ANIMATING JELLYFISH THROUGH NUMERICAL SIMULATION AND SYMMETRY EXPLOITATION

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
David Rudolf

PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

ABSTRACT

This thesis presents an automatic animation system for jellyfish that is based on a physical simulation of the organism and its surrounding fluid. Our goal is to explore the unusual style of locomotion, namely jet propulsion, which is utilized by jellyfish. The organism achieves this propulsion by contracting its body, expelling water, and propelling itself forward. The organism then expands again to refill itself with water for a subsequent stroke. We endeavor to model the thrust achieved by the jellyfish, and also the evolution of the organism's geometric configuration.

We restrict our discussion of locomotion to fully grown adult jellyfish, and to the resonant gait, which is the organism's most active mode of locomotion, and is characterized by a regular contraction rate that is near one of the creature's resonant frequencies. We also consider only species that are axially symmetric, and thus are able to reduce the dimensionality of our model. We can approximate the full 3D geometry of a jellyfish by simulating a 2D slice of the organism. This model reduction yields plausible results at a lower computational cost. From the 2D simulation, we extrapolate to a full 3D model. To prevent our extrapolated model from being artificially smooth, we give the final shape more variation by adding noise to the 3D geometry. This noise is inspired by empirical data of real jellyfish, and also by work with continuous noise functions from the graphics community.

Our 2D simulations are done numerically with ideas from the field of computational fluid dynamics. Specifically, we simulate the elastic volume of the jellyfish with a spring-mass system, and we simulate the surrounding fluid using the semi-Lagrangian method. To couple the particle-based elastic representation with the grid-based fluid representation, we use the immersed boundary method. We find this combination of methods to be a very efficient means of simulating the 2D slice with a minimal compromise in physical accuracy.

ACKNOWLEDGEMENTS

Most of all, I would like to thank Professor David Mould for his years of support, financially, academically, and socially. David may be the most supportive supervisor I have ever heard of, and is also a good friend.

Many thanks also to Professor Eric Neufeld for his unconditional encouragement and endless sarcasm. Eric single-handedly convinced me to go to graduate school, yet kept me from taking it too seriously.

Thanks also to Mike Baribeau and SED Systems for helping finance my graduate education and giving me valuable industry experience. Without them, I would be just be another institutionalized graduate student.

Thanks also are in order for the various members of the Imaging, Multimedia, and Graphics Research Group at the University of Saskatchewan for providing me with interesting discussion and ideas over the years.

Thank you to the numerous friends and family members who supported me through this whole ordeal and/or providing me with ample distraction and opportunity for procrastination.

CONTENTS

List of Figures

List of Symbols

Symbol	*Description*	*Sections*
$\vec{a}_i$	The acceleration of particle i.	2
$\vec{c}$	The axis of collision between two bodies.	5.2.1
c	A scalar distance by which an umbrella point is perturbed to introduce variation to its surface.	7.3
d	The discrete distance of a point in the jellyfish rendering model from some target point.	7.3
e_{W_i}, e_{E_i}, e_{N_j}, e_{S_j}	The edge flags for the west, east, north, and south edges (respectively) of a pressure grid, used for solving the Poisson equation at the fluid boundaries.	4.1.4
$f(y,t)$	The derivative function for a differential equation.	2.4
$\vec{f}$	The acceleration vector field of a fluid due to external forces.	3.1
$\vec{F}_i$	A force acting on particle.	2
$\vec{g}$	The gravity vector.	2.3.2
$\vec{G}(t)$	An intermediate calculation of the fluid velocity field, used when solving the Poisson equation for pressure.	4.1.2, 3.3
h	The radius of an SPH smoothing kernel.	5.1
I	The identity matrix.	4.2.3
l_c	The current length of a Hookean spring between two points.	2.1
l_r	The rest length of a Hookean spring between two points.	2.1
m_i	The mass of particle i.	2
n_p	The number of particles in an SPH simulation.	5.1.2
$P(\vec{w})$	A projection of a divergent vector field $\vec{w}$ to a non-divergent field, achieved using the Helmholtz-Hodge decomposition.	4.2.1
p	Pressure, specifically the scalar pressure field of a fluid.	3.1
q	An arbitrary scalar field.	4.2.1, 5.1
Q	An interpolation function.	7.2, 7.3
$\vec{s}$	The axis of symmetry for a jellyfish umbrella.	7.3

t	Time.	2
$\vec{u}$	The velocity of the fluid, specifically a vector field of velocities.	3.1
V_i	Volume, specifically the volume of influence for an SPH smoothing kernel.	5.1
$\vec{w}$	An arbitrary vector field.	4.2.1
w_b	The width of a boundary object in an SPH simulation.	5.3
$W(\vec{x})$	A smoothing kernel function for SPH systems.	5.1
$\vec{x}_i$	The position of particle i.	2
$\vec{x}_{ij}$	The vector between particle positions $\vec{x}_i$ and $\vec{x}_j$.	2.1
$y(t)$	The value function for a differential equation.	2.4
α	Sensitivity parameters for a PID controller.	1.2.5, 7.1
ε_p	The residual error field of the pressure grid, used to estimate the accuracy of a given solution to the Poisson equation for pressure.	4.1.4
$\vec{\varepsilon}_p$	A random displacement vector given to the initial configuration of an thermalized SPH simulation.	5.1.2
γ	The scalar ratio of contraction for a jellyfish. Essentially, this number tells us by what factor a jellyfish will contract its umbrella's radius.	7.1
κ_l	The coefficient of elasticity for Hookean springs.	2.1
κ_d	The coefficient of drag for a fluid.	2.3.1
κ_p	The gas constant for pressure forces in an SPH simulation.	5.1
κ_θ	The coefficient of elasticity for angular springs.	2.2
ν	The viscosity of a fluid (i.e. the inverse of the Reynolds number).	3.1, 5.1
θ_{ijk}	The angle between vectors $\vec{x}_{ij}$ and $\vec{x}_{jk}$.	2.2
ρ	Density, specifically of a fluid.	3.1, 5.1
σ_j	The longitudinal angle of a point $\vec{x}_{i,j}$ that is on a 3D umbrella surface.	7.3
τ_i	The torque acting on particle i.	2.2
ϑ	The thickness of a jellyfish umbrella.	7.2
ω	Relaxation parameter for the successive over-relaxation solver for the Poisson equation of pressure.	4.1.4

CHAPTER 1

INTRODUCTION

For decades, audiences have been delighted by computer animations, from early work in Disney's "Tron", to more modern works such as "Toy Story", "Shrek", and "Futurama". The computer animation industry is growing substantially as computing power is increasing and making more sophisticated animation systems possible. Animators are decreasingly burdened with the low-level details of character animation, and can instead focus their attention on higher-level aspects such as story and content. We are also entering an era in which many complicated animations can be carried out in real time, and are being utilized by the video game industry. Such animation systems increasingly need to handle interaction with arbitrary character behavior. User-controlled elements of the system may do unpredictable things, and the animation system needs to be able to respond.

To date, interactive animation techniques have been developed for a wide variety of characters. Miller [68] developed a physically inspired model of snakes, worms, and caterpillars. Fish have been modeled by Tu and Terzopoulos [113], and other marine life have been simulated by Frölich [35]. Raibert and Hodgins [90] studied the motion of legged mammals in general. Legged locomotion has also been studied in much detail by the robotics community [3, 12, 11]. An incredible amount of research has been aimed at animating various aspects of human motion [48, 100, 111, 114]. However, little has been done to model invertebrates, especially in a marine environment.

This thesis describes a means of animating jellyfish that is based on physical simulation. We do not aspire to create a physically accurate model of jellyfish, but merely develop a means of generating animations of the organism that would be convincing to the general population. However, we choose to utilize simulation techniques so that our virtual jellyfish could physically interact with other objects in the scene. We also use our model to explore the unique mode of locomotion that is utilized by jellyfish, and gain understanding as to how such locomotion could be utilized by the graphics community. We aspire toward a model that will run on conventional hardware at an interactive rate. We do so with a series of model reductions, and physically simulate the system on the reduced 2D model. We then extrapolate the results of our simulations to higher resolutions and also to a 3D space. In doing so, we are able to efficiently animate jellyfish whose motions are qualitatively similar to that of real jellyfish, and whose appearance is visually appealing. Figure 1.1 shows captured video of a jellyfish in swimming, illustrating the kind of motion that we aspire to animate.

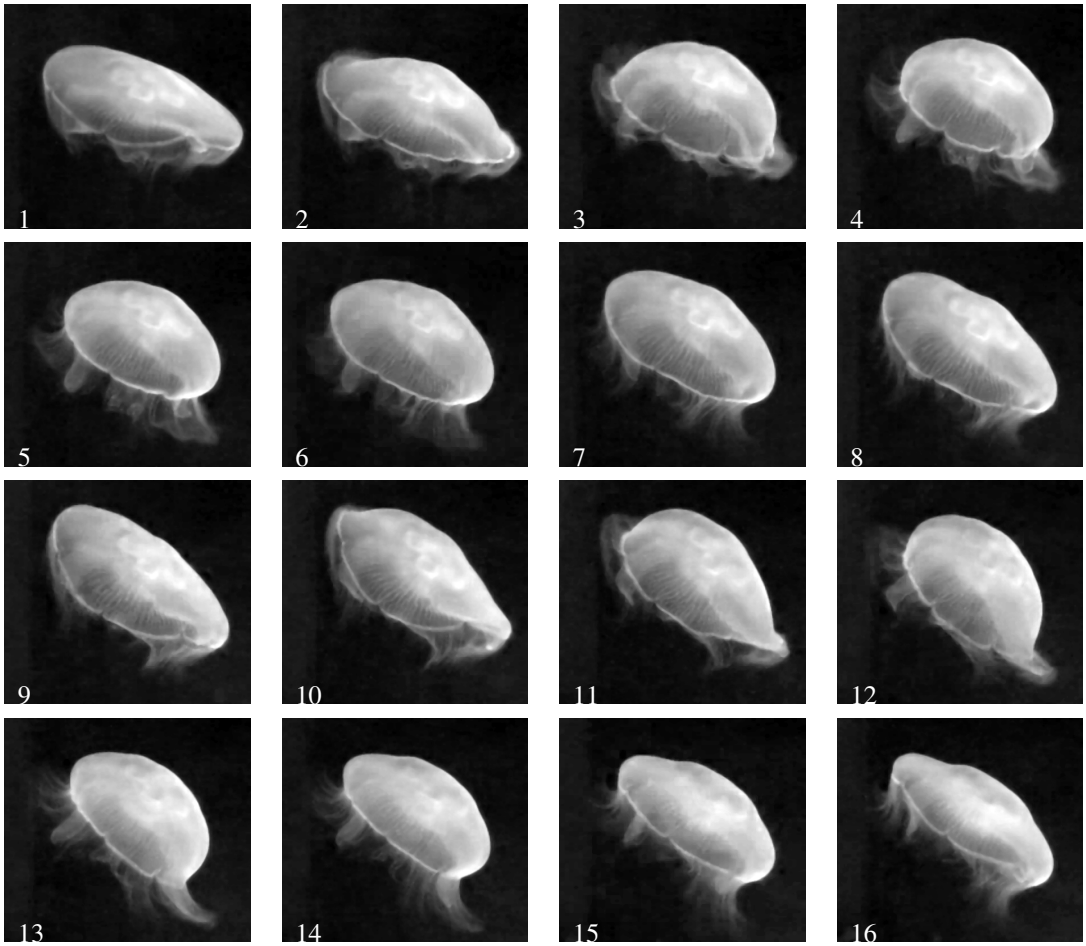

Figure 1.1: Captured footage of a jellyfish swimming, as filmed by Cummins [19].

1.1 Motivation

Jellyfish present an animation challenge. They move about by means of jet propulsion: they contract their bodies to expel water and thrust themselves forward. Jellyfish are not overly complex creatures, in that they are invertebrates with no significant cranial capacity [2]. As Beer et al. [3] discuss, we can learn much about motion and control systems by studying and attempting to mimic biological or otherwise natural systems in the world around us. Their work aims to further the field of robotics by designing robots that act like organic creatures, specifically insects such as cock-roaches [107]. Beer et al. [3] believe that we can avoid much experimental design by building on the ideas that have proven themselves through millennia of evolution. Beer et al. [3] chose to mimic a relatively simple creature such as a cock-roach so that they could more intimately concentrate on the smaller details of the creature's motion.

In our case, we investigate ways to control invertebrate marine life for animation purposes. We also find jellyfish particularly interesting because of their relative simplicity. Much current research in animation is

aimed toward active characters that have complex physical systems, like humans and vertebrates in general [48, 82, 114]. These systems typically have a large number of muscles, each of which needs a separate controller to activate muscles and otherwise cause the body to go into motion. For some creatures such as birds, snakes, and fish [117, 69, 113], approximations to the physical systems are used so that the animation has some semblance of realism. For humans, it is common to resort to motion capture [45, 58]. However, any captured data is specific to the motion that it represents, and creating new motions can be difficult. Simulations and procedural models in general allow us to generate more motions. However, the onus on procedural animators is that their models must be designed to produce the desired motions, and thus a deep understanding of the organism is generally required. Our hope is that by learning about this relatively simple animal, we may yet get a better idea of how to extend our ideas to more complicated creatures.

Despite the relative structural simplicity of jellyfish, their lack of skeletal structure and the environment in which they exist make for some interesting simulation challenges. Also, despite the biology community's extensive knowledge of a jellyfish's mechanical properties, little is known about how the organism controls its body and takes advantage of its structure to achieve locomotion. Some work has been done to study jet propulsion in general [21, 64], but these works are focused on engineering a propulsion system for a specific application. There is still much to be learned about how jet propulsion is utilized in the natural world.

1.2 Animation Techniques

Our work with animation is based on physical simulation, but we give a brief overview of other areas in the field of computer animation to give a sense of how our work fits with the field in general. Some areas of animation are similar to our simulation-based model and have direct bearing on our work. Other animation areas are far removed from our work with simulation, but are also plausible techniques to be used to animate jellyfish.

1.2.1 Procedural Animation

The concept of procedural animation broadly encompasses any type of animation that we can automate with a well-defined algorithm or procedure. However, this broad definition encompasses most parts of the field of animation, including simulation. More colloquially, procedural animation refers to any animation technique that uses ad-hoc or artistically chosen methods to generate the desired movement, yet in an automated way. This automation might make use of a repeated parabolic curve to mimic the position of a bouncing ball, for instance. Another example that comes straight from research literature is the model of ocean waves that was presented by Fournier and Reeves [34]. Their model makes use of periodic parametric curves to generate the wave surface, and (in their own words) "kludges" to control the height and concavity of the waves as they approach the shore line. These types of procedural models can be very efficient and convincing, but are usually tailored for specific cases, and do not apply to the general problem.

1.2.2 Key-Framing

Original work in computer animation began in a manner that was similar to traditional animation. For each frame of an animation, the animator would have to configure their model, and then render the scene. Often the spatial continuity between frames was smooth enough that the animator could merely specify a few key model configurations or character poses, and then intermediate configurations can be interpolated from the configurations of the so-called key frames [49]. For the results to be plausible, the key frames and the interpolant must be chosen to suit the desired motion. One modern example of key-framing is actually the jellyfish scene from Pixar's film "Finding Nemo" [1], whereby the umbrella of the jellyfish was animated via key-framing, and the tentacles were simulated with a reduced physics model.

Key-framing has another important application. Suppose we have several independent animations of an object, obtained either artistically, through simulation, motion capture, or some combination of techniques. We may want to combine the individual animations to make a longer one. For example, we may have an animation of a virtual character running, and another of the character tripping and getting up. We may want to combine the sequences into a mid-run mishap, and could do so by inserting keyframes from the tripping sequence into key frames of the running sequence, and allowing the interpolation to blend the two motions together.

Keyframing does have some issues of concern. One such issue is the so-called *foot-skate* problem, whereby the interpolation between key frames may cause a character's foot (or any other object in contact with another) to move in an implausible manner. For instance, the interpolated object might interpenetrate another, or conversely might lose contact with the object, or even just slide about the surface in an artificial manner. These types of problems must be addressed in the final animation.

1.2.3 Motion Capture

To animate complex movement, motion capturing systems are becoming increasingly popular, and decreasingly expensive. The idea of motion capture is to have a creature perform an action, record that action with some capturing technique, fit the recorded movements to a virtual model, and then move and deform the virtual model to correspond to the captured movements. Conventional motion capture involves the use of motion capture suits that have marker lights or reflectors placed at key parts of the creature's anatomy. However, the use of capture suits is not always practical, and so image-based techniques have been developed that use videos from multiple perspectives to reconstruct the motion of the creature [111]. The success of these image-based methods depends on the presence of visible features (either geometric or textural patterns) on the object whose motion is being captured. In fact, Dabiri and Gharib [22] have image-based empirical data of jellyfish, but the organism's lack of features prohibit a detailed analysis of its movement.

A large drawback to motion capture techniques in general is that the creation of a novel movement from a set of captured ones can be difficult and computationally expensive [100], or is limited in the kinds of

interactions that can be edited into the captured data [119]. Thus, difficulties arise when animating the interaction of a captured model with an arbitrary environment.

