lfloor y \rfloor\) or \(\lfloor y \rfloor + 1\), \(z' = \lfloor z \rfloor\) or \(\lfloor z \rfloor + 1\) and \(t' = \lfloor t \rfloor\) or \(\lfloor t \rfloor + 1\). The first step is to compute a pseudorandom gradient for \((x', y', z', t')\). We create and store a hash table which contains a random permutation of integers from 0 to 255. This hash table is indexed by the \(x'\) coordinate first. The value at position \(x'\) is added to \(y'\) coordinate and the sum is used to index the hash table again. This process is repeated until \(t'\) is incorporated. The result is a pseudorandom integer for the corner \((x', y', z', t')\). This pseudorandom integer is used to look up into a table of 4D gradient vectors. The content of this 4D gradient table is precomputed in the initial stage.

We refer the reader to [89] for more details. In the second step, we obtain a noise value for the corner \((x', y', z', t')\) by computing a dot product between the gradient vector and the fractional

---

Figure 6.1: Visualizing the Perlin wind field: The bounding box of the wind field is shown in red. The spacebox of the wind field has been subdivided at a resolution of \(4 \times 4 \times 4\). We sample and visualize the Perlin wind (blue line segment) at every grid point (shown in green, \(5 \times 5 \times 5\)). To improve visualization, the noise contains a single frequency component.
6.3 Computing the Randomized Wind

Our wind consists of a constant wind (a 3-vector) with the given direction and magnitude, and a randomized spacetime Perlin noise (a 3-vector). Such a randomized noise is able to model strong, turbulent winds that cause our plants to deform to large deformations. The noise is made up of several components at different frequencies. If they are sorted in ascending order by frequency, the contribution of each component follows an exponential decay function, that is, \( g(i) = (1 - d)^i \) where \( d \) is the decay rate within the range \([0, 1]\), and \( i \) is the frequency index and \( g(i) \) is the contribution of the \( i \)th frequency.

Therefore the input to our algorithm includes the direction and the magnitude of the constant wind, the position of the sample points \((x, y, z)\), the time instant \( t \), the initial amplitude of the randomized wind \( A \), the number of frequencies, and the decay rate \( d \). We now explain how to compute each component of the randomized wind. This is done by concatenating the sample spatial position and time instant, scale it with the current frequency, and provide it to the Perlin noise generator (Section 6.2) to generate an initial noise. We multiply the initial noise with the contribution \( g(i) \) and the initial amplitude \( A \) to obtain the final noise. We add this final noise to the constant wind to form the randomized wind. We repeat this process for the other 2 components of the wind.

6.4 Applying Wind Forces to the Plant

We compute the axis-aligned bounding box of the scene and scale it by a ratio (by default, \( ratio = 1.2 \)). The new bounding box defines the space of the randomized wind field. To visualize such a vector field, we first divide the space into cubic voxels based on user-defined resolution. Then we sample the wind force at every grid vertex and render it on the screen.
Figure 6.2: Applying Perlin wind to the plant: The tea bush (*Camellia Sinensis*) is placed in a Perlin wind field. (a) The tea bush is swinging in the wind. (b)(c) Visualization of the wind forces on sample points of the tea bush. The sample points are shown in red. The wind forces applied on sample points are shown in green. For better visualization quality, only one point is sampled per domain (number of samples: \( s = 1 \)) (d) Visualization of the Perlin wind field. The noise contains 3 frequency components, and the decay rate is 0.05.
6.4 Applying Wind Forces to the Plant