1.2.4 Rigid Body Simulation and Articulated Skeletons

Simulation by itself is a large area, but we will discuss the specific simulation of fluids and elastics throughout the remainder of this thesis. For now, we pay special attention to the simulation of rigid bodies, because of their wide-spread use in computer graphics. This type of body does not experience any deformation, regardless of the forces acting on it. Rigid bodies are an approximation to bodies which undergo small external forces relative to internal forces that arise from the molecular bonds within the solid. Solids of this nature do not undergo a significant amount of deformation, and thus can be thought of as rigid. We represent the state of a body by the position of its center of mass $\vec{x}$ and its orientation $\vec{\theta}$. In a simple 2D case, $\vec{\theta}$ is a vector with only one element. In this simple case, the motion of the body is described as a combination of its Cartesian velocity $\vec{v} = \frac{d\vec{x}}{dt}$ and its angular velocity $\vec{\omega} = \frac{d\vec{\theta}}{dt}$. Based on forces that are acting on parts of the body, we can compute the changes to both velocities, and simulate the system over time [42, 70].

We can combine many rigid bodies into a skeletal system. Usually, each rigid body is a simple geometric primitive, such as a line or cube. These primitives are then connected by pivoting joints, creating a mesh of rigid segments and joints. Usually, these meshes are more like trees than arbitrary cyclic graphs. For example, to simulate a human hand, one might have a pivot point for the wrist, then another at the base of each finger, plus two more along the fingers to represent the knuckles. The animator then has to define how each joint will pivot at each point in time. Any pivots that occur on a joint that is lower in the tree (e.g., at the wrist) will have an effect on the joints that are higher in the tree (such as the knuckles), but the reverse is not true. We refer to the mesh of rigid segments as an articulated skeleton.

Defining the pivot motions artistically can be tedious. Thus, we can simulate the results by imposing torques on the joints, but also by restricting each joint's range of motion. These torques could be animated directly, or result from physical forces, like gravity or wind. Pai et al. [82] simulated the elastic properties of the human muscular structure to generate torques for their skeletons. However, we also want our rigid bodies to interact with solid objects, or the animator might want to directly manipulate the position of a body that is high up the skeletal tree. For these situations, we could use inverse kinematics [118] to propagate pivot positions from higher parts of the pivot tree to lower parts.

1.2.5 Process Control

A challenge in all simulation-based animation techniques is that we need a way to automatically control the bodies that are in motion. As a simple example, suppose we want to animate a ship that is sailing toward an island. In the simplest case, we would be dealing with still waters and have no obstacles, so we could set a course and merely sail straight for the island. However, sailing is never this simple in reality. One issue that might arise is that obstacles may block our path to the the target island. We might have to go

around a peninsula or navigate through a fjord. In these cases, we could devise a control system that makes predetermined course changes to navigate to the island. This type of controller is referred to as an *open-loop* or *non-feedback* controller because it does not make its control decisions based on feedback from the environment. Instead, open-loop controllers merely have behaviour that is specifically predetermined.

Suppose that we were to add unpredictable elements that affect our system. In our ship example, these elements could be winds and sea currents that pull the ship away from its course. We need a control process that can correct for the effects of unpredictable elements. A control process that measures its environment to determine its action is known as a *closed-loop* controller. Such a controller for our ship would detect if the ship is veering too far to the right, and would react by turning the steering wheel to the left, or vice versa. More quantitatively, if the desired heading at time t_i is $\theta_d(t_i)$, our current heading is $\theta_c(t_i)$, and the wheel is at angle $\theta_w(t_i)$, then we could design a closed-loop steering process as follows:

$$\theta_w(t_{i+1}) = -\alpha_p \varepsilon(t_i), \quad \text{where } \varepsilon(t_i) = \theta_d(t_i) - \theta_c(t_i). \tag{1.1}$$

We call ε the error in our system, and we want to minimize this error. The parameter α_p is a sensitivity parameter of the controller, and dictates how much we react to the error in our course. The choice of α_p is a delicate one. A small value of α_p may mean that we do not take enough action for a change in course, and our steering process might be overpowered by the winds and sea currents. If α_p is too large, then our steering process will over-react to each course change, and the trajectory of the ship will be jerky. In fact, the system can become unstable if the sensitivity to error is too high [24].

We refer to the type of control process described by Equation 1.1 as a *proportional* controller, because the reaction of the controller is proportional to the error of the process. We can improve our simple controller by considering the history of our system, or by trying to predict where the system will take us in the future. The robotics community has long studied ways of automating such simple control processes [24], and we can draw on their literature for our work in computer animation.

As a first step to improving the controller, we could introduce a term that considers the history of our error. For example, if we have been consistently making too small of course corrections on our ship, then we would like to make larger corrections in the future to compensate. If, on the other hand, we have been over-reacting to errors, then we would like to make smaller adjustments in the future. Thus, if we integrate the error that we have been incurring over the recent past, we can use that integral to determine a future correction.

Also, if we have knowledge of how the system is changing, we can compensate for the change in error. In our ship example, we may know that our error has gone up recently. We if we assume that it will continue to go up without any change in our compensation, then we can increase the compensation to combat the increase in error. Alternatively, if our error is decreasing with the current compensation, then we may want to lower our compensation. Mathematically, we can make an estimate to the derivative of our error, and compensate accordingly.

By combining the proportional error reaction with other reactions based in the error integrals and derivatives, we get what is commonly referred to as a *proportional-integral-derivative* controller, or PID. Just as the proportional control term in Equation 1.1 had a sensitivity parameter α_p, the integral and derivative terms also have sensitivity parameters that we will call α_i and α_d, respectively. Thus, our expression for the reaction of our ship steering PID controller would be

$$\theta_w(t_{i+1}) = -\left(\alpha_p \varepsilon(t_i) + \alpha_i \int_{t_{i-n}}^{t_i} \varepsilon(t)dt + \alpha_d \frac{d\varepsilon(t_i)}{dt}\right) \tag{1.2}$$

Much art and science goes into choosing sensitivity parameters and derivative estimates, as well as integration intervals $[t_{i-n}, t_i]$. An even greater challenge to the animation community is deciding what the goal state of the system should be. For example, to animate a walking human character, we would need goal states for each muscle and joint on the muscular model, and we would have to know states for each of these degrees of freedom in order to control them. Again, the appeal of motion capture techniques becomes evident, as it alleviates the need to determine a function for the goal state.

1.2.6 Sensor-Actuator Networks

In cases where a novel model is to be animated, we will have neither motion capture data, nor will we generally have an intuition on how the model should move. For instance, suppose we want to animate a robotic creature with one motor-driven wheel at its rear and three legs along its front. Such a creature has no analogy in our natural world, and although we could come up with all kinds of hypotheses on how to animate such a creature, we may try dozens of ideas before we find one that vaguely works. Such a task would consume much of the animator's time.

Van de Panne and Fiume [67] give us an alternative means of developing controllers for an articulated skeleton systems by means of process optimization. We can define a number of binary sensors on our articulated skeleton, such as sensors to detect contact with other objects, or that trigger when a joint's orientation is outside of a certain range. We can even have visual sensors that trigger when certain types of objects come into a field of view. The output of the sensors is then used to control *actuators*, which affect some change on the articulated skeleton. In the original work by Van de Panne and Fiume [67], they used two types of actuators: angular actuators that pivoted parts of skeletal tree, and length actuators that actually modified the size of rigid segments of the skeleton.

Once we have a set of sensors and a set of actuators, we can set the output of each sensor to have some weighted affect on each actuator. Each sensor is connected to each actuator, although the weight of its contribution to the actuator's action may be zero. Thus, we have a system of $n_s \times n_a$ weights to be decided, where n_s is the number of sensors and n_a is the number of actuators. For a large number of sensors and actuators, the shear number of weights make the manual tuning of these values tedious and frustrating. Van de Panne and Fiume [67] instead propose a search algorithm which iteratively tries many combinations of sensor weights and determines which combination produces the most pleasing results. In essence, they

perform an optimization of some objective function f to find the largest (or smallest) values of f. This objective function could be a measure of how far the creature moved in a set amount of time, or perhaps the amount of energy needed to achieve movement, or how much damage the creature sustained in its travels, or some combination of metrics. Harsh penalties can be given to parameterizations that cause the creature to fall or collide with other objects, etc. With this type of optimisation, the system can learn modes of motion of which we would perhaps never have thought, and determine which ones produce the most desirable movement characteristics. Van de Panne and Fiume [67] were able to discover many novel and interesting modes of locomotion for even the simplest of creatures. Figure 1.2 shows a number of locomotive modes for a simple model with only two rigid segments.

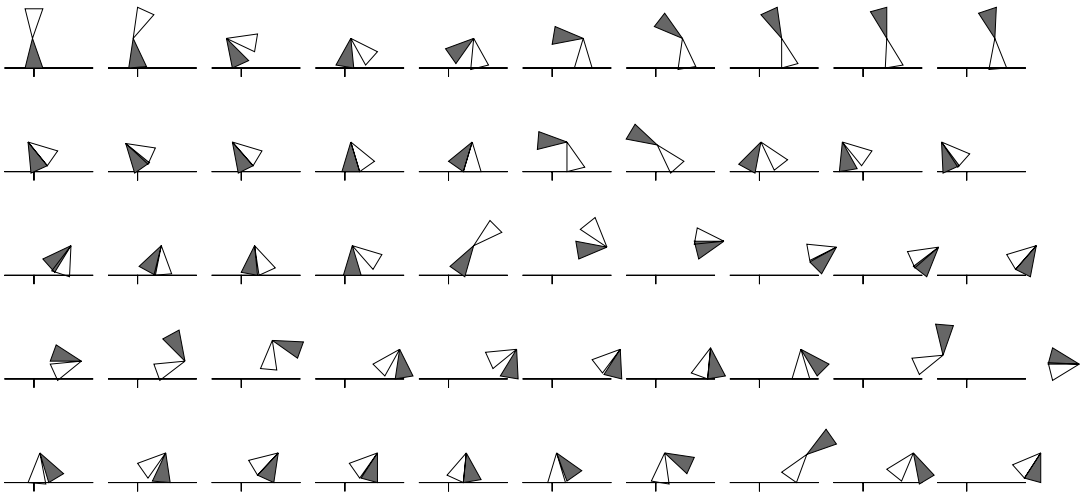

Figure 1.2: Several different modes of locomotion for a simple sensor-actuator network, as given by van de Panne and Fiume [67].

1.3 Goals and Interests

Our interest in jellyfish is with regard to the computer graphics industry. We simply wish to generate animations of the organism that would be convincing to the general public. We also endeavour to gain an understanding of the jellyfish's mode of locomotion. By contrast, we are not as concerned with the complete physical accuracy of our simulations as we would be if we were experimental biologists or applied mathematicians. In those fields, concerns of accuracy and the ability to predict future phenomena are of primary importance. We are not burdened with the need for that level of accuracy. In fact, an extremely accurate biomechanical model of jellyfish would currently be exceedingly difficult. As discussed in Section 1.2, much of the challenges in animation are not in the actual simulation, but in the control side of the system. To date, the scientific community does not have a solid understanding of exactly how jellyfish control their movements, and requires much more research on this matter from the field of experimental biology. Our work is merely an attempt to replicate that movement as best we can with current biological

and physiological knowledge of the creatures.

Specifically, we wish to model the interaction of the jellyfish with its surroundings. To properly capture this interaction, we need a method of simulating an elastic body that is submerged in a fluid. Several methods exist, and we strive for one that is computationally efficient. We wish to have something that can run at interactive rates. Since the model is meant to interact with other objects in the scene, we strive for a real-time animation system that can react quickly to user input.

We endeavour to animate the main physiological aspects of jellyfish that affect its locomotion. We model the jellyfish's umbrella, which comprises the outer hull of the organism. We model the contraction of the umbrella, which is used to achieve jet locomotion. To capture the contraction properly, we aim to model the muscles that line the under-side umbrella, and simulate the contractions of those muscles. We also model the tentacles of the jellyfish, which are attached to the aperture at the bottom of the umbrella. These tentacles generally just drift and do not actively effect locomotion, but do add an element of drag into the physical system. We ignore all other organs of the jellyfish such as its mouth and gonads, because they do not directly affect locomotion in a significant way.

A full 3D simulation of a jellyfish would be expensive. Thus, we search for ways to reduce our computational model. To this end, we limit our interest in jellyfish to those that are axially symmetric. Essentially, these organisms can be approximated by a parabolic curve segment that is rotated about an axis that goes through the peak of the parabola. Many species of jellyfish have this sort of symmetry, although some do not. We aspire to take advantage of this symmetry to reduce the complexity of our model. We simulate a single plane of the jellyfish, which the axis of symmetry bisects, and then extrapolate the results of that plane to a 3D volume. In doing so, we hope to reduce the computational cost of the simulation while maintaining an acceptable level of physical accuracy. However, when extrapolating our simulations to a 3D model, we realize that the 3D version will lack the subtle geometric complexity that is seen in real jellyfish. We aspire to develop a means of adding variation back to the 3D geometry of our model. We make use of biology literature in an attempt to develop a model of what variation a jellyfish will exhibit.

The biology community does have some empirical data on the results of a jellyfish jet propulsion. This data tells us, at a high level, how the organism changes its geometric configuration as it moves, and the resulting thrust that it generates. This kind of data is hard to reproduce exactly because of differences in species of jellyfish, and even differences in individuals within the same species. However, we aim with our model to produce data that is qualitatively similar to that of captured data. In some cases, biological researchers disagree about among the behaviour of the organism. Specifically, the biologists do not agree upon the rate at which the jellyfish contracts and expands to achieve optimal thrust via its jet propulsion. In this case, we run trials with our model to find a contraction frequency that produces the optimal thrust for our simulated jellyfish.

So, without accuracy concerns, we could easily animate jellyfish using less rigorous procedural techniques or key-framing. In fact, similar methods have already been used to animate jellyfish [1, 115, 30].

However, those techniques make difficult the task of interacting the jellyfish model with other objects in their environment, and we do want this interaction between objects. We also want to explore hypotheses of how a jellyfish is able to achieve its locomotion, and we could not do so without some notion of the physical process of jet propulsion. However, techniques to simulate elastic bodies such as jellyfish that are immersed in fluid are very computationally expensive. We also want our model to be as efficient as possible, especially as cinematic scenes involving jellyfish may have many instances of the organism. Thus, we make trade-offs between the interactivity of the animation and its computational cost. The most notable trade-off that we make is the exploitation of the axial symmetry of jellyfish. Their bodies are shaped like ellipsoids, and so we can approximate the organism's 3D geometry with a parabola on a 2D plane, and then rotate that plane about its axis of symmetry. With this sort of symmetry, we simulate the organism as a 2D slice, and then extrapolate those results to 3D. With this model reduction due to symmetry, we also reduce the cost of our simulations.

Because we are simulating a 2D slice of a jellyfish, a naïve approach to reconstruct the 3D geometry would yield an artificially smooth surface model. However, we wish to capture the variation along the umbrella that is seen in real jellyfish. We aim to use biological models of this variation, and can augment them further with a combination of artistic decoration and continuous noise functions that are commonly used in the graphics industry.

With respect to the physiology of jellyfish, the organism has several organs and physical features that are not involved in its locomotion. We ignore these inert features of jellyfish, and concentrate on those features that are important to its swimming behaviour: the outer hull, called the *umbrella*, and the tentacles that line the aperture at the bottom of the umbrella. The umbrella itself is the part of the organism that actively contracts and produces a propulsive jet, and the tentacles merely drift and add drag to the motion of the jellyfish.

We limit our interest in jellyfish to certain types of jellyfish, and certain behaviours by the organism. Because we are trying to exploit the symmetry of the organism, we only consider species that have symmetry to be exploited. We also restrict our attention to jellyfish that are in the adult phase of their life cycle. During this phase, the jellyfish demonstrates its most powerful and sophisticated swimming ability. Even within a single species and stage of development, the jellyfish may exhibit a number of different motion behaviours or modes of locomotion [65]. We limit our interest to what is known to biologists as the *resonant* mode, which is marked by a regular swimming action at a rate that is close to a resonant frequency of the organism [65].

1.4 Thesis Layout

The remainder of this document describes the biology of jellyfish, as well as our model of that biology and how we go about simulating the organism with our model. Since we represent the body of the jellyfish as an elastic volume, Chapter 2 discusses particle-based methods for representing elastic bodies. The chapter

describes how we calculate various forces, especially elastic forces, that will act on the particles within the elastic body in order to govern the motion of those particles. Chapter 2 also derives several numerical methods that we use to integrate the force function, and thus simulate the evolution of the elastic body over time.

To simulate the sea water that surrounds the jellyfish, we make use of existing work in the field of computational fluid dynamics. Chapter 3 introduces the Navier-Stokes equations that describe the flow of an incompressible fluid such as water. Chapter 4 discusses methods of simulating the Navier-Stokes equations using Eulerian approaches, which represent the volume of the fluid with regular grids or static meshes. In particular, this chapter contrasts the classic Eulerian approach that is very accurate yet slow, and the semi-Lagrangian method that is faster, though at the cost of accuracy. Chapter 4 also describes a means of coupling a grid-based fluid representation with a particle-based elastic representation. Chapter 5 introduces an alternative method of modeling fluids that represent the fluid volume with moving particles.

To give a sense of the physiology of jellyfish, Chapter 6 presents research from the biology community. In this chapter, we look at the anatomical aspects of the organism that are relevant to its locomotion, and discuss empirical data and theoretical models of how the jellyfish controls its body to achieve jet propulsion. Chapter 7 then describes our numerical approximation to the biological aspects in Chapter 6, and integrates the numerical techniques in Chapters 2, 4, and 5 to simulate jellyfish with our model. Specifically, Chapter 7 describes how we simulate a 2D slice of the jellyfish with a relatively coarse model, and interpolate that coarse model to obtain a higher resolution 2D rendering model. The chapter then introduces our means of extrapolating the rendering model to 3D, and discusses ways of adding noise to the 3D model to prevent it from looking artificially laminar. Chapter 7 finally describes a simple means of rendering our final 3D model using conventional rendering techniques.