(Figure 6.1) In order to apply the wind to the simulation, we sample vertices on the volumetric mesh where the wind forces will be applied. A naive idea is to use all the vertices on the surface of the volumetric mesh. This is not practical since Perlin wind evaluations and reduced force projections are computationally expensive and the number of sample points is too large. The solution adopted in our system is as follows. The user supplies the desired number of samples \( s \leq 1 \) per domain. We use a constant number of samples for all the branches (typically \( s = 5 \)), but the number could be scaled with domain size, e.g., \( s = 1 \) for leaves. The sampled vertices for a domain are then determined automatically, as follows. We first build a mesh graph for our voxel domain mesh, where nodes are volumetric mesh vertices and nodes are connected if they are adjacent mesh vertices. We then use Dijkstra’s algorithm to compute the minimum graph distance of each volumetric mesh vertex to the set of fixed vertices for this domain. Let \( D \) denote the maximum distance. Then, for each \( i = 1, \ldots, s \), we select any vertex with distance \( \lfloor iD / (s + 1) \rfloor \) as our sample. Because plant branches are long and slender, and the running time of Dijkstra’s algorithm is linear in the number of vertices of each domain, such a strategy produces well-distributed samples in negligible time. In order to model the fact that large branches are more exposed to wind, the wind magnitude for a sample point is set to \( \text{mass} / s \) where \( \text{mass} \) is the mass of the volumetric mesh of each domain.

To apply wind forces to the plant, we compute the randomized wind force vector on every sample point on every domain and accumulate it in our simulator (Figure 6.2). In Figure 6.3, we demonstrate the motion of a grand fir subject to the Perlin wind.
Figure 6.3: Grand fir (Abies Grandis) in Perlin wind: Frame numbers are provided.
Chapter 7

System Overview

In this chapter, we provide an overview of our system which consists of two applications. The first one is the botanical preprocessor and the second one is the botanical simulator. We introduce the graphical user interface (GUI) for each program separately and describe how to tune the parameters on the GUI.

7.1 Botanical Preprocessor

7.1.1 Command Line

User can use the command-line (no GUI) if the hierarchy is already known. Pass the hierarchy via -h. User has to also pass -a to avoid the GUI. In this way, user can pre-process the tree completely in the shell, without any user interaction. The hierarchy file has a simple text format where each line lists a domain and its parent. Root is assigned -1 as its parent. You can save a hierarchy from the GUI by pressing “Save”.

7.1.2 Preprocessor

User needs to use the GUI if the hierarchy is not known or if it is not connected or contains cycles. When the program is launched, a panel is shown (Figure 7.1).
7.1 Botanical Preprocessor

Figure 7.1: Plant preprocessor  **Left:** Botanical object display window.  **Right:** Preprocessor UI panel

**Selecting fronds and leaves**  This makes it possible to separate the domains into branches, fronds and leaves.  If all domains are branches, user can simply press the “Frond & Leaf Selection Done” button.  Otherwise, select some domains, and press ‘f’ to tag them as fronds or ‘l’ as leaves (Figure 7.2).  The user can delete domains by pressing the ‘del’ key.

**Selecting root**  Select the root domain and press “Set Current Group As Root (F5)” button or ‘F5’ (Figure 7.3).

**Building hierarchy**  This button builds the hierarchy, by performing collision detection on the triangle meshes.  If more than one connected component is detected, and/or there are loops, additional steps are necessary (below).

**Connecting the domains**  The GUI will display the connected components.  Scroll via ‘,’ and ‘.’.  User can delete domains by pressing the ‘del’ key.  To connect two disconnected components, select the first domain (LMB) from the first component, press ‘c’.  Then, select
7.1 Botanical Preprocessor

Figure 7.2: Select fronds and leaves (a) leaves are selected and highlighted in cyan, (b) After pressing ‘l’, leaves are tagged, (c) Fronds are selected and highlighted in cyan, (d) After pressing ‘f’, fronds are tagged.