We present the results of our system in Chapter 8. There, we first discuss the results of our simulation methods for elastics and fluids in general. We then show results specifically of our jellyfish model, and compare these results with captured data from biology literature. We also experiment with parameters that control the contraction behaviour of the jellyfish. We then show the results of different stages of our model: simulation, resolution resampling, 3D extrapolation, and noise application. Lastly, we show some final renderings of our model. Throughout this chapter, we give the parameters of our systems, and the values that we used for those parameters.

Chapter 9 concludes the thesis with a review of our model and its results. The chapter also discusses future avenues of research. This future work includes research possibilities for the graphics industry, but also the fields of experimental biology and computational fluid dynamics. From all of these perspectives, we still have much to learn about jellyfish.

CHAPTER 2

ELASTIC BODIES

Since jellyfish are best treated as elastic volumes, this chapter discusses numerical methods for simulating elastic bodies. First, we need a means of representing the volume of the elastic body. We cannot represent the body with a continuous volume, so we instead use discrete point samples, and we denote the position of point i to be $\vec{x}_i$. To add the notion of mass to our discrete volume, we also assign each point a mass value m_i. Thus, it is common to refer to the point as a *point-mass*, although graphics literature often refers to it as a particle. Because the points represent a body in motion, each point will have a velocity vector $\vec{v}_i$.

Since the elastic body will generally be in motion, and the regions of space that it occupies will be changing, we allow the point-masses to move. A discrete system where the particles can move is generally referred to as a *Lagrangian* description of the space. By contrast, we could require that the points be stationary, and we merely simulate the velocities of the body each point. This approach of fixing the sample locations is commonly called an *Eulerian* description.

To model the evolution of our Lagrangian point-masses, we need to derive forces that are acting on them and imposing accelerations on the points. To simulate an elastic body, we model elastic forces between points. The simplest type of interaction between points is by means of springs. We can represent a volume of elastic material by creating a mesh of springs in a finite-elements fashion. This mesh is then used to approximate the forces acting upon the volume of the elastic object. We refer to this combination of point-masses and springs as a spring-mass system.

Once we have an expression for the forces acting on a point-mass, we derive their acceleration using Newton's second law:

$$\vec{a}_i = \frac{\partial^2 \vec{x}_i}{\partial t^2} = \frac{\vec{F}_i}{m_i} \tag{2.1}$$

where m_i is the mass of the Lagrangian point, $\vec{x}_i$ and $\vec{a}_i$ are the point's location and acceleration respectively, and t is time. The vector $\vec{F}_i$ is the sum of all forces acting on the point. We thus have a second-order differential equation to be solved. If our forces are easy to integrate, then we can we can solve the equation analytically as follows:

$$\vec{x}_i = \frac{\int\int \vec{F}_i (\partial t)^2}{m_i} \tag{2.2}$$

When available, these analytical solutions are much preferred, as they allow us to compute a position vector in a constant amount of time. However, if we cannot analytically integrate the forces acting on each point,

then we must turn to numerical integration techniques, which make discrete approximations to the integrals. Numerical techniques are generally more computationally expensive than analytical solutions, although exceptions to this are systems whose analytical solutions involve infinite series or special functions, for example.

The remainder of this chapter describes how we calculate various forces that would act on an elastic body that is being deformed, and also how to numerically simulate those forces over time. Sections 2.1 and 2.2 discuss different types of elastic springs that attempt to constrain certain geometric relationships between points.

2.1 Linear Springs

A linear spring exerts force along the line between two points $\vec{x}_i$ and $\vec{x}_j$ in a way that obeys Hooke's law. In one dimension, Hooke's law is defined as

$$F_i = \kappa_l \Delta x_i, \tag{2.3}$$

where Δx_i is the change of the position of point x_i, relative to the other point x_j. The constant κ_l is the elastic coefficient, often referred to as the "stiffness" of the spring. The elastic coefficient controls how easily the spring can be deformed, and depends on the type of elastic material that is being simulated. Thus, for example, a material such as leather will have a high coefficient because it is hard to stretch, whereas soft rubber will have a relatively small coefficient. As we will see in Section 2.4, the stiffness of the spring also has implications on which numerical method is preferred for our simulation.

A linear spring has a rest length, which is the natural distance between the spring's two end-points. The spring will always induce a force that will push or pull the two end-points so that the distance between them is the spring's rest length. So, if we have a linear spring that connects two points $\vec{x}_i$ and $\vec{x}_j$, then the spring's current length is

$$l_c = |\vec{x}_{ij}| \quad \text{where } \vec{x}_{ij} = \vec{x}_j - \vec{x}_i. \tag{2.4}$$

So, if the spring has a scalar rest length of l_r, then the forces acting on the two points are given by the following vector version of Hooke's law [20]:

$$\vec{F}_i^{Hooke} = \kappa_l (l_r - l_c)(\hat{x}_{ij}), \quad \text{where} \hat{x}_{ij} = \frac{\vec{x}_{ij}}{||\vec{x}_{ij}||}. \tag{2.5}$$

The intuition behind Equation 2.5 is that, if the spring's rest length is less than the current length, the point $\vec{x}_i$ will be pulled toward $\vec{x}_j$, and vice versa. If the current length is less than the rest length, the points will be pushed apart. In either case, the magnitude of the force is proportional to the difference between the rest length and the current length. Figure 2.1 illustrates a Hookean spring in several configurations.

In the above simple case with two points, Hooke's law can be integrated analytically to derive the positions as functions of time. However, when many points are connected with a network of springs, we have to sum up the forces that act on each point. Since the end-points of each spring are moving and thus the

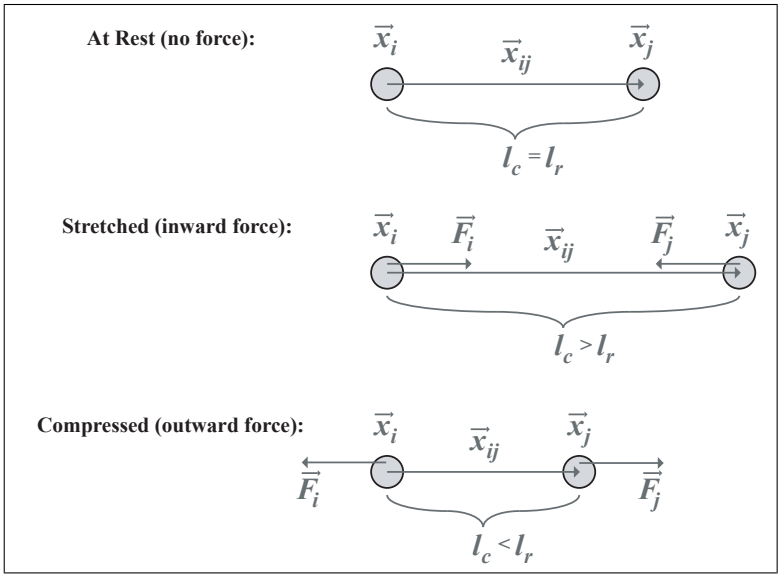

Figure 2.1: The effects of a linear spring. The top image shows the spring at its natural rest length, and thus no forces are acting on the end-points. The middle image shows a spring that is extended past its rest length, and thus the two end-points undergo a force that pulls them inward. In the bottom figure, the spring is compressed below its rest length, and so the spring forces the end-points outward.

axes on which elastic forces act are in constant flux, analytical derivations for the system's forces become extremely difficult. In such cases, it is common to resort to numerical simulation, the details of which will be discussed in Section 2.4.

2.2 Angular Springs

Another group of springs that we consider are angular springs. These springs operate on three points $\vec{x}_i$, $\vec{x}_j$, and $\vec{x}_k$. At any given time, these three points create two line segments $\vec{x}_{ij} = (\vec{x}_j - \vec{x}_i)$ and $\vec{x}_{jk} = (\vec{x}_k - \vec{x}_j)$. Those two lines are oriented at angles with each other. The goal of angular springs is to move the system toward a certain rest angle, just as linear springs move the system toward a rest length. The three points will lie in a plane and move about each other within that plane. We define the surface normal $\vec{n}_{ijk}$ of that plane to be

$$\vec{n}_{ijk} = \vec{x}_{ij} \times \vec{x}_{jk}, \tag{2.6}$$

where the $\times$ symbol denotes the standard cross-product operator for 3D vectors. If the spring has a rest angle θ_r and its current angle is θ_{ijk}, then the torque vector introduced by the spring is

$$\vec{\tau}_i = \kappa_\theta(\theta_{ijk} - \theta_r)\hat{n}_{ijk} \tag{2.7}$$

$$\vec{\tau}_k = \kappa_\theta(\theta_{ijk} - \theta_r)\hat{n}_{jki} \tag{2.8}$$

where again the variable κ_θ is the elastic coefficient constant for the material being simulated. Note that the torque vector is orthogonal to the two lines $\vec{x}_{ij}$ and $\vec{x}_{jk}$. The torque vector's direction represents the

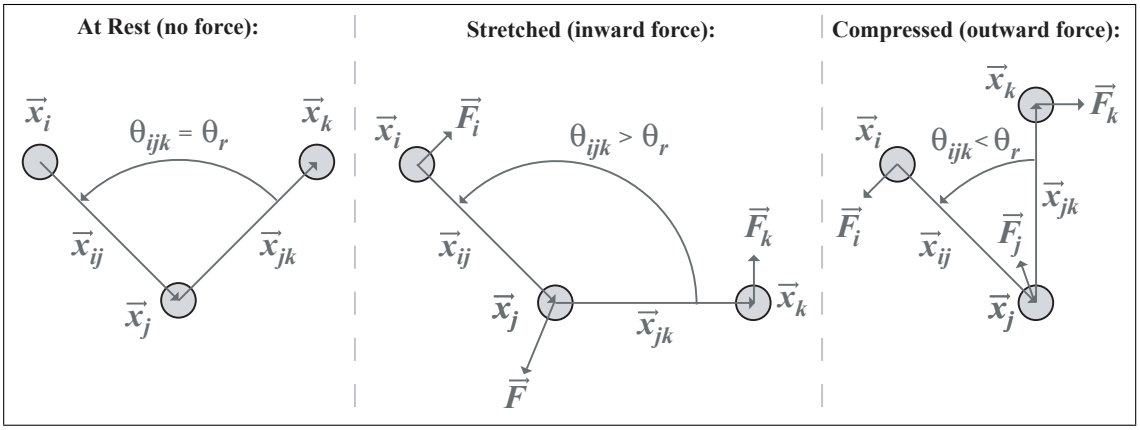

Figure 2.2: The effects of an angular spring. In the left image, the spring is at its natural rest angle, and so no net forces are in play. In the middle image, the spring is opened past its rest angle, and so the two end-points are pulled inwards while the middle point is pushed outward, all to return the spring to its rest angle. In the right-most image, the spring is compressed so that its angle is less than the rest angle, and thus the end-points are pushed apart while the middle point is pulled inward.

axis about which the other two points will rotate, and the torque's magnitude is proportional to the angular acceleration that the points will undergo. To calculate a force from the torque vector, we have to use the following relation:

$$\vec{\tau}_i = \vec{x}_{ij} \times \vec{F}_{i_{angle}}. \tag{2.9}$$

The axis of rotation for the point $\vec{x}_i$ is described by the following parametric line equation:

$$\vec{x} = \vec{x}_j + a\vec{\tau}. \tag{2.10}$$

for all real values of the parameter a. The direction of the force is orthogonal to the torque vector $\vec{\tau}_i$ and the displacement of point $\vec{x}_i$ from $\vec{x}_j$. So,

$$\hat{F}_{i_{angle}} = \frac{\vec{\tau}_i \times \vec{x}_{ij}}{|\vec{\tau}_i \times \vec{x}_{ij}|}. \tag{2.11}$$

We can then find the magnitude of the force by solving for the magnitude of the force the following scalar version of the equation:

$$|\vec{\tau}_i| = |\vec{x}_{ij}||F_{i_{angle}}|. \tag{2.12}$$

Also, we need an expression for the force acting upon point $\vec{x}_j$. By Newton's third law, we know that for each force $\vec{F}_i^{angle}$ and $\vec{F}_k^{angle}$ acting on the neighbours of $\vec{x}_j$, an equal but opposite force also acts on $\vec{x}_j$. Thus, we define

$$\vec{F}_j^{angle} = -(\vec{F}_i^{angle} + \vec{F}_k^{angle}). \tag{2.13}$$

Figure 2.2 illustrates the forces of an angular spring.

To simulate angular springs, we first have to find the angle between two line segments. For our purposes, we are only interested in the 2D case. Thus, we can find the angle θ_{ij} of each line $(\vec{x}_j - \vec{x}_i)$ with respect to

standard polar coordinates such that

$$\hat{x}_{ij} = (cos(\theta_{ij}), sin(\theta_{ij})). \tag{2.14}$$

If the angular spring has a rest angle of θ_r, then we only need to consider angles that are within π (or 180°) of the rest angle. Thus, our system loses physical meaning when the current angle increases to be more than π radians away from the rest angle. For example, our system will confuse angles of 190° with an angle of -170°, because standard inverse trigonometric functions only yield an angular range of 2π radians. However, our system is simple enough that we can easily ensure that we do not approach reflex angles.

2.3 Other Forces

Until now, our system is a purely second-order differential equation that does not mention the first derivative. In other words, the differential equation is of the form

$$\frac{d^2\vec{y}}{dt^2} := f(\vec{y}(t), t). \tag{2.15}$$

We use the position of the particles to calculate their acceleration, and have no direct need to explicitly know a particle's velocity (i.e., the position's derivative with respect to time). In this situation, we would prefer to use a method that deals directly with this type of second-order equations, such as Runge-Kutta-Nyström methods [29, 6]. However, we might want to introduce other parts to the equation that deal with the first derivative, either by directly computing velocities based on position, or by computing a force that relates to the velocity. An example of the latter case is drag force, which is described in the following section. These forces change the differential equation to be of the form

$$\frac{d^2\vec{y}}{dt^2} := f\Big(\frac{d\vec{y}}{dt}, \vec{y}(t), t\Big). \tag{2.16}$$

Although Runge-Kutta-Nyström methods can handle this form of equation, the advantages in terms of computational efficiency are diminished.

2.3.1 Drag Force

Drag is a force that slows an object in motion with an exponential decay profile. This force is sometimes referred to as dampening in elasticity literature. When an object is moving through air (or any fluid for that matter), the air will exert a resistive drag force. This force is proportional to the volume of fluid that will need to be moved so that the object can continue its trajectory. Since that volume depends on the relative speed of the object within the fluid, a simple model of drag force is

$$\vec{F}_i^{drag} = -\kappa_d(\vec{v}_i - \vec{v}_{i_{fluid}}) = -\kappa_d\Big(\frac{\partial\vec{x}_i}{\partial t} - \vec{v}_{i_{fluid}}\Big). \tag{2.17}$$

In Equation 2.17, $\vec{v}_{i_{fluid}}$ is the velocity of the fluid at location $\vec{x}_i$, and this velocity is usually assumed to be zero everywhere in the fluid. The scalar κ_d is a constant referred to as the drag coefficient. Like the elastic

coefficients κ_l and κ_θ in Sections 2.1 and 2.2, this coefficient κ_d controls how easily the volume of fluid can be deformed, and is analogous to the viscosity of the fluid, which will be discussed in Chapter 3.1. However, simulating drag in this fashion is not very physically accurate. The above drag equation is a convenient approximation that we use for testing and experimenting with our system.

2.3.2 Gravity Force

Another force that we simulate in some of our test applications is that of gravity. We make the assumption that our systems are small-scaled enough that gravity is uniform over the entire space of the simulation. Thus, the force acting on a point $\vec{x}_i$ is merely the product of the point's mass m_i and the gravity vector $\vec{g}$ for that environment. For instance, on the Earth's surface, the gravity vector is $\vec{g} \approx (0, -9.8\ m/s^2, 0)$. Our equation for gravitational force is then

$$\vec{F}_i^{gravity} = m_i \vec{g}. \tag{2.18}$$

2.4 Numerical Simulation of Ordinary Differential Equations

Now that we have described all of the forces in our system, we need a means of simulating their effect on the positions of the elastic particles. We can define a force function $\vec{F}(\vec{x}, \vec{v})$ which is the sum of all forces that were considered in the previous sections of this chapter. In doing so, we end up with the following system of first-order differential equations:

$$\frac{d\vec{v}_i(t)}{dt} = \frac{\vec{F}(\vec{x}_i(t), \vec{v}_i(t))}{m_i}, \qquad \frac{d\vec{x}_i(t)}{dt} = \vec{v_i}(t). \tag{2.19}$$

We simulate the above equations with standard integration methods for ordinary differential equations, of which we will give a brief overview in the following sections. For simplicity of discussion, we will assume that the positions $\vec{x}_i$ and velocities $\vec{v}_i$ are combined into a single large vector denoted as $\vec{y}$. Thus, we reduce our equation to the standard first-order differential form:

$$\vec{y}'(t) = f(\vec{y}(t), t), \quad \text{where} \quad \vec{y}' \ \text{denotes} \ \frac{d\vec{y}}{dt}. \tag{2.20}$$

We know the initial value of $\vec{y}(t_0)$ at the initial time t_0. We can find the solution to the system at a later time $t_e = t_0 + \Delta t$ by integrating the function f over the time interval:

$$\vec{y}'(t_e) = \vec{y}'(t_0) + \int_{t_0}^{t_e} f(\vec{y}(t), t)dt. \tag{2.21}$$

We need to integrate the function $f(\vec{y}(t), t)$. If we could compute this integral analytically, we would have a solution to the system. If not, then we need a numerical approximation to the integral, so that we can solve for $\vec{y}(t_e)$ at a later time t_e.