Figure 7.4: Connecting the domains: Select a domain from the first component and press ‘c’, the domain will be highlighted in blue. Then select another domain from the second component and press ‘c’ again. These two domains will be connected together.
the second domain from the second component and press ‘c’ domain. The two domains are now connected in the domain graph. There is no undo for this. However, after pressing the first ‘c’, user can press ‘C’ to cancel. Continue connecting until there is only one connected component. At any moment, user can press ‘Build Hierarchy’ and the hierarchy will be re-built. This way, user can see how many components were already fixed. User can press ‘s’ to save the connectivity graph to the disk. It is saved to:

output.reducedMultiDomainDeformationFactory.connectivityGraph.<.suffix>.

The “<.suffix>” is one of the following:

1. none: it is automatically saved by the system after doing collision detection to build the connectivity graph;

2. ‘.c + number’: c marks the program is in the resolving disconnected component stage, and the number indicates how many disconnected components are existing. For example: .c7;

3. ‘.b + number’: b marks the program is in the resolving loop stage, and the number indicates how many loops are remaining. For example: .b26;

4. ‘.allResolved’: it means all the ambiguities were resolved.

This functionality allows the user to save the current work and continue preprocessing at a later time. To load the connectivity graph, use -G option and specify the filename when launching the preprocessor GUI via the command line. In this mode, a new button “Load” appears beside the “Build” button. User should go through all the previous steps described above and then click the “Build” to initialize the hierarchy for the first time. Then press the “Load” button to load the connectivity graph and continue preprocessing from there.

**Resolving loops:** If there are loops, they must be resolved manually using the provided interface. Our software uses an intelligent algorithm which minimizes the amount of user work required (see Section 4.1). As opposed to visiting every pair of colliding branches and asking the user if this is a genuine collision or not, the algorithm automatically selects a much smaller
7.1 Botanical Preprocessor

Figure 7.5: Resolving Loops: (a) Loop visualization. Branches, fronds and leaves are rendered. They can be hidden by pressing ‘1’, ‘2’, ‘3’. (b) Visualization of the loop while only branches are rendered. (c)-(e): cycle through the edges by pressing ‘{’ or ‘}’ and break the loop by pressing ‘d’.

A set of loops (call it S) in such a way that resolving those loops provably removes all loops. For each loop in S, the user needs to specify which edge in this loop is spurious. For example, if a branch accidentally collides with another branch, that would be a spurious edge. Note that an “edge” here refers to a pair of colliding branches. Nodes means branches while edges means colliding branches. (Figure 7.5) The spurious edge is selected using keys ‘{’ and ‘}’. You will see the branches in the loop in yellow, and the two branches whose joining edge is currently selected are shown in red and pink. If this is the edge to be removed, press ‘d’. Otherwise use ‘{’ and ‘}’ to select the correct spurious edge, and then press ‘d’. User can also go to previous or next loop using ‘[’ and ‘]’, but this is typically not needed. Once all the loops are fixed, the program will allow the user to continue to the next stage (model reduction). User can undo with ‘z’. And the branches can be hidden/shown using ‘1’ which is useful to see just the loop.

We summarize the keys for loop resolution:

- ‘{’ and ‘}’: go to previous or next unresolved loop
- ‘[’ and ‘]’: cycle through neighboring branch pairs within the current loop
- ‘d’: break the cycle at the current branch pair (edge) (shown in red and pink)
- ‘z’: undo the previous cycle resolving operation
- ‘1’: hide or show the entire tree (keeping only the loop visible)
- ‘s’: save the connectivity graph
This functionality is also available on the UI panel. Note that once the hierarchy is a tree, user can save it to a file via the “Save” button. It is saved to:
output.reducedMultiDomainDeformationFactory.hierarchy.

This hierarchy can then be loaded to the preprocessor via the ‘-h’ command-line option. (In this way, the user only has to perform connections and loop resolution once.) Note that the user can load the hierarchy and avoid UI as long as no domains needed to be deleted before the loading. If deletion is required, user has to use the UI. However, connectivity graph (if saved before) could still be loaded to save the preprocessing time. Also, once the hierarchy is built, the program will output the employed branches, fronds and leaves to files:  
out.<d>.obj
where d is a digit and can be ‘0’, ‘1’ and ‘2’ corresponding to branches, fronds, and leaves. In this way, even when some domains were deleted, command line mode can still be used with -h.