2.4.1 Forward Euler Method

The most naïve numerical integration method is the forward Euler method, which merely assumes that the derivative $\vec{y}(t+\Delta t)$ is constant over some time interval Δt. Thus, we approximate the new values of $\vec{y}$ as follows:

$$\tilde{y}(t+\Delta t) := \vec{y}(t) + \Delta t f(\vec{y}(t), t). \tag{2.22}$$

The above stepping method is called an explicit method because it is a direct calculation of the solution of the system at the next time. The method is also called a first-order method because it uses a first-order estimate of the integral: the derivative is assumed to be constant over the time-step, and so the solution is approximated as a straight line. Figure 2.3 illustrates the forward Euler method, and compares it to other integration methods. The forward Euler method is most prone to error when the derivative $\vec{y}'(t)$ is not constant over the time-step. The accuracy of the approximation greatly depends how quickly the derivative $\vec{y}'(t)$ varies, and on the size of step Δt that we take. So, one way to improve our simulation's accuracy is to decrease the step size, though doing so also increases the computational cost of the simulation. Other methods have been developed that try to make a more accurate approximation to the integral without decreasing the step size.

With specific regard to simulating spring-mass systems, the forward Euler method makes an estimate at the start of the step, which allows the total energy in the system to accumulate artificially. To see why, let us first discuss the two types of energy that are involved in an elastic system. The first type of energy is *potential* energy E_p, which is the energy that is stored in the system. This type of energy is defined as:

$$E_p = -\int_{\vec{x}_0}^{\vec{x}_f} \vec{F} \cdot d\vec{x}. \tag{2.23}$$

The right side of Equation 2.23 represents the work done while the object is moving from point $\vec{x}_0$ to $\vec{x}_f$. For an elastic spring, potential energy can be defined as

$$E_p = \frac{1}{2}\kappa_l ||\vec{x}||^2. \tag{2.24}$$

The second type of energy is *kinetic* energy E_k, which is the energy needed to stop a body in motion, defined as:

$$E_k = \frac{1}{2} m ||\vec{v}||^2. \tag{2.25}$$

In a real physical system, the sum of the two energies $E_p + E_k$ should remain constant at all times. With the forward Euler method, we assume that the derivatives that we calculate at the start of the time-step are constant across the entire step. When the spring's length is moving away from its rest length, the velocity of the points will be greatest at the start of the step, while the elastic force will be lowest at this time. Thus, a forward Euler step will fail to slow each end-point sufficiently, and the magnitude of the spring's oscillation will increase. With each oscillation, the points will travel farther than the previous oscillation, and the potential energy in the system will increase over time. Conversely, when the spring's length is changing toward its rest length, the velocities will be smallest and the elastic forces will be greatest at the start of the

time-step. Since the forces are increasing the speed of the end-points, the forces that are over-estimated by the forward Euler step will introduce a positive error in the kinetic energy of the system. In both cases, we are accumulating energy erroneously, and the system will become unstable. Larger κ values will cause the derivative to vary more quickly, and intensify the problem. Of course, we could take smaller steps to get a better estimate of the integral. However, the size of the steps we would need to take would make simulation prohibitively expensive. The next few sections briefly describe more sophisticated stepping methods that generate more accurate estimates.

2.4.2 Predictor-Corrector Method

With the forward Euler method, we made one estimate for the solution at the end of a time-step. We could further refine that estimate and iteratively converge on another solution to the system. For example, suppose we use the forward Euler method above to get the following initial guess:

$$\tilde{y}_0 := \vec{y}(t) + \Delta t f(\vec{y}(t), t). \tag{2.26}$$

We then iteratively refine the estimate by plugging the new value $\tilde{y}_n(t+\Delta t)$ back into the integration formula

$$\tilde{y}_{n+1} := \vec{y}(t) + \Delta t f(\tilde{y}_n, t+\Delta t). \tag{2.27}$$

Essentially, we are approximating the derivative at the end of the step, rather than directly computing the derivative at the beginning of the step as we did with the forward Euler method. Even though we are still dealing with an approximation, the correction stage of the integrator will bring the estimate back toward the true solution, and reduce the over-estimation. In the case of simulating elastic springs, a predictor-corrector method will reduce the amount of energy in the system, rather than allow it to grow. The energy reduction is because the derivatives used in the linear approximation come from the end of the time-step, rather than the start as they did in the forward Euler method. Thus, rather than the simulation becoming unstable, it now undergoes numerical dampening.

We can iterate the above approximation to some convergence tolerance if we like. However, the most significant gains are made on the first iteration. Thus, it is usually considered optimal to only iterate once. However, no matter the number of iterations, the fact remains that we are assuming a derivative that is constant across the time-step. Consequently, this type of method also has first-order accuracy.

2.4.3 Leap-frog Method

In the case of a particle system, we really have a second-order differential equation of the form:

$$\frac{\partial^2 \vec{y}(t)}{\partial t^2} = f(\vec{y}(t), t), \tag{2.28}$$

that we can convert to a system of first-order equations that have the following form:

$$\vec{y}_1'(t) = f_1(\vec{y}_1(t), \vec{y}_2(t), t), \tag{2.29}$$

$$\vec{y}_2'(t) = f_2(\vec{y}_1(t), \vec{y}_2(t), t) = \vec{y}_1(t). \tag{2.30}$$

Each equation in the system represents a different derivative (either the first or the second). In some cases, we can get better estimates of the solution by integrating each equation on a different time scale. A particular method that uses this approach is the so-called *leap-frog* method. Mathematically, the method could be described as follows:

$$\tilde{y}_1(t+\frac{\Delta t}{2}) = \vec{y}_1(t-\frac{\Delta t}{2}) + \Delta t f_1(\vec{y}_1(t), \vec{y}_2(t), t), \tag{2.31}$$

$$\tilde{y}_2(t+\Delta t) = \vec{y}_2(t) + \Delta t f_2(\vec{y}_1(t+\frac{\Delta t}{2}), \vec{y}_2(t), t+\frac{\Delta t}{2}). \tag{2.32}$$

The idea behind Equations 2.31 and 2.32 is that we take half-steps, but on each half-step we only integrate one half of the system. Thus, in a sense, we get twice the resolution in the time axis, but with the same number of integration steps.

2.4.4 Backward Euler Method

Implicit methods are preferred for spring-mass simulations, especially when the springs have large values for their elastic coefficients κ_l and κ_θ that were described in Sections 2.1 and 2.2. Indeed the computer graphics community prefers implicit methods for simulating spring-mass systems [113, 69, 51, 15, 108]. However, when spring forces are small or dampening forces such as drag are large, the advantages of implicit methods fade away.

The simplest implicit method is the backward Euler method. Instead of explicitly calculating the derivative at the beginning of the time-step as we did with the forward Euler method, we instead attempt to use the derivative at the end of the time-step, as follows:

$$\tilde{y}(t+\Delta t) := \vec{y}(t) + \Delta t f(\vec{y}(t+\Delta t), t). \tag{2.33}$$

Since $\tilde{y}(t+\Delta t)$ appears on both sides of the equation, the Equation 2.33 is a non-linear system in which we must solve for $\vec{y}_{t+\Delta t}$ using numerical techniques. Several methods could be used to solve the system, including *Newton's method* and *fixed-point iteration* [96]. No such method is perfectly robust. We give a brief description of Newton's method here, since it is commonly used with the backward Euler method. To solve an equation of the form

$$\vec{y} = g(\vec{y}), \tag{2.34}$$

or rather

$$\bar{g}(\vec{y}) = g(\vec{y}) - \vec{y} = 0, \tag{2.35}$$

Suppose we have a guess $\tilde{y}_i$ at the true value of $\vec{y}$. Assuming that we can compute derivative $\frac{d\bar{g}(\vec{y})}{d\vec{y}}$, then we can make a new estimate $\tilde{y}_{i+1}$ as follows:

$$\tilde{y}_{i+1} = \tilde{y}_i - \left(\frac{d\bar{g}(\vec{y})}{d\vec{y}}\right)^{-1} \bar{g}(\vec{y}). \tag{2.36}$$

The derivative $\frac{d\bar{g}(\vec{y})}{d\vec{y}}$ in Equation 2.36 really a matrix known as the *Jacobian* matrix. If this matrix is close to zero, then its inverse will be large, and the estimated solution y_{i+1} will diverge rather than converge.

Newton's method also has other requirements on the function $\bar{g}(\vec{y})$. First, the method assumes that our initial guess $\tilde{y}_o$ is relatively close to true root of the equation. Under this assumption, the function's derivative will more reliably point in the direction of the root. Second, Newton's method requires that the derivative of the function is continuous.

Another possible issue with Newton's method is its computational cost. Ideally, we would have an analytical way of computing the Jacobian matrices directly. However, if that is not the case, then we would have to generate the matrix numerically. For large systems of differential equations, the Jacobian matrix can be quite large, and numerical approximations thereof can be prohibitively expensive. We have not attempted to implement Newton's method for our simulations because we have not been able to find a reliable and efficient means computing the Jacobian matrices.

2.4.5 Higher-Order Methods

We can also improve the integration scheme by recognizing that the derivative can change over the course of our step. Runge-Kutta methods [28] take several samples along the step and estimate the derivative as a linear combination of these samples taken over the time-step. Samples are taken in sequence, and the information from the previous samples is used to estimate the next sample. As an example, imagine taking a small step with the forward Euler method. We first compute the derivative at the start of the time-step:

$$\vec{y}_1'(t) := f(\vec{y}(t), t). \tag{2.37}$$

We then use that derivative to estimate the integral at some point c_1 along the length of the time-step:

$$\vec{y}_1(t + c_1\Delta t) := \vec{y}(t) + (c_1\Delta t)\vec{y}_1'(t). \tag{2.38}$$

We could use this new value $\vec{y}_1(t + c_1\Delta t)$ to get an estimate of the derivative at location c_1 along the time-step:

$$\vec{y}_2'(t + c_1\Delta t) := f(\vec{y}_1(t), t + c_1\Delta t). \tag{2.39}$$

We could then use some combination of the two derivative estimates $\vec{y}_1(t + c_1\Delta t)$ and $\vec{y}_2'(t + c_1\Delta t)$ to make another estimate to the solution at some later time:

$$\vec{y}_2(t + \Delta t) := \vec{y}(t) + \Delta t(a_1\vec{y}_1' + a_2\vec{y}_2'), \tag{2.40}$$

where a_1 and a_2 are weights given to each estimate. If we choose $a_1 = a_2 = 0.5$ and $c_1 = 1$, then we obtain the common second order Runge-Kutta method known as the *trapeziodal rule* [38].

We could keep generating samples in the above manner, and then take a linear combination of them to get our estimate of the integral. If the samples and their weights are chosen carefully, we can generate methods that represent high-order estimates of the integral. Figure 2.3 illustrates the second-order Runge-Kutta (RK2) method. In our work, we make use of the standard RK2, RK4, and Dormand-Prince methods,

which are second-order, fourth-order, and fifth-order methods, respectively. The details of these methods are given by Dormand and Prince [28], as well as by any elementary book on numerical analysis [38].

Finally, we could also get high-order implicit methods. We have not tried to implement any of these methods, but discuss them briefly here. Some higher-order implicit methods are part of the Runge-Kutta family of methods. Others are so-called multi-step methods that use solutions from previous steps to estimate future solutions. Since we have a history of values $\vec{y}(t_0)$ to $\vec{y}(t_i)$ and their respective derivatives, we can estimate $\vec{y}(t_{i+1})$ by fitting a high-order curve through the constraints of the values and the derivatives. We can then extrapolate into the current time-step in order to get the next value of the function. Doing so is referred to as an *Adams-Moulton* method. This sort of method has many implementation subtleties, and we refer the reader to Glawdell et al. [38] for details.

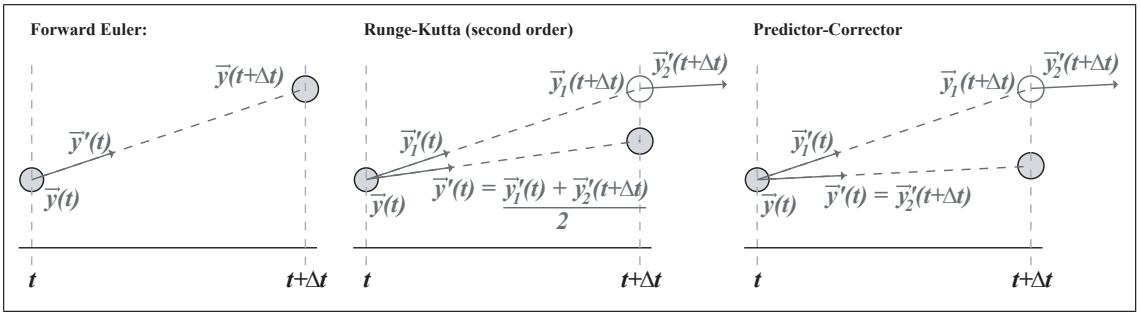

Figure 2.3: Three different types of numerical methods. The left-most diagram illustrates the forward Euler method, a first-order explicit method. It merely calculates the derivative at the start of the time-step, and then approximates the function's derivative by projecting that starting derivative across an entire time-step. The middle diagram shows a second-order Runge-Kutta method, which is a higher-order explicit method. It computes a forward Euler step, and then recalculates the derivative at the end of the step. This second-order method uses the average of the two derivatives to then approximate the integral over the entire time-step. Lastly, the right-most illustration shows a predictor-corrector method that is a first-order explicit method. It also takes a forward Euler step and computes the derivative at the end of the step, but then integrates the entire step with this end derivative.

2.4.6 Time-Stepping and Error Estimation

As a final consideration, we would like a means of determining how large of a time-step to take. We can estimate the error incurred by a particular step size by comparing two different estimates of the integral. A simple, but relatively expensive way to obtain a second estimate would be to take two steps, each with half the original step size. We could then compare the results of the two smaller steps with the larger one. The two smaller steps are assumed to be a more accurate estimate of the integral. However, this step-doubling error metric requires three times as many integration calculations as a single step.

Since Runge-Kutta methods have a number of estimates to the integral embedded in them, we can often construct two different methods merely by placing different weights on the intermediate intermediate stage values. Each method will have a different order of accuracy. With a pair of embedded methods, we can

get our two estimates of the integral, all for the minor cost of recombining the stage values. Thus, when embedded pairs are available, they are a much preferred error metric. Error estimation is described in more detail by Gladwell et al. [38].

2.4.7 Numerical Stability and Stiffness

With regard to numerical integration, *stability* and *stiffness* are issues that govern the step sizes that we can take in our simulations. The stability of a simulation depends on the system being simulated, but also the numerical method and the size of the time-steps being used. A simulation is considered stiff when the time-steps are restricted by stability rather than accuracy. In many cases, an implicit integration method makes the simulation more stable than an explicit one. In fact, in Section 4.2, we will discuss an implicit method of simulating fluids that is unconditionally stable. A simulation is considered stiff if an implicit method allows for a more efficient simulation than an explicit one. In our case of spring-mass systems, high values for the elastic coefficients κ_l and κ_θ make the simulation more stiff. When the elastic coefficients are large, the magnitude of the elastic forces will, on average, be larger at the start of a time-step then they will be at other times within the time-step, and the estimated integral will be over-estimated. Although the use of high-order implicit methods can attenuate the over-estimation, implicit methods can solve for an estimate that is not as poorly estimated. Thus, erroneous force magnitudes will not accumulate as quickly, and the simulation will be more stable.

2.5 Previous Applications

Spring-mass systems have been widely discussed in graphics literature. These types of systems were first used to simulate passive objects like cloth [51, 15] and hair [10, 52]. Wu and Popović [117] modeled the motion of bird feathers with a Lagrangian system. More general research into elastic bodies has been conducted [108, 83]. The graphics community has gone so far as to even model tearing and other geometric changes due to elastic deformation [71]. A wide range of optimizations have been made that spatially group complex objects into more simple models [78, 79]. These techniques then compute the deformation on the low-resolution simulation model, and interpolate the deformation onto the higher resolution rendering model.

Spring-mass models have also been used to simulate active objects. Miller [69] used a simple spring-mass system to animate worms and snakes. Tu and Terzopoulos [113] animate the motion of fish using an elastic body simulation. This last work is a curious diversion from other elastic body simulations. Since fish are vertebrates, the conventional wisdom in the graphics community would be to simulate the organism as an articulated skeleton [92]. Pai et al. [82] have also used a combination of spring-mass and rigid body models for simulating the human musculoskeletal system. Lee and Terzopoulos [57] developed a similar system for the human head and neck, with mostly elastic forces used to animate complex facial expressions.

In all of these cases with active elastic objects, motion is achieved via a contraction of the elastic volume. We can simulate this by dynamically changing the rest lengths l_r and rest angles θ_r of the springs, and also by adjusting the elastic coefficients κ_l and κ_θ of the springs.

In our work with jellyfish, we also simulate an active elastic organism. Thus, we build a spring-mass system to represent the flesh of the jellyfish, and modify the rest lengths of the springs in regions where muscle density is high. However, to account for the sea water that surrounds the organism, we do not merely simulate the spring-mass system alone. Instead, we incorporate a full fluid simulation, and model the interaction between the two media. Chapter 3 will cover the theoretical aspects of fluid dynamics. Chapters 4 and 5 will describe two main classes of fluid simulators, and how they can be integrated with a spring-mass system. Chapter 7 then describes our specific model of jellyfish.