**Model reduction**  After the domain graph has been connected and all loops resolved, user needs to press “Perform Model Reduction”. This will compute simulation meshes (via voxelization), determine the interfaces and perform model reduction as in [8]. At the end of this step, all the necessary simulation files will have been created. Now, you are ready to launch the runtime driver with the simulation config file as the only input.

**Aging mode**  This option is accessible via the “Configure” rollout menu in the preprocessor. When the aging mode is on, branch stiffness is assigned procedurally. It is enabled by default. If user turns it off, a tree has constant material stiffness everywhere. Note that the user can change this at runtime very easily (no additional preprocessing required). In the pre-processor, the user is only selecting whether the aging mode is enabled by default when...
launching the runtime driver.

### 7.1.3 Useful Tools

In this section, we present some independent, useful tools along with the preprocessor.

**Prune a plant:** The user is able to create plants obtained by keeping the first $N$ levels of a plant, using the “prunePlant” utility. Given the input obj mesh and the hierarchy file, this will create the obj mesh that keeps the first $N$ hierarchical levels.

**Create a forest:** This utility helps the user to create a forest (see Section 4.2). It consists of following steps:

1. Pre-process the individual tree/plant (using the preprocessor) that will be replicated into a forest.

2. Create an instance description file (i.e., forest file, extension .forest). This text file looks like this:
   
   ```
   <num plants>
   x,y,z of plant 1
   x,y,z of plant 2
   ...
   ```

3. Use the utility “instancePlants” to create domain and parent list filenames for the forest. Input parameters are the domain and parent list filenames created by the preprocessor. Usually they are called “base-<obj name>-domainList.txt” and “base-<obj name>parentList.txt”.

For example:

```bash
./instancePlants base-myBigTree-domainList.txt \ 
   base-myBigTree-parentList.txt myBigForest.forest \ 
   base-myBigTree-domainList.forest.txt \ 
   base-myBigTree-parentList.forest.txt
```
4. Open the simulation configure file and modify the following lines (even better, copy the simulation configure file and edit the copy):

*domainList
#base-domainList.txt
base-myBigTree-domainList.forest.txt
*parentList
#base-parentList.txt
base-myBigTree-parentList.forest.txt

Here, # means that the line is disabled. In other words, we are disabling the old hierarchy and enabling a new one.

The forest simulation can be launched at this point. Currently, only replicated copies of the same plant can be used. But this could be changed to allow different plants in the same scene.

### 7.2 Botanical Simulator

Our real-time simulator must be launched with a simulation configure file generated by the preprocessor. This simulation file contains several initial settings, such as timestep, damping coefficients, number of threads, camera position and angles, window size, etc., and data filenames such as the domain list filename and parent list filename. Figure 7.6 gives a screenshot of the runtime driver. User is able to pull on the plant with the left mouse button, apply gravity (on GUI), or use Perlin wind (on GUI, or specified in configure file).

#### 7.2.1 Simulation Control

The simulation control panel is presented in Figure 7.7.

**Procedural damping** User is able to control the damping via the dimensionless $\xi$ parameter (see Section 5.3).
7.2 Botanical Simulator

Figure 7.6: Botanical Simulator: **Left**: Simulation display window. It allows user to interactive with the simulation by mouse. **Middle**: Frequency tuning panel. User could adjust the material stiffness on this panel. **Right**: Simulation control panel. User could change timestep, damping, mouse force strength, wind etc. on this panel.

1. $\xi = 0$: no damping
2. $0 < \xi < 1$: under-damped system
3. $\xi = 1$: critical damping
4. $\xi > 1$: over damped system

User can control the value of $\xi$ as a function of the frequency of each domain. This is done by modeling $\xi$ as a constant function of $\nu$ for $\nu \leq \nu_{\text{low}}$ and $\nu \geq \nu_{\text{high}}$, and a straight line in between (as explained in Section 5.3).

**Quality** Quality refers to the *quality control coefficient* introduced in Section 5.2. The simulator uses asynchronous timestepping (see Section 5.4). It will step each domain at such a timestep that $Q$ steps are made per longest oscillation period (lowest frequency) of this domain. Parameter $Q$ (quality) therefore controls how many integration steps are performed, relative to the period of the domain. $Q = 100$ is a good initial value. Higher values of $Q$ will produce
better motion (less numerical damping), at the expense of more computation. The “Set to 1x speed” button applies a heuristic to tune quality in such a way that the simulation runs at the same rate as physics.

**Timestep**  This sets the graphics timestep on a physical scale. It is usually set to $1 / 30$ s or $1 / 24$ s. When Timestep quality control is enabled, asynchronous timestepping is used. The timestep for each domain is subdivided automatically, as explained in the “Quality” section above (also see Section 5.2).

**Liveliness and Inertia**  These two important parameters control the strength of system forces (forces applied to a branch because of the motion of its parent, Section 3.3) and inertia forces (forces applied to a branch because of the motion of its children, Section 3.3). Tuning these values well is important. Default liveliness is 1.0 but the user can increase liveliness to 2.0 or 3.0, causing the tree to be more lively. In such cases, $Q$ (quality) may need to be increased somewhat. User can also decrease the liveliness to a value below 1.0. This may help stabilizing the simulation without increasing $Q$.

**Hierarchical levels:**  This makes it possible to simulate only a subset of the tree (levels up to a given hierarchical depth). This is controlled via “Max simulated level”. You can also make levels beyond a certain level rigid, via “Max flexible level”. “Max simulated frequency” is a cutoff frequency, namely, domains above that frequency
7.2 Botanical Simulator

will be made rigid as they vibrate too quickly to be seen in the animation. Typically it is set to 20 Hz.

**Perlin wind**  See Chapter 6. User can enable it on the GUI. Standard Perlin wind parameters are used. User can specify a constant wind direction plus the space-time 4D Perlin wind. To visualize the Perlin wind, press ‘c’ to show the Perlin wind field, and/or ‘C’ to show the Perlin wind forces applied on sample points.

**Animation player**  Makes it possible to record the simulation to a buffer and then play it back.

**Rendering**  We describe some rendering switches on keyboard here. Use ‘e’, ‘E’, ‘w’, ‘W’ to show or hide solid voxel mesh, solid triangle mesh, wireframe voxel mesh and wireframe triangle mesh. By pressing, ‘1’, ‘2’, ‘3’, user can show/hide branches, fronds and leaves.

### 7.2.2 Frequency Tuning

This is an independent GUI panel to tune the aging mode and stiffness of each hierarchy level at runtime. See Figure 7.8.

**Frequency tuning**  This panel makes it possible to set the stiffness multiplicator individually for each individual level. A global multiplicator (multiplies all levels simultaneously, in addition to the level multiplicators) is available via “Freq Scaling”. User can enable “aging mode” (procedural stiffness, Section 5.1) or “absolute mode” (constant stiffness). With procedural stiffness, stiffness is determined based on age of each branch. Age is determined based on the branch geometry (size). Aging mode has the exponent parameter that controls how aggressive the aging mode is. A value of 2.0 is a good choice. Higher values will make small branches more lively. A value of 0.0 is equivalent to absolute mode (constant stiffness). Aging randomization randomizes the frequency somewhat, so that we don’t get an entire tree resonating at 1Hz in the aging mode with exponent 2. Frequencies will be multiplied with
a random factor sampled uniformly on the interval \([1.0/(1.0 + agingRandomization), 1.0 + agingRandomization]\). Note that user needs to select the levels before clicking on “Upload”. Important: aging mode or absolute mode changes will only apply to the selected levels. User can select all levels via “Select all”. A good way to tune a tree is to set the first few levels stiff, i.e., 3.0, 5.0 or 10.0. Then, make higher levels softer, i.e., 1.0.