CHAPTER 3

THEORETICAL FLUID MECHANICS

To account for the sea water that surrounds the jellyfish, we need a means of simulating fluids. Several mathematical models exist, each with a variety of numerical methods. We chose to use a model based on the famous Navier-Stokes equations, because of their ability to model a wide variety of natural phenomena. These equations describe how the flow of a fluid will evolve over time, given the properties of the fluid (such as the viscosity and density), external forces acting on the fluid, and any external constraints that we may want to place on the flow. This chapter discusses the Navier-Stokes equations, and subsequent chapters discuss numerical methods for simulating these equations.

We have a few different choices of the style in which we simulate fluids. We look at two broad categories of simulators, which differ mainly in the way in which they represent space within the fluid volume. First, we explore mesh-based fluid simulators, which give us velocity values at fixed points in space. Velocities at other locations must be interpolated from the fixed locations. As we saw in Chapter 2, the elastic bodies are represented by moving points in space. In order to model the interaction between the two bodies, we need a method that will simulate the evolution of the velocity field of the fluid so that we know how the elastic particles will be moved by the fluid. We also need a means of introducing acceleration back onto the fluid grid so that the spring forces of the elastic blob will properly affect the fluid's flow field. Other techniques of modeling fluid-solid interactions also require the knowledge of the pressure field across the fluid, so that objects can be pushed from regions of low pressure to neighbouring regions of high pressure. Chapter 4 will discuss mesh-based methods in more detail.

As an alternative to a mesh-based simulator, we could also use a particle system to model the motion of the fluids. With this method, velocities are attached to moving points in space, and the velocity field at any arbitrary location is a weighted sum of the particle velocities within some local neighbourhood. Again, we get a velocity field from this particle-based method. However, we can directly model the interaction between the fluid and the elastic object by simulating collisions between the solid object and the fluid particles. Chapter 5 discusses particle-based methods in more detail.

The incompressible Navier-Stokes equations describe the evolution of a fluid's velocity field over time. The equations have two parts. One equation accounts for the conservation of momentum in the system. Another equation ensures that the fluid is incompressible (i.e., the fluid's density is uniform over space and time, and the volume of fluid does not change). In other words, when the fluid is under pressure, the

change in the fluid's volume is zero (or negligible). Water is extremely hard to compress. At a pressure of 1 lb/in^2 (6.89 kPa) water at 20°C will compress by a factor of 0.0000034. Thus, we use the incompressible Navier-Stokes equations to simulate liquids such as sea water.

The velocity field that we get from the equations dictate how objects in the fluid will be carried by the flow of the fluid. The equations also have a term for any external forces acting on the fluid. Thus, objects in the fluid can exert forces on the fluid and affect the flow field.

3.1 Momentum Equation

The first part of the Navier-Stokes equations is the momentum equation. This equation is defined as follows:

$$\frac{\partial \vec{u}}{\partial t} = \frac{-\vec{u}\cdot\nabla\vec{u} - \nabla p + \nu\nabla^2\vec{u} + \vec{F}}{\rho}. \tag{3.1}$$

where $\vec{u}$ is a vector field that represents the fluid's velocity, p is a scalar field for pressure, ρ is the density of the fluid, ν is the coefficient of kinetic viscosity, and $\vec{F}$ is a vector field for external forces that are acting on the fluid. The symbols ∇ and ∇^2 are the standard calculus operators for the spatial gradient and Laplacian, respectively. Since the density of water is close to $\rho = 1\ kg/L$, varying with temperature by a few percent in its liquid phase, the factor ρ is often dropped from the equation.

Equation 3.1 tells us how the flow field will change in the future. The first term of the above equation accounts for advection of the fluid flow, and essentially says that the flow field's future velocity value will be affected by whatever is behind it (with respect to the current flow direction). The second term of the momentum equation accounts for forces that arise because of differences in pressure, and states that the flow will tend from regions of high pressure toward regions of low pressure. The third term accounts for the viscosity (thickness) of the fluid, and that the flow in one area will tend to coincide with the flow of adjacent regions, at least to some degree. The last term handles any outside forces that are acting on the fluid to induce an acceleration in the flow, such as gravity or wind or, as we shall see, forces from solid objects. A complete derivation of the Navier-Stokes equation is given by Griebel et al. [40].

3.2 Mass Equation

The second equation in the system enforces incompressibility by ensuring that the density of the fluid is constant, and that mass is conserved. In 3D, this equation is:

$$\nabla\cdot\vec{u} = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z} = 0, \qquad \text{where } \vec{u} = (u_x, u_y, u_z). \tag{3.2}$$

Essentially, the above equation ensures that if a certain volume of fluid leaves a region, that volume is replaced by the same volume from another region. In practice, incompressibility is enforced in a few different ways. The *divergence operator* $\nabla\cdot$ measures how much the vector field acts like a sink or a source of fluid flow. This operator is really the composition of the gradient (∇) and the vector dot-product ($\cdot$). In fact, the

Laplacian operator ∇^2 in Equation 3.1 is really the divergence of the gradient $\nabla \cdot \nabla$. Since Equation 3.2 ensures that the system has no such sink or source, we call the flow *divergence-free*.

3.3 Poisson Equation for Pressure

At first glance, Equation 3.1 looks like a partial differential equation that we could discretize and solve using standard solvers for ordinary differential equations. However the second term involving the pressure gradient is a complex one. We do not have an explicit formula for the pressure, but rather the following implicit one:

$$\nabla^2 p = \frac{\nabla \cdot \vec{G}(\vec{u})}{\partial t}, \tag{3.3}$$

where

$$\vec{G}(\vec{u}) = \vec{u}(t) + \int \Big(\frac{\partial \vec{u}}{\partial t} + \frac{\nabla p}{\rho} \Big) dt = \vec{u}(t) + \int \Big(-\vec{u} \cdot \nabla \vec{u} + \nu \nabla^2 \vec{u} + \vec{f} \Big) dt. \tag{3.4}$$

In Equation 3.3, G is a function of the velocity field $\vec{u}$. In our case, it is the calculation of the velocity field's future value, but without any notion of the pressure term. Equation 3.3 is a type of relation known as a Poisson equation. Thus, the pressure term presents a problem because the pressure values implicitly depend on the velocity field, yet the velocities depend explicitly on the pressure. How we solve this Poisson equation depends on the way we simulate the entire fluid system. The following chapters describe several simulation methods that we tried in the course of our research. Each method has its own way of representing the fluid volume with discrete points, and then solving approximate versions of the Navier-Stokes equations over that discrete space.

CHAPTER 4

GRID-BASED FLUID SIMULATION

Classical methods of simulating the Navier-Stokes equations stem from physics and engineering communities [40]. These methods use an *Eulerian* spatial representation, which treats space as a static mesh (usually a grid). At each node in the mesh, we store a velocity value. We then simulate the Navier-Stokes equations numerically in order to model the evolution of the velocities at the mesh nodes. Such mesh-based approaches are most effective at representing incompressible fluids.

This chapter discusses two different ways of simulating the Navier-Stokes equations over meshes. The first method, which is a classical Eulerian method, comes from literature in the mathematics and engineering communities. This classical Eulerian approach is relatively slow, yet extremely accurate. As an alternative, we also present a semi-Lagrangian method that was developed by Stam [102] specifically for the graphics community. Stam's method is more stable, and generally less computationally expensive, though at the cost of physical accuracy.

No matter which grid-based method we choose to simulate the motion of fluids, we end up with a velocity field that describes the fluid's flow. Thus, we can use the velocities to determine how the flow will push any solid objects that are in the fluid. Conversely, we can have the objects exert force back onto the fluid, changing the flow field. Section 4.3 describes several ways to model the interaction between grid-based fluids and the particle-based elastic models that were described in Chapter 2.

4.1 Classical Eulerian Method

We now describe classical methods of simulating the Navier-Stokes equations over a grid-based spatial discretization. These methods are used by the scientific community to obtain extremely accurate simulation results, without concern for performance issues. Later, in Section 4.2, we will see an alternative method whose purpose is to accelerate the simulation, though at the cost of physical accuracy.

4.1.1 Finite Differences Approximation of Spatial Derivatives

In the simplest case, the Eulerian method represents space as a uniform grid. Each cell of the grid stores the velocity vector at that location in space. Since we also need to know the pressure gradient to calculate fluid accelerations, we store a pressure value at each cell as well. So, if $\vec{u}_{i,j}$ is a velocity vector at the cell in the

i^{th} column and the j^{th} row of the grid, then we could approximate derivatives of the velocity at that grid cell (i, j) with the so-called central difference:

$$\frac{\partial \vec{u}_{i,j}}{\partial x} \approx \frac{\vec{u}_{i+1,j} - \vec{u}_{i-1,j}}{2\Delta x}. \tag{4.1}$$

Notice that the above equation makes no mention of the mid point $\vec{u}_{i,j}$ itself. That point could have any arbitrary value, and the computed derivative would not be affected. The use of central differences with an Eulerian spatial discretization is known to be unstable [40, 8]. One possible solution would be to use a one-sided difference, like the following:

$$\frac{\partial \vec{u}_{i,j}}{\partial x} \approx \frac{\vec{u}_{i,j} - \vec{u}_{i-1,j}}{\Delta x}. \tag{4.2}$$

However, the above difference will introduce a bias into the system that can become apparent in the results. One could also attenuate these biases by increasing the spatial resolution of the grid, but instabilities could still remain. Instead, Harlow and Welch [47] introduced the concept of a staggered grid, which places velocity values at the edges of the grid cells, and keeps the pressure in the center of the cell.

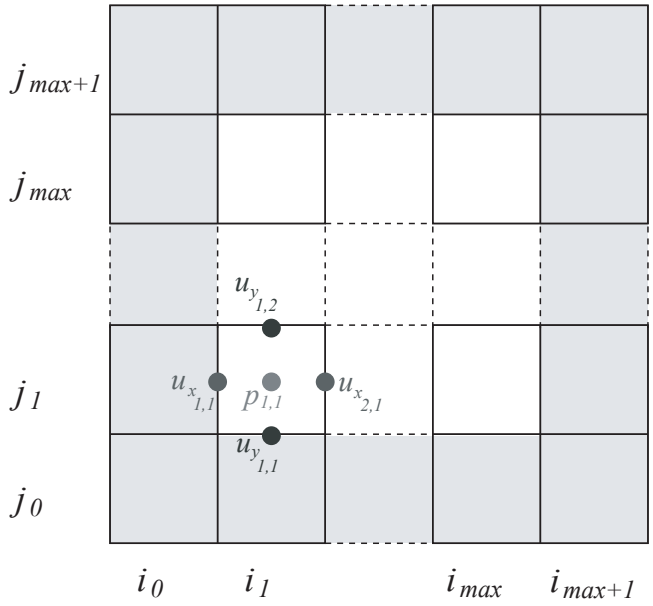

Figure 4.1: Harlow and Welch's staggered grid [47]

.

Figure 4.1 shows the spatial configuration of our staggered grid. With this configuration, we can get more robust approximations of the spatial derivatives. For example, if we want to know the horizontal velocity at the pressure location in the center of cell (i, j), we can compute that pressure value as

$$\frac{\partial \vec{u}_{x_{i,j}}}{\partial x} \approx \frac{\vec{u}_{x_{i+1,j}} - \vec{u}_{x_{i,j}}}{\Delta x}. \tag{4.3}$$

Similarly, the pressure derivatives at the boundaries of the cell can be approximated as follows:

$$\frac{\partial p_{x_{i,j}}}{\partial x} \approx \frac{p_{i,j} - p_{i-1,j}}{\Delta x}, \qquad \frac{\partial p_{y_{i,j}}}{\partial y} \approx \frac{p_{i,j} - p_{i,j-1}}{\Delta y}. \tag{4.4}$$

4.1.2 Time-Stepping

Because of complications in the pressure term of the Navier-Stokes momentum equation, we use a so-called "splitting" [8] approach to solving the equation. Suppose we have a differential equation that consists of two terms, as follows:

$$\frac{\partial \vec{y}}{\partial t} = f(\vec{y}) + g(\vec{y}). \tag{4.5}$$

We can derive a time-stepping method that treats each term of the differential equation in turn. A first-order explicit method that involves splitting would be the following:

$$\tilde{y}_f := \Delta t f(\vec{y}(t)).$$

$$\vec{y}(t+\Delta t) := \tilde{y}_f + \Delta t g(\tilde{y}_f).$$

In our case, we want to break off the pressure term from Equation 3.1 by computing a forward Euler step without any regard for that pressure term, as was briefly explained in Section 3.3. The discrete version of our function $\vec{G}$ is:

$$\vec{G}(t) := \vec{u}(t) + \Delta t[-\vec{u}\cdot\nabla\vec{u} + \nu\nabla^2\vec{u} + \vec{f}]. \tag{4.6}$$

With the above intermediate estimate to the velocity field, we can then solve for the pressure field, and return the system to a divergence-free state. The means of calculating the pressure field is more thoroughly explained in Section 4.1.4.

The last stage of the time-step is to "fix" our velocity estimates with our pressure values, as follows:

$$\vec{u}(t+\Delta t) := \vec{G}(t) - \Delta t\Big(\frac{\nabla p}{\rho}\Big). \tag{4.7}$$

Finally, we have to consider how large of time-step can be taken. Several stability conditions govern the size of our time-steps. The first criterion is the standard Courant-Friedrichs-Lewy condition [88], which states that we must not allow advection to travel farther than one grid spacing (in each dimension). Mathematically, the condition is:

$$\Delta t < \frac{\Delta x}{max(u_{x_{i,j}})} \tag{4.8}$$

and similar for the y and z dimensions, etc. The final criterion is to account for instability that is incurred by fluids with high viscosity. Intuitively, if the fluid has higher viscosity, then velocities in one region of the fluid will get distributed onto adjacent regions. Thus, the advection approximation that we make will be less accurate, as the velocity field will become more diffuse. Griebel et al. [40] give us the following stability condition to take viscosity into account:

$$\Delta t < \frac{1}{2\nu}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-1} \tag{4.9}$$

If we adhere to all of the above criteria, we will prevent the simulation from becoming unstable. However, note that the stability criteria do not explicitly control the error of the simulation.

4.1.3 Spatial Boundary Conditions

Our finite differences method described in Section 4.1.1 requires us to have approximations for derivatives everywhere, but values at the edges of our grid will need special consideration. Instead of computing these boundary values with the Navier-Stokes equations, we set the velocities and pressures at the boundary explicitly. The conditions at the grid boundaries can take several forms, depending on the simulation. We treat these conditions at the boundary differently, depending on what is happening at the boundary. The boundary could be a solid wall, either stationary or moving, in which case the frictional properties of the wall will dictate the way we handle the boundary conditions. If there is no solid object at the boundary, then the fluid could be allowed to flow freely, or it could have a flow imposed upon it by some external forces.

We follow the work of Griebel et al. [40] in our treatment of boundary conditions. Their work focuses mostly on non-moving rigid boundaries, but their methods are easily adapted to moving objects [8]. Griebel et al. [40] identify four types of boundary conditions, which fall into two categories of either solid boundaries or flowing boundaries. For solid boundaries, no flow should go through the boundary. Note that these solid boundaries could represent the surface of a moving object, as discussed in Section 4.3.1. The solid boundary might be a *no-slip* type, in that the boundary has frictional characteristics which force the flow at boundary grid cells to have the same velocity as the boundary itself. Alternatively, a solid boundary could be a *free-slip* type, which forces the flow to have the same velocity as the boundary in the normal direction of the boundary surface. However, free-slip boundaries do not modify the flow that is tangential to the surface. The free-slip type of boundary models a frictionless surface that prevents flow from going through the boundary surface, but freely allows flow to go along the surface.

For experimentation and testing purposes, we also use flowing boundary types that represent openings in the fluid region. The simplest of these other boundary types is the *free-flow* boundary, which makes no modifications to the flow. Free-flow boundaries are used to model openings that are completely unrestricted. Numerically, we model this type of boundary by copying the values to the boundary grid cells from their immediate neighbours. By contrast, *in-flow* boundaries model jets and other forced flows. In-flow boundaries dictate directly what the flow vectors are at their locations. In-flow conditions are actually implemented in the same manner as no-slip conditions. The difference in terminology exists because no-slip conditions are used for solid objects that could be moving, and in-flow conditions are used to model a forced stream on an open boundary. Lastly, a boundary can be *periodic*, which is used to represent a cyclic configuration. For example, a long pipe with regular notches along it can be simulated as a single short segment with one notch, and periodic boundary conditions. Numerically, we simulate periodic boundaries by copying velocity values from the other side of the grid.

Figure 4.2 summarizes the different types of boundaries that we consider, and the mathematical definitions thereof. Note that the table has two definitions of the boundary values. The *normal grid* definition is how we would modify the velocity grid if we were to use a normal (non-staggered) grid. However, with a *staggered grid* as depicted in Figure 4.1, we have to set the staggered velocity values so that the interpolated

value at the cell centres will be equal to our desired boundary value. Thus, the table also gives a definition of the boundary condition that is to be used with a staggered grid.