![Frequency tuning panel](image.png)

**Figure 7.8: Frequency tuning panel**
Chapter 8

Conclusion

We presented a robust system for stable physically based simulation of anatomically realistic botanical systems. We first presented a real-time algorithm for simulation of reduced nonlinear flexible multibody systems undergoing large deformations. The algorithm was made possible by deriving the first and second time derivatives of the rotation matrix used in polar decomposition. The algorithm supports localized deformations, requires no constraints, and runs in time linear in the number of domains. This algorithm forms the foundation of our botanical simulator.

We then demonstrated a preprocessor to pre-process “polygon soup” plant geometry for domain decomposition simulations. Our system scales to the complexity of real-world adult trees, flowers and bushes. We have pre-processed over 100 plants from several publicly available vegetation model libraries. Our system accommodates unorganized, unprocessed triangle input geometry, including billboards. We extended our simulator to support fracture, interactive plant design, and forest simulation.

Third, we proposed a series of procedural methods to automatically tune timestep, stiffness, and damping coefficient for every domain. These tuning techniques could be incorporated into any modal-based FEM simulation method. We extended our single-core multi-domain dynamics algorithm to an asynchronous parallel timestepping algorithm that utilizes multi-core computing. We demonstrated that this algorithm greatly enhanced the simulation speed and numerical stability under large deformations.
Finally we described how to produce a robust, 4D randomized wind force field necessary for plant simulation.

**Limitations and future work:** Our multi-domain dynamics algorithm is limited to domain topologies without loops. Several bodies connected and rooted to the ground form a loop, and must be simulated as a single domain in our system. Loops could be closed by adding springs; or more formally, using extensions paralleling those for rigid articulated chains [26]. We assume that the domain interfaces undergo only a small amount of non-rigid deformation. This assumption is valid when the interfaces are small, and worked well in our examples. Flexible interfaces could be simulated by adding additional “boundary” modes to each domain [92]. Related to that, it may seem plausible to add additional terms to the equations of motion that would explicitly couple the elastic deformations of two domains meeting at an interface. Note that for strictly rigid interfaces such terms are identically zero, and that any external forces already are properly propagated to parent domains via our interface forces. In our work, frames follow the motion imposed by the parent exactly, without any freedom for deviation. Some of the six degrees of freedom could be relaxed by introducing joints, which would lead to a method supporting both articulation and large deformations.

Our preprocessor cannot handle plants that are in continuous contact with their natural environment, such as a vine climbing a mesh fence or ivy climbing a tree. For plants in simpler (ground) contact such as zucchini or watermelons, frictional contact could be handled using constraint solvers, or even penalty forces. Our system avoids loops, which has not been a problem in practice as the vast majority of plants do not have loops. Loops could be addressed using penalty forces. Our material parameters could in the future incorporate mechanical properties or other observation data from real plants. Our instancing can be easily extended to domains that consist of several disjoint components, each texture-mapped with a distinct texture map. A more challenging case, however, would be to support *hierarchical instancing*, where instances themselves consist of replicated copies, e.g., replicated blooms, each consisting of replicated but otherwise identical petals. We note, however, that many blooms in practice are modeled as a single global quadrilateral billboard, replicated in different orientations, where our single-
level instancing is sufficient. Methods that use alternative simulation bases [29] may be an interesting approach to simulate plants. We use a semi-implicit integrator to timestep our models which provides stability but also introduces artificial damping. It would be interesting to seek integration schemes that can work with model reduction and that can avoid artificial damping. Simulator realism would be improved by handling self-collisions, and simulating two-way coupling between the wind and the plant.
Bibliography