Type	Description	Normal Grid Definition	Staggered Grid Definition
No-slip	No flow at boundary.	$\vec{u}_{i,j} := \vec{u}_b$	$u_{x_{i,j}} = u_{b_x} - u_{x_{i+1,j}},\ \ u_{y_{i,j}} := u_{b_y}$
Free-slip	No flow through boundary.	$\vec{u}^{\perp}_{i,j} := \vec{u}^{\perp}_b,\ \ \vec{u}^{\|\|}_{i,j} := \vec{u}^{\|\|}_{i+1,j}$	$u_{x_{i,j}} := u_{b_x} - u_{x_{i+1,j}},\ \ u_{y_{i,j}} := u_{y_{i+1,j}}$
Free-flow	No restriction on flow.	$\vec{u}_{i,j} := \vec{u}_{i+1,j}$	$u_{x_{i,j}} := u_{b_x} - u_{x_{i+1,j}},\ \ u_{y_{i,j}} := u_{y_{i+1,j}}$
In-flow	Dictated flow velocity.	$\vec{u}_{i,j} := \vec{u}_b$	$u_{x_{i,j}} := u_{b_x} - u_{x_{i+1,j}},\ \ u_{y_{i,j}} := u_{b_y}$
Periodic	Cyclical flow.	$\vec{u}_{i,j} := \vec{u}_{i_{max}-1,j}$	$u_{x_{i,j}} := u_{x_{i_{max}-1,j}},\ \ u_{y_{i,j}} := u_{y_{i_{max}-1,j}}$

Figure 4.2: Boundary conditions for finite differences simulations. As an example, this table shows how one would compute the boundary values with respect to a west-ward boundary at the pressure location $p_{i,j}$. The symbols $\vec{u}^{||}$ and $\vec{u}^{\perp}$ refer to the components of the fluid's velocity that are respectively parallel (tangential) and perpendicular (orthogonal) to the surface of the boundary. Also, $\vec{u}_b$ represents the velocity of the boundary itself (in the solid boundary cases), or the velocity of the inward flow (for the open boundary cases).

The following section talks about how we deal with the boundaries of the pressure field. Essentially, we assume that all solid objects have a constant pressure (usually 0), and so we set the pressure at the boundaries to 0.

4.1.4 Pressure Equation

Several methods for solving the Poisson equation for pressure are discussed by Bridson et al. [8]. We chose to use a successive over-relaxation (SOR) method, as described by Griebel et al. [40]. Successive over-relaxation is an iterative method that makes the pressure grid converge toward a stable configuration. At each iteration, we estimate a residual value that is proportional to the divergence of the current system configuration. We then allow each cell of the pressure grid to absorb some amount of that residual pressure from its neighbouring cells.

The first step in the SOR method is to identify edges of the pressure grid. The pressure values at the edges of the grid are assumed to be 0. So, we assign those edge values to have no weight in our pressure calculations. We eliminate the edges of the pressure grid with the following coefficients:

$$e_{W_i} = \begin{cases} 0 \text{ if } i = i_{min} \\ 1 \text{ otherwise} \end{cases} \qquad e_{E_i} = \begin{cases} 0 \text{ if } i = i_{max} \\ 1 \text{ otherwise} \end{cases} \qquad e_{S_j} = \begin{cases} 0 \text{ if } j = j_{min} \\ 1 \text{ otherwise} \end{cases} \qquad e_{N_j} = \begin{cases} 0 \text{ if } j = j_{max} \\ 1 \text{ otherwise} \end{cases} \tag{4.10}$$

Then, we define a relaxation iteration as follows:

$$p_{k+1_{i,j}} := (1-\omega)p_{k_{i,j}} + \left[\frac{\omega}{\frac{e_{E_i}+e_{W_i}}{(\Delta x)^2} + \frac{e_{N_j}+e_{S_j}}{(\Delta y)^2}}\right]\left[\frac{e_{E_i}p_{k_{i+1,j}} + e_{W_i}p_{k_{i-1,j}}}{(\Delta x)^2} + \frac{e_{N_j}p_{k_{i,j+1}} + e_{S_j}p_{k_{i,j-1}}}{(\Delta y)^2} - \frac{\nabla \cdot \vec{G}(\vec{u})}{\Delta t}\right]. \tag{4.11}$$

The parameters Δx and Δy are the grid spacings in the x and y axes, and ω is used to control the speed of the relaxation. Griebel et al. [40] discuss the optimal choice of this parameter for different types of fluid.

For this iterative process, we need an initial pressure field p_0, and we use the pressure values from the previous time-step. We calculate $p_{i,j}^{k+1}$ over the entire grid for each iteration of the relaxation. The number of iterations can be controlled by estimating the residuals of the pressure field as follows:

$$\varepsilon_{p_{i,j}} := \frac{e_{E_i}(p_{i+1,j}^k - p_{i,j}^k) - e_{W_i}(p_{i,j}^k - p_{i-1,j}^k)}{(\Delta x)^2} + \frac{e_{N_i}(p_{i,j+1}^k - p_{i,j}^k) - e_{S_i}(p_{i,j}^k - p_{i,j-1}^k)}{(\Delta y)^2} - \frac{\nabla \cdot \vec{G}(\vec{u})}{\Delta t}. \quad (4.12)$$

We then iterate the relaxation process until some norm of the residual field is within a tolerance of our choosing. We use a norm to reduce the residual field into a single scalar number. We can test to see if our solution has sufficiently converged by comparing the norm of the residual field to a suitable tolerance. Again, we follow the work of Griebel et al. [40] and use the maximum $\varepsilon_{p_{i,j}}$ value as our norm, because we want to limit the residual across the entire pressure field. Other norms like L_2 are often used.

4.2 Semi-Lagrangian Method

Stam [102] introduced the concept of a semi-Lagrangian method to the graphics community. His method simulates the Navier-Stokes equations in a manner that is different from classical grid-based methods. Stam's approach is still a type of Eulerian method, in that space is still discretized as a mesh. The semi-Lagrangian method gets its name from the fact that it uses particle tracing to handle the advection term in Equation 3.1. Because of this advection technique, Stam's method is much more stable than classical Eulerian methods, generally allowing for much faster simulation, though at the cost of physical accuracy. The graphics community, however, has deemed the semi-Lagrangian method to be convincing enough for its purposes.

4.2.1 Projection to Non-divergence

Stam's method [102] starts in a similar manner to that of the classic Eulerian method. His approach computes an intermediate approximation of the velocity field without any regard for pressure forces. The method then projects the velocity field into a divergence-free space. This projection makes use of the *Helmholtz-Hodge decomposition* [16], which separates any vector field $\vec{w}$ into a divergence-free field $\vec{u}$ and a scalar field q, which are related as follows:

$$\vec{w} = \vec{u} + \nabla q. \quad (4.13)$$

We can then define the projection

$$P(\vec{w}) = \vec{w} - \nabla q = \vec{u} \quad (4.14)$$

and thus get the following approximation to the momentum equation:

$$\frac{\partial \vec{u}}{\partial t} = P\big(-(\vec{u} \cdot \nabla)\vec{u} + \nu \nabla^2 \vec{u} + \vec{f}\big). \quad (4.15)$$

For applications which require pressure values (such as the pressure gradient interaction method in Section 4.3.1), the scalar field q multiplied by the density of the fluid ρ can serve as an approximation.

4.2.2 Semi-Lagrangian Advection

Stam's method [102] computes the advection term of the momentum equation by treating each grid cell as a particle location. These particles are then traced backward through the flow field for the duration of the time-step. Essentially, Stam's particle tracing technique is trying to find the location of the fluid that will flow into the cell over the course of the time-step. Velocity values are then taken from that old location and assigned to the grid cell. Of course, this back-traced location could be between grid cells, in which case the values are interpolated from within the particle's neighbouring cells. One other special case is that a particle might drift off of the edge of the grid. In these cases, Fedkiw et al. [31] show that the particle locations can be clamped against the edge of the grid's boundary regions and still produce good results.

The aforementioned advection scheme has the property that any new velocity values cannot exceed the values of the grid before the time-step. This essentially means that the stability conditions due to advection described in Section 4.1.2 do not apply. However, the physical accuracy of the simulation does diminish with larger time-steps, as would be expected. Also, when external forces are added to the system, the combined system can still be unstable, as we shall see in Section 4.3.2.

4.2.3 Implicit Diffusion Operator

To compute the diffusion of the velocity field that is caused by viscosity, one could simply use a forward Euler approach as follows:

$$\vec{u}_\nu(t) = \vec{u} + \nu \Delta t \nabla^2 \vec{u}(t). \tag{4.16}$$

However, if the viscosity ν is large, then the system becomes unstable, as was discussed in Section 4.1.2. Stam [102] instead shows the following implicit formulation of the diffusion:

$$(I - \nu \nabla^2)\vec{u}_\nu(t) = \vec{u}(t) \tag{4.17}$$

where I is the identity operator. We could use standard matrix inversion techniques, such as those described by Lang [56], to solve for $\vec{u}_\nu$ at each step. However, Stam [102] states that the system is sparse and linear, and so is more efficiently treated by the famous Gauss-Seidel relaxation method [103].

4.2.4 Term Splitting

Recall from Section 4.1.2 that the classical Eulerian approach requires us to split off the pressure term of the momentum equation, so that it could be solved separately. Stam's method also splits the terms of the momentum equation. However, his method splits each of the terms into their own steps, as follows:

1. Add external forces: $\tilde{u}_a(\vec{x}) := \vec{u}(\vec{x},t) + \Delta t f(\vec{x},t)$
2. Diffuse velocities for viscosity: $\tilde{u}_b(\vec{x}) := (I - \nu \Delta t \nabla^2)^{-1} \tilde{u}_a(\vec{x})$
3. Advect particles: $\tilde{u}_c(\vec{x}) := \tilde{u}_b(\vec{x} + \Delta t \nabla \tilde{u}_b(\vec{x}))$
4. Project velocities onto a divergence-free field: $\vec{u}(\vec{x}, t + \Delta t) := P(\tilde{u}_c(\vec{x}))$

In practice, the advection step is more accurate if the velocity field is divergence-free, but the accelerations due to viscosity and external forces would add divergence to the field. Thus, another projection step is commonly done just before computing the advection.

4.2.5 Accuracy Concerns

Although Stam's method is unconditionally stable, it does have issues with regard to accuracy. In particular, the method has been shown to introduce a fair amount of numerical dampening to the fluid flow [102, 31]. This dampening will cause the fluid's velocity field to slow down and become motionless relatively quickly compared to the classical Eulerian method. To counteract this deceleration due to dampening, animators commonly lower the viscosity value for their simulations, or even remove the viscosity term from the Navier-Stokes equations altogether [8].

Another consequence of the high dampening in Stam's method is that small-scale variations in the flow, such as vortices or turbulence, will be smoothed from the velocity field rather quickly. Although the same small-scale flows will be removed from any Eulerian simulation, and is a numerical issue with all discrete simulations, this smoothing effect is particularly noticeable with Stam's method. To combat artifacts due to smoothing, Fedkiw et al. [31] gave a means of exaggerating small-scale flow features in order to keep them from disappearing. Similarly, one could just add artificial variation back into the fluid flow to give the appearance of a more chaotic flow [97, 91]

4.3 Grid-based Fluids and Spring-Mass Elastics

As we saw in Chapter 2, we can use a mesh of springs to represent a volume of elastic material. The nodes of the mesh are point-masses that represent finite amounts of mass and volume, and links in the mesh are springs that introduce elastic forces. However, when interacting point-based solid representations with grid-based fluids, we have to reconcile the differences in representation. In graphics literature, the most common way to do this reconciliation is to lock any grid cells that are occupied by the spring-mass system [9, 41, 32, 14]. The locked grid cells are then treated as no-slip or free-slip boundary conditions, and these conditions were described in Section 4.1.3. Essentially, the velocity of the locked fluid will be modified explicitly so that it coincides with the velocity of the solid boundary. If the boundary is a no-slip type, then the cell's velocity will literally be that of the solid boundary. If, however, the boundary type is free-slip, then cell's velocity will be broken up into components that are respectively normal to the surface, and tangential to the surface. The normal component must match the solid's velocity along that axis, whereas the tangential component will remain unaffected. In either case, the object is directly manipulating the velocity field of the fluid grid. To obtain two-way interaction between the fluids and solids, pressure differences between the solid and the fluid are computed across the boundaries, and are used to determine a force that is exerted by the fluid onto the solid. The details of this method are further explained in Section 4.3.1. The engineering and

mathematics communities use the opposite approach [87]. The velocity of each solid particle is interpolated from the fluid's grid, and the particles are advected by those velocities. Then, elastic forces are calculated for each solid point, and are spread out back onto neighbouring nodes of the fluid mesh.

No matter which of the above two methods we use, we have no means of preventing fluids from leaking across the boundary of a solid object that is in motion. This delicate problem is due to the reconciliation between the particle-based representation of the the elastic body, and the grid-based representation of the fluid. The leakage is especially noticeable when the boundaries are thin, and much work has gone into combating this leakage issue [14, 41, 104]. As we shall see in Chapter 5, particle-based fluid systems can more naturally prevent leaking, but have disadvantages with respect to their speed of simulation.

Interaction methods between fluid meshes and elastic bodies are typically more computationally expensive to simulate than either system alone. Generally, the elastic forces needed to move the fluids are quite large, and thus very strong springs would be needed to generate those forces. As we saw in Section 2.4, larger spring coefficients make the simulation of the elastic body more unstable. Also, the stronger springs induce larger velocities in the fluid. As we saw in Section 4.1.2, the stability of the fluid's advection is dependent on the velocity of the fluid. Despite the large amount of dampening that occurs in most fluids, these interaction problems are quite numerically stiff [104, 53]. In our own experiments, we found that systems involving both springs and grid-based fluids (either Eulerian or semi-Lagrangian) were far more unstable than the undampened spring-mass systems alone. Similarly, grid-based fluid methods are much more stable on their own than when elastics are integrated via the immersed boundary method.

4.3.1 Pressure Gradient Method

When modeling the interaction between a grid-based fluid and a particle-based solid, we are primarily interested in which fluid cells are occupied by the solid. With the so-called pressure gradient method, the interaction is modeled by treating the surface of each occupied cell as a boundary, as discussed in Section 4.1.3. We then ignore the contents of that cell in our fluid simulation. The velocities of the solid-occupied boundary cells are set to those of the part of the solid that occupies that cell. The type of boundary that is enforced (i.e., no slip, or free slip, as discussed in Section 4.1.3) can be chosen to suit the surface properties of the solid object.

In order to achieve a two-way coupling, the fluid must exert a force back onto the solid object. Generally, the solid object will have a uniform pressure within it that will be different than that of the fluid. We thus define a force due to the difference in pressure between the fluid mesh and the solid object. The force is then defined as:

$$\vec{F}_p = -\frac{\nabla pA}{m}, \tag{4.18}$$

where A is the surface area of the solid's boundary. The main problem with the pressure gradient method is that multiple solid particles (or multiple surface edges along the solid) could be in the same fluid cell. These multiple solid elements present two problems. First, each particle in a cell could have different velocities.

We would then have to interpolate some velocity from those particles, and apply it to the boundary condition for that fluid cell. Second, the surface edges could have different surface normals, and so would push the fluid in different directions. Again, we could interpolate a surface normal from the different edges. Still, both cases can cause a noticeable amount of fluid to leak through the solid, especially when the solid is thin with respect to the resolution of the fluid mesh [41]. However, the pressure gradient method is much preferred for rigid objects that are relatively thick. Rigid body motion is characterized by a Cartesian velocity in the center of mass and also an angular velocity to describe the solid's change in orientation, as we discussed briefly in Section 1.2.4. The fluid's pressure forces give us derivatives to the Cartesian velocity, but also if the force about the solid object is nonuniform, then the forces will induce a rotational torque on the object as well. Thus, rigid objects can be handled quite elegantly with the pressure gradient method. However, elastic solids are best served with other methods, as described in the following section.

4.3.2 Immersed Boundary Method

An alternative method for coupling mesh-based fluids with particle-based elastics is the immersed boundary method [87]. This method can be thought of as the opposite to the pressure gradient method, in which the solid object explicitly dictates the velocity of the fluid, and the fluid implicitly responds by imposing a force on the solid object. Conversely, the immersed boundary method imposes velocities on the solid particles from the fluid mesh, and then distributes elastic forces from the particles onto the fluid mesh. As such, this model works well with elastic solids, but does not work well with rigid solids because it does not enforce rigid body transformations. Also, the immersed boundary method can be used in conjunction with any grid-based fluid solver we might choose to use [104].

The immersed boundary method first computes the forces $\vec{F}_i$ acting on each of the particles in the elastic body, as described in Chapter 2. We then transmit that force onto the fluid grid using the term $\vec{f}$ in the Navier-Stokes momentum equation in Section 3.1. This term represents the acceleration of the fluid grid due to an external force. To determine the raw acceleration, we need to know the mass of each grid cell affected by the elastic particle. In our case of jellyfish flesh, the elastic body has roughly the same density as the surrounding sea water [22]. Since the fluid is essentially incompressible, the mass of each cell is the product of the fluid's density and the volume of that fluid. Our simulations are in 2D, so the mass is always

$$m = \rho \Delta x \Delta y. \tag{4.19}$$

We then have to spread the force from a continuous location of the elastic particle onto the discrete locations of the fluid grid. Literature from the mathematics community discusses ways of spreading the elastic forces over large neighbourhoods of grid cells [87, 104]. However, since our fluid grids for graphics applications are generally more coarse than those of scientific simulations, using large neighbourhoods would effectively smooth away small-scale variances in the force profile. Thus, we merely transmit the force of each elastic particle only onto the four grid cells that are closest to that particle.

With the vector field of forces calculated, we can then simulate a time-step of the fluid simulator. After that time-step, we then advect the elastic particles along the velocities of the fluid grid. Mathematically, the particle's position update can be characterized as

$$\vec{x}_i(t+\Delta t) = \vec{x}_i(t) + \Delta t \vec{u}(\vec{x}_i(t), t+\Delta t), \tag{4.20}$$

where $\vec{u}(\vec{x}_i(t), t+\Delta t)$ is the velocity of the fluid at position $\vec{x}_i(t)$ at the end of the time-step.

To ensure that our simulation is physically accurate, we must ensure that fluids cannot leak through the boundary of the elastic object. Guendelman et al. [41] proposed a complex means of ensuring that fluid in a grid-based system does not leak across a triangulated surface. Their work limits the grid cells from which values can be interpolated so that values are always taken from one side of the surface. This method requires processing the surface for each interpolated value, and can be quite expensive.

Instead, we make use of the general properties inherent to the immersed boundary method that help us attenuate leaking across boundaries. Since the elastic boundary is always moving with the same velocity as the fluid (within numerical error), less fluid leaks across the boundaries. For leaking to be prevented, the immersed boundary method has requirements on how far apart the elastic particles can be placed. Since elastic forces are calculated only for particles in the spring-mass system, as described in Chapter 2, we have to ensure that the density of particles is higher than the cell spacing of our fluid grid. Peskin [87] states that at no time should the space between spring-connected particles be greater than half of the grid spacing. Thus, we have at least two elastic particles per grid cell. Any more than that will still be physically accurate, though will increase the cost of the simulation. Martin et al [63] discuss a means of resampling the elastic body when linked particles are too far or too close to each other [87, 104]. In our experience, the relative cost of computing extra elastic forces is greatly overshadowed by the cost of computing the fluid velocities over the grid. Thus, we simply use a static mesh that is sampled to handle the largest amount of elastic deformation.

Time-stepping is another important consideration. Stockie and Wetton [104] discuss different time-stepping methods for an immersed boundary simulation, including forward Euler and higher order explicit methods such as RK4, as well as a semi-implicit method. They find that the semi-implicit method is about an order of magnitude faster than that of the forward Euler method. However, we also find that using the semi-Lagrangian method instead of a traditional Eulerian method brings the run-time down much more than that, and more than cancels out the gain of the semi-implicit method. Thus, we choose to solve the system with an explicit method that incorporates the semi-Lagrangian fluid solver. One future avenue of research would be the development of a semi-implicit time-stepping method that makes use of semi-Lagrangian principles.

We still have to determine a reasonable size for our time-steps. Stockie and Wetton [104] discuss the stability regions of several numerical methods, but their analysis is for the immersed boundary method that is coupled with an Eulerian simulation of the Stokes equations, which are essentially the predecessor to the Navier-Stokes equation that did not incorporate the notion of advection. Also, they do not provide concrete stability conditions to which we can adhere, but merely stability regions for particular parameteri-

zations. More recently, Kim and Peskin [53] give experimental results that suggest that the stability of the immersed elastic fibres is proportional to the square-root of elastic modulus of the fibres, though an exact relationship has yet to be derived. Thus, we merely use the standard stability conditions that are described in Section 4.1.2, scaled by a small safety factor. Experimentally, we find that these conditions prevent the simulation from becoming completely chaotic, but if the safety factor is too high, the resulting step sizes will allow a oscillatory artifacts to appear in the simulation. We presume that these artifacts come from the over-advection of the spring-mass particles when too large of a time-step is used, although further investigation into this phenomenon is required. Using a scaling factor on the order of 10^{-3} seems to alleviate instabilities in our system. Although this scaling factor may seem small, using it in conjunction with the stability conditions in Section 4.1.2, we do get better efficiency than merely using a constant step size.

CHAPTER 5

PARTICLE-BASED FLUID SIMULATION

Particle systems are quite common in the graphics industry, dating back to the original work by Reeves [93] in 1983. Particle systems are used to model fuzzy objects and approximate volumetric data, especially when that data is sparse and a grid or mesh representation would have many empty regions. Graphics literature saw several early attempts to use particle systems to represent liquid-state fluids [68, 112, 109]. Desbrun and Gascuel [26] first introduced the technique of smoothed particle hydrodynamics (SPH) to the graphics community as an alternative to mesh-based methods of simulating fluids. SPH had its beginnings in physics literature as a means of simulating stars and cosmic gases [36, 74, 72, 73, 63].

The fluid dynamics community has found that particle-based fluid simulations have advantages. Specifically, Lagrangian approaches are well suited for applications where the fluid volume is sparse within a scene, such as rivers or pools collecting on rugged terrain, especially when topologies are changing over time. Also, when the fluids are interacting with complex objects, mesh-based systems require us to reconcile the arbitrary geometry of the object with the fluid mesh. Particle-based approaches do not have such explicit restrictions.

Particle-based approaches also allow us to intrinsically enforce that fluids do not leak through solid objects. Fluid particles that try to enter a boundary region can be deflected away from the surface of the object. By contrast, mesh-based systems that interact with solid objects (either rigid or elastic) run the risk of leaking small amounts of fluid across the objects's boundaries, especially if the width of these boundaries is smaller than the resolution of the mesh. Much work has gone into correcting these leaks across mesh links, but methods thus far are extremely expensive [41, 104].

Also, most mesh-based methods lack the resolution to capture the subtle and complex geometries that can occur at the surface of a fluid. To counteract the resolution limitation of mesh-based methods, particle systems, especially particle level-sets [81], have been used to track the evolution of the fluid surface throughout the simulation. Again, we are forced to reconcile the particle-based surface representation with the mesh-based velocity field. With a particle-based fluid simulator, we intrinsically have particles from which to generate a surface, and have a wide array of methods to use to construct that surface.

The main downfall to particle-based approaches is that they do not take into account the nondivergence part of the Navier-Stokes equations. Thus, we have no way of enforcing that the fluid is incompressible. Note, however, that since the particles never change mass, we can intrinsically enforce conservation of

mass. For graphics applications which only deal with passive objects, we can still get very convincing fluid animations even without enforcing compressibility. However, for applications where the interacting solid objects greatly depend on the physical properties of the water, we do need at least an approximate notion of incompressibility. We cannot have absolute incompressibility with a particle-based system, so we simulate nearly incompressible fluids, though at a very high computational cost.

5.1 SPH Background

At its core, SPH is just a particle system with complex forces acting on the particles. We derive properties of the fluid at an arbitrary point $\vec{x}$ by taking a weighted average of that property from each particle within some neighbourhood region V near $\vec{x}$. The SPH method gets its name from the fact that values within the field are a smoothed version of the values within the neighbourhood. Thus, we approximate a property field q with the integral:

$$\tilde{q}(\vec{x}) = \int^V q(\vec{x}_j)W(r)dV, \quad \text{where} \quad r = |\vec{x}_j - \vec{x}|. \tag{5.1}$$

In the above equation, V is the volume (in 3D) or area (in 2D) that has influence over $\vec{x}$. In the continuous case above, $\vec{x}_j$ is a point in V, and so we are integrating over the entire neighbourhood. Lastly, W_h is the smoothing kernel, which is normalized so that the total weight given to the entire smoothing volume V is one. So, the first approximation that SPH makes, before even discretizing the integral, is that property values are smoothed away, effectively limiting the frequency at which $\tilde{q}$ can vary based on the size of V and the shape of W. We usually define the kernel W to be radially symmetric, and parameterize the kernel so that it has a desired radius of influence h. Thus, V is a circle or sphere with a radius of h. In fact, the original kernels used by Gingold and Monaghan [36] were Gaussian-like functions. We will discuss other possible kernels in Section 5.1.1.

Since we do not have a continuous field of point data, we further approximate the integral in Equation 5.1 as the weighted sum of values at the particle locations. The weight given to each particle's value depends not only on the smoothing function W, but also the ratio of the particle's mass to the density of fluid in that region (m_j/ρ_j). Thus, the discrete approximation for a property field is

$$\tilde{q}(\vec{x}) \approx \sum_{j=1}^{n} \frac{m_j}{\rho_j} q(\vec{x}_j)W(r_j). \tag{5.2}$$

Derivatives of the property field are approximated in a fashion similar to Equation 5.2, except that property values of each particle are weighted by the smoothing kernel's derivatives.

$$\nabla\tilde{q}(\vec{x}) \approx \sum_{j=1}^{n} \frac{m_j}{\rho_j} q(l_j)\nabla W_h(r_j). \tag{5.3}$$

$$\nabla^2\tilde{q}(\vec{x}) \approx \sum_{j=1}^{n} \frac{m_j}{\rho_j} q(l_j)\nabla^2 W_h(r_j). \tag{5.4}$$

The first property of the fluid, which we need to calculate others, is the density, which is defined as:

$$\rho(\vec{x}) \approx \sum_{j=1}^{n} \frac{m_j}{\rho_j} \rho_j W_h(r_j) = \sum_{j=1}^{n} m_j W_h(r_j). \tag{5.5}$$

A few different variations exist on how to calculate forces like those of pressure and viscosity. We follow the formulations given by Müller et al. [77]. They define pressure forces as follows:

$$\vec{F}^{pressure}(\vec{x}_i) \approx \sum_{j=1}^{n} \frac{m_j}{\rho_j} \frac{p_i + p_j}{2} \nabla W_h(r_j), \quad p_k = \kappa_p(\rho_k - \rho_0), \tag{5.6}$$

where ρ_0 is the rest density of the fluid, ρ_k is the smoothed density value for position $\vec{x}_k$ as determined in equation 5.5, p_k is a scalar that represents the pressure at a point $\vec{x}_k$, and κ_p is the gas constant (which is inversely proportional to the compressibility of the fluid). Just as with the elasticity coefficient κ_l and κ_θ in Section 2.4.4, the gas constant κ_p will make our system more unstable if it is very high. To simulate sea water, we would need an incredibly high gas constant, which makes the SPH method prohibitively expensive. For further discussion, Section 5.3 will show performance results of time-stepping methods with different fluid parameters.

Müller et al. [77] define the force of viscosity as:

$$\vec{F}^{viscosity}(\vec{x}_i) \approx \mu \sum_{j=1}^{n} \frac{m_j}{\rho_j} \vec{v}_j \nabla^2 W_h(r_j) \tag{5.7}$$

where μ is the dynamic viscosity of the fluid such that $\mu = \nu\rho$. Thus, if we put Equation 5.7 in terms of the kinematic viscosity, we get

$$\vec{F}^{viscosity}(\vec{x}_i) \approx \frac{\nu}{\rho_i} \sum_{j=1}^{n} \frac{m_j}{\rho_j} \vec{v}_j \nabla^2 W_h(r_j). \tag{5.8}$$

Other forces have been derived, such as surface tension [77] and drag [73], though we refer the reader to our references for further details.

You will note that we now have explicit functions for calculating the viscosity and pressure terms of the Navier-Stokes momentum equation that was defined in Section 3.1. We can incorporate external forces easily enough by applying those forces directly to the particles, as we did for elastic forces in Chapter 2. The advection term of the equation does not appear explicitly in our formulations. We get the effects of advection purely from the fact that the particles themselves move along their flow velocities. However, we do have to account for incompressibility, which we approximate with high gas constants in the pressure term of Equation 5.6.

5.1.1 Smoothing Kernels

The choice of a smoothing kernel is important, for accuracy and stability reasons. We shall see that certain kernels (and their derivatives) can cause certain visual artifacts to arise in our animations. We have a few possible choices for our smoothing kernel W. The original work of Gingold and Monaghan [36] used a normalized Gaussian function. Later, Monaghan and Lattanzio [74] introduced a more computationally

efficient kernel based on beta-splines. Monaghan [72] later warned that Equation 5.1 does not have an exact physical interpretation when used with anything but the Gaussian kernel. However, the graphics community is more concerned with believability and speed. Thus, various kernels have been developed with purely graphics interests in mind.

One problem that has been encountered by the graphics community with SPH is related to the fact that the gradients of both the Gaussian and beta-spline kernels approach zero when r approaches zero. Since the force of pressure depends on the gradient of the kernel, particles that are close together will exert only negligible pressure forces (as described by Equation 5.6) upon each other. This mathematical quirk is not desirable because it causes particles to clump together, rather than spread apart. Desbrun and Gascuel [26] combatted this clumping issue by introducing a kernel that is spiky and peaks at $r = 0$. Their smoothing kernel has the largest gradient when $r = 0$, thus causing closer particles to push away with greater force, as one would expect. Figure 5.1 shows the Gaussian kernel that was used by Gingold and Monaghan [36], and the spiky kernel that was proposed by Desbrun and Gascuel [26].

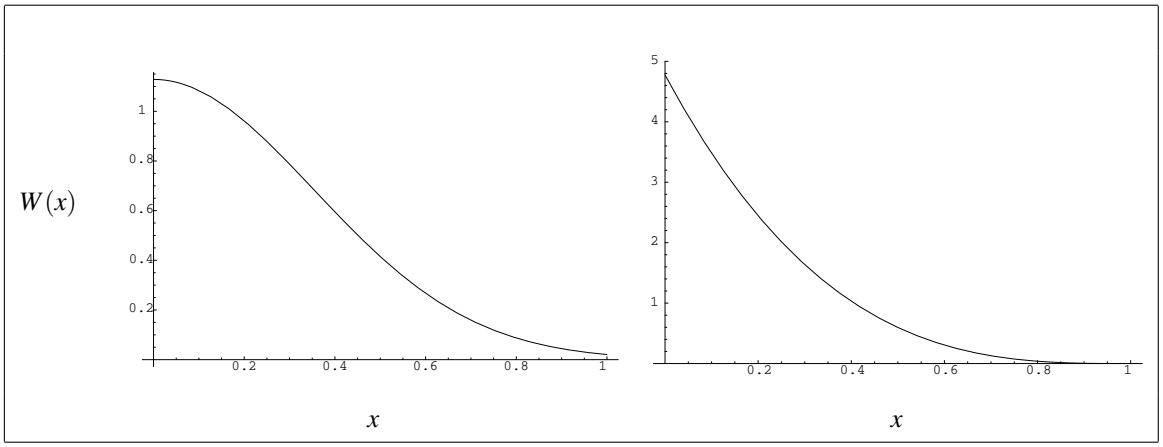

Figure 5.1: Possible smoothing kernels. The left plot shows the original kernel by Gingold and Monaghan [36], which is simply a normalized Gaussian function. The right graph shows the spiky kernel that Desbrun and Gascuel [26] used for pressure calculations. In both cases, the x axis is the distance between particles, and the $W(x)$ axis is the response of each kernel at that distance.

Müller et al. [77] introduce some new kernels to address other problems. Their work is the first in the graphics literature to propose different kernels for each term of the force equation. Like Desbrun and Gascuel [26], they suggest a spiky kernel for the pressure term, but they also give two other kernels: one used for viscosity, and another that is used everywhere else. Our work most closely resembles theirs; the main difference is that their kernels were tailored specifically for 3D simulations, but our work is in 2D. Thus, we had to develop our own kernels, though based on those of Müller et al. [77].

When developing a kernel, we have four constraints on the function. The first three constraints are that the value $W(r)$, gradient $\nabla W(r)$, and Laplacian $\nabla^2 W(r)$ all must be zero at $r = h$. Keep in mind that the

gradients and Laplacians are defined with respect to the coordinate system being used. In 3D, a spherical polar system is normally used, so the Laplacian is defined as:

$$\nabla^2 W(r) = \frac{\partial^2 W(r)}{\partial r^2} + \frac{2}{r}\frac{\partial W(r)}{\partial r}. \tag{5.9}$$

However, for our 2D simulations, we use cylindrical polar coordinates where the values with respect to the Z-axis are constant. The Laplacian for this 2D coordinate system is defined as:

$$\nabla^2 W(r) = \frac{\partial^2 W(r)}{\partial r^2} + \frac{1}{r}\frac{\partial W(r)}{\partial r}. \tag{5.10}$$

The gradient vector, in both 2D and 3D cases, we get as follows:

$$\nabla W(\vec{x}) = \hat{x}\frac{dW(|\vec{x}|)}{d|\vec{x}|}. \tag{5.11}$$

Note that we are assuming that the kernel is radially symmetric, which is true of all the kernels we have described. Thus, the above Laplacians and gradients do not have terms involving derivatives with respect to the angles.

The final constraint on our kernel is that it must be normalized: that is, the kernel's definite integral over the smoothing region V must be 1. In 3D, the integral is

$$\int^{V_{3D}} W(|\vec{x}|)dV_{3D} = 4\pi\int_0^h r^2 W(r)dr, \tag{5.12}$$

and the 2D version is

$$\int^{V_{2D}} W(|\vec{x}|)dV_{2D} = 2\pi\int_0^h rW(r)dr, \tag{5.13}$$

So with both the spiky and polynomial kernels given by Müller et al. [77], the three requirements that $0 = W(h) = \nabla W(h) = \nabla^2 W(h)$ are already satisfied. However, the given kernels are normalized with respect to 3D polar coordinates, so we must renormalize for 2D. Thus, our 2D kernels are:

$$W_{2D}^{spiky}(r) = \frac{10}{\pi h^5}\Big\{(h-r)^3\Big\}, \quad r \in [0,h], \tag{5.14}$$

and

$$W_{2D}^{poly6}(r) = \frac{4}{\pi h^8}\Big\{(h^2-r^2)^3\Big\}, \quad r \in [0,h]. \tag{5.15}$$

The last kernel that Müller et al. [77] give is for calculating the force of viscosity:

$$W_{3D}^{viscosity}(r) = \frac{15}{2\pi h^3}\Big\{-\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} + 1\Big\}, \quad r \in [0,h]. \tag{5.16}$$

Note that the kernel has a singularity at $r = 0$ because of the third inner term. At first, this singularity may seem troubling. However, the viscosity force only depends on the Laplacian of the smoothing kernel, as seen in Equation 5.7. The spherical polar Laplacian shown in Equation 5.10 actually removes the term with the singularity, and we get the following well-behaved function:

$$\nabla^2 W_{3D}^{viscosity}(r) = \frac{45}{2\pi h^6}\Big\{h-r\Big\}, \quad r \in [0,h]. \tag{5.17}$$

Since the kernel itself and the gradient are never actually used in the viscosity term, we can use the above kernel for 3D smoothing without any problems. Also, we can normalize this kernel properly, because the $\frac{h}{2r}$ term is also removed when integrating over a 3D sphere. Intuitively, the fact that the kernel has extremely large values for a small radius r is counteracted by the fact that a sphere with radius r has an extremely small volume. Thus, the kernel is safe for use in 3D. Despite the fact that this term is removed from the Laplacian and the integral, we still need it to satisfy the constraints that the function and its gradient are both zero when $r = h$. However, we cannot simply renormalize the kernel for 2D uses. The Laplacian in 2D polar coordinates does not remove the troubling term in the same manner. Instead, we look for a new kernel that works in 2D.

Since we have four constraints on our kernel, we need four degrees of freedom to satisfy those constraints. The Laplacian must be linear, so we need a term involving r^3. We need another degree of freedom that will disappear when integrated over the entire smoothing volume, and also when differentiated with respect to the 2D polar coordinate system. We find that the natural log $ln(x)$ does just that. The other terms that we choose for our kernel involve 1 and r^2. Thus, the equation is:

$$W_{2D}^{viscosity}(r) = a\big(r^3\big) + b\big(r^2\big) + c\big(ln(r)\big) + d, \tag{5.18}$$

where the factors a, b, c, and d are factors for which we must solve. With our four constraints on our kernel that were mentioned above, we get the following smoothing kernel:

$$W_{2D}^{viscosity}(r) = \frac{10}{\pi h^5}\left\{\frac{-5h^3 + 9hr^2 - 4r^3 + 6h^3(ln(h) - ln(r))}{9}\right\}, \quad r \in [0,h]. \tag{5.19}$$

The above kernel has all the same properties in 2D that its counterpart does in 3D, except that its Laplacian is slightly different:

$$\nabla^2 W_{2D}^{viscosity}(r) = \frac{40}{\pi h^5}\Big\{h - r\Big\}, \quad r \in [0,h]. \tag{5.20}$$

Figure 5.2 shows the three smoothing kernels that we use in our SPH simulations.

5.1.2 Initial Conditions

As with any initial value problem to be solved, the SPH system needs initial conditions. First, we need to decide how many particles to place in the system, or rather the density of spatial samples. Martin et al. [63] state that the Gaussian-like smoothing kernels that are used by the astrophysics community behave best when each neighbourhood has around 30 particles. It is not clear that this statement holds for the different smoothing kernels that are used by the graphics community. However, it seems reasonable that the spatial resolution of our simulation should be inversely related to the size of the smoothing volume V.

Since the smoothing kernel removes high-frequency features from the property fields, we have to choose the smoothing kernel's radius h so that it is small enough to allow us to capture the features that we want. In our case, the strong contractions of our elastic bodies create sharp features in the pressure gradient. We thus need a small smoothing radius, which also implies that we need a larger number of particles.

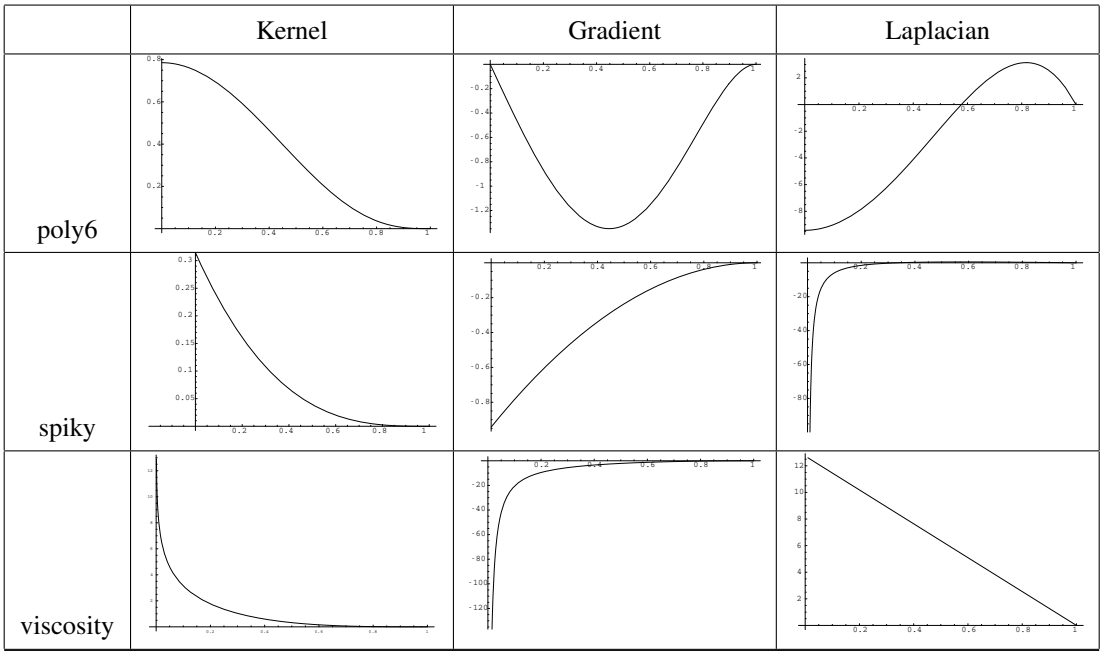

Figure 5.2: Plots of several smoothing kernels that we use in our 2D simulations, along with their gradients and Laplacians. In all cases, the *x* axis is the distance between particles, and the *y* axis is the response of the smoothing function.

Once we know the sample density for an SPH system, we need to determine the positions and velocities of all points (both fluid particles and solid point-masses). Martin et al. [63] discuss a few different configurations of fluids in general. Basically, the probability distribution of the particle velocities is used to categorize the state of the fluids: *chaotic* fluids have uniformly distributed velocities, *thermalised* fluids are characterized by a Maxwell-Boltzmann distribution, and *crystalline* fluids have no kinetic energy (i.e., no velocity) whatsoever. One could imagine more complicated configurations that involve specific flows and streams.

For now, we discuss the relatively simple configurations brought up by Martin et al. [63]. We can easily generate particle velocities that initially are any of the above-mentioned flow configurations, but the system will very quickly try to move toward its natural configuration. Sometimes, the system will explode or collapse in the process of trying to stabilize. So, the good placement for particles is essential, not just for realism and accuracy, but for stability at the onset of the simulation. Monaghan's initial work [72] talks about placing the points in a regular grid (specifically, at the center of the grid cells). However, we find Monaghan's approach to be an unstable configuration. We prefer a triangular mesh, in the same way that triangle meshes are preferred for spring-mass systems over quadrilateral grids. In a triangular mesh, the particles can be more tightly packed, as illustrated in Figure 5.3. In the grid packing on the left part of Figure 5.3, a downward force such as gravity would cause an entire row of particles to shift downward either to the left or right at the onset of the simulation. If each row were to shift in this manner, then the entire

configuration would become triangular, though only after adding an overall downward flow to the entire fluid. We can prevent this initial flow by packing the particles in a triangular manner, so that each particle is supported by the two that are below them. We can generate the triangle mesh in several ways, depending on the desired result.

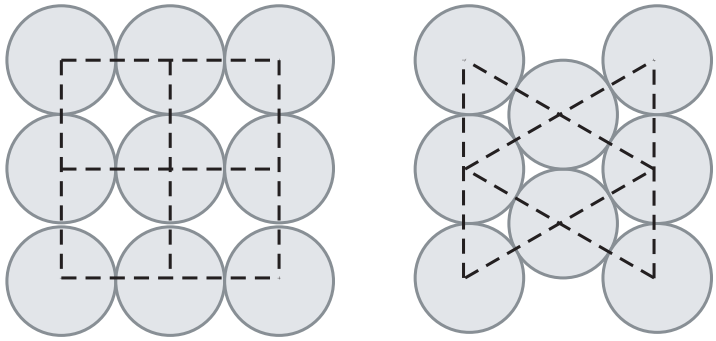

Figure 5.3: Different particle layout configurations. The left diagram shows particles that are packed into a grid-like pattern. The right diagram shows a triangular structure in which particles are more tightly packed.

In applications where a crystalline structure is desired, one could use Lloyd's iterative Voronoi algorithm [59] to find an optimal packing of the samples. However, we still find that our implemented system will have some variability in density, especially along the boundaries of the fluid. Thus our system does have some initial energy, as pressure forces will introduce movement into the system. Of course, with crystalline configurations, all particles have initial velocities of zero.

For thermalised fluids, we do not need particle placements that are as precise as what Lloyd's algorithm yields. We can merely pack the particles in a mesh of equilateral (or even isosceles) triangles. To add some initial turbulence to the system, we can merely add a vector $\vec{\varepsilon}_p$ of uniform or Gaussian noise to the position of each particle. We then initialize the velocities to be in the opposite direction as $\vec{\varepsilon}_p$ (as that is often close to the direction in which the pressure force pushes), with the length of each velocity vector roughly two orders of magnitude smaller than $\vec{\varepsilon}_p$.

Lastly, for both thermalised and crystalline configurations, we have to ensure that we populate the fluid regions with the proper densities. Thus, we not only have to assign each particle a position and velocity, but also a mass. Many SPH systems assume that all particles have the same mass, though there is no reason why particles could not have different masses. However, allowing each particle to have its own mass makes harder the decision of what the mass of each particle should be. So, we also decide to use uniformly massive particles. Still determining which mass to use is hard to do analytically, especially when different smoothing kernels are being used. A trivial and naïve approach is to use the volume V of the fluid and the number of

particles n_p and the desired rest density ρ to get the mass per particle m_p like so:

$$m_p = \frac{\rho V}{n_p}. \tag{5.21}$$

However, when the smoothed values of the densities are computed using the smoothing kernel W, the density of the initial population will not generally equal that of the desired rest density. We could integrate the kernel to find a better analytical answer.

Instead, we propose a suitable numerical solution that works uniformly across all kernels, and does not require us to derive integrals for each kernel. We can use Equation 5.21 as an initial guess for a bisection search that will give us a density profile that is within some tolerance of the desired rest density. We can also search based on the acceleration of particles (or equivalently, the force acting on particles) that arise from our particle population and mass assignments. For thermalised fluids, the tolerances used can be very liberal, or the mass optimizations can be done before adding noise to population. For crystalline structures, more precise mass values are needed, and one could potentially use various minimization techniques, like hill-climbing and simulated annealing [94], to find the mass value that yields the lowest acceleration.

One drawback of any numerical optimization of the initial mass values is that it requires computing the initial velocities of all particles for each iteration of the optimization. We note that, in cases when the fluid is completely enclosed (i.e., there is no free surface of the fluid), the rest density ρ_0 is irrelevant. As Equation 5.6 showed, we are more interested in the gradient of the density, or rather the gradient of the pressure, and other forces acting on each particle depend on the ratio of the particle's density to that of the surrounding region. Effectively, we can more liberally choose the initial masses of the particles without affecting the stability of the system at the beginning of the simulation. However, in doing so, we may reduce the physical accuracy of the simulation, which may limit the usefulness of the simulation for some applications.

5.1.3 Spatial Boundaries for SPH

In our system, we do not explicitly enforce any boundary conditions such as those described in Section 4.1.3. We merely detect collisions between fluid particles and solid surfaces. Once a collision is detected, we can compute the change in velocity both for the fluid particle and the solid object (if the solid is not fixed to be motionless). This collision-handling process is the sole means of interaction between solid boundaries and the fluid. The details of the collision detection and handling process will be explained more in Section 5.2, where we further discuss methods of interacting fluids with elastic bodies.

Using collisions as a means of fluid-solid interaction is akin to the free-slip boundaries for grid-based fluid simulators, as was described in Section 4.1.3. The particles are allowed to move along the boundaries without frictional forces or other restrictions. However, our collision-handling method prevents the particles from flowing through the boundary. Müller et al. [80] discuss the treatment of other types of boundaries explicitly.

For the static boundaries of the simulation itself, we have to repel the particles away from the boundary. If we allowed the particles to be pressed directly against the boundaries, then fluid would essentially lose volume, as the region that is occupied by the fluid would be smaller. Some researchers [80, 63] have suggested placing fake fluid particles along the boundary. Doing so will cause fluid particles that are far away to be pulled toward the boundary with artificially strong forces. Conversely, particles that are close to the boundary will be pushed away too quickly. We handle these boundaries by mirroring each particle (both its position and velocity) within the smoothing radius h from the boundary. Thus, pressure forces from the mirrored particle will repel the actual particle away from its mirrored self as it gets too close to the boundary. By repelling particles away from the solid boundaries, the fluid particles will not get pushed up immediately against the boundary These same pressure forces will attract a particle to the boundary when the region's density is less than the rest density ρ_0. The particle will also align its trajectory with the boundary because viscosity forces will involve a weighted sum of the particle's velocity and its mirrored velocity. Figure 5.4 illustrates the mirrored particle boundary approach that we use.

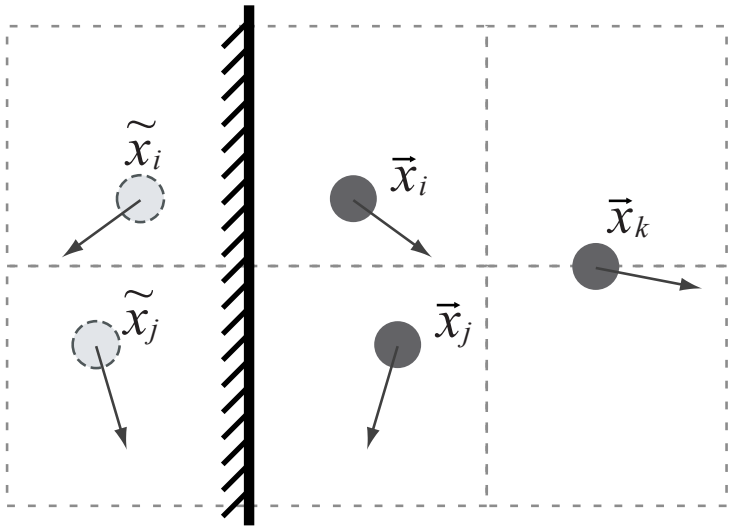

Figure 5.4: Fluid particles being mirrored by a solid boundary. Note that particle $\vec{x}_k$ is not mirrored, because its distance from the boundary is greater than the smoothing radius h. Thus, the particle's mirrored position would be too far away from any real particles to have an influence on them.

5.2 SPH Fluids and Spring-Mass Elastics

Müller et al. [80] have investigated the use of SPH codes with elastic bodies. Their work does much to faithfully capture the physical processes of fluids as they are repelled away from solid boundaries by pressure forces, and vice versa. In their work, pressure forces across boundaries are generated by placing fixed

particles along the boundaries of the elastic body. Müller et al. [80] do describe a robust means of placing these boundary particles. However, when the surface of the elastic body undergoes large deformation, the distribution of particles may become too sparse, and the surface would need to be resampled. Another issue with their method is that it still depends on collision handling, as fluids that are highly compressible will still allow fluid particles particles to come into contact with the solid boundary. Instead of simulating pressure forces between the fluid and the solid object, as did Müller et al. [80], we let particle collisions serve as the sole means of interaction between the fluid and the solid body. Since we have to compute collisions anyway, we may as well use them for interaction purposes. Doing so reduces the computation of each step in the simulation, for several reasons. First, by ignoring solid pressure forces, we reduce the number of particles over which we have to compute force terms at the boundaries. Also, we do not have to resample the boundary particles along the solid at each step. Lastly, we do not have to worry about confusing the particles within a solid with those of the fluid volume, thus simplifying methods for generating free surface interfaces.

One last issue with the work of Müller et al. [80] is that their method can trap thin layers of fluid between two boundaries. If two objects, whose surfaces are parallel to each other, close upon fluid particles, the pressures exerted between the fluid and solids will leave a thin layer of fluid between the two surfaces. Although this is technically what happens in the real world, the amount of fluid that is trapped between two such surfaces would be negligible compared to the volume of fluid that an SPH particle represents. Our method allows the two boundaries to come into actual contact, and we explicitly coerce fluid particles to move out of the gap between the two boundaries.

With regard to collisions, we can draw on classical physics to determine how simple objects react to collisions [4, 44]. When handling more complicated objects, graphics literature gives an efficient means of doing so [7]. A more interesting and difficult a problem is the detection of collisions. Again, we have a fair body of literature on the efficiency of collision detection algorithms [7, 50, 110].

5.2.1 Handling collisions

Before we discuss how we detect a collision and determine the position and time of that collision, let us first discuss how we handle the collision once it is detected. We will discuss exactly how we go about detecting and locating the collision in Section 5.2.2. Once we know that a collision has occurred and found its location $\vec{x}_c$, we then have to determine the change in velocities of the particle and the boundary. Physics literature tells us much about how to handle the collision of two rigid objects, and allows for different types of collisions. Suppose that we have two masses m_a and m_b, whose initial velocities before the collision are $\vec{v}_{ai}$ and $\vec{v}_{bi}$, and whose final velocities are $\vec{v}_{af}$ and $\vec{v}_{bf}$. A characteristic of all types of collisions is that momentum is conserved. The momentum of a particle is $p_j = m_j v_j$, and so conservation of momentum can be expressed as follows [44, 4]:

$$\vec{p}_{ai} + \vec{p}_{bi} = \vec{p}_{af} + \vec{p}_{bf} \;\Leftrightarrow\; m_A\vec{v}_{af} + m_b\vec{v}_{bf} = m_a\vec{v}_{af} + m_b\vec{v}_{bf}. \tag{5.22}$$

We are interested in a particular type of collision: *elastic* collisions. These types of collisions conserve kinetic energy as well as momentum. We assume that solid objects (both rigid and elastic) will be involved in elastic collisions. For simplicity, we also assume that fluid particles will collide elastically with solid obstacles. To enforce conservation of kinetic energy, we first observe that the total kinetic energy in the system is defined as [44, 4]:

$$E_k = \sum_j m_j \frac{\left|\vec{v}_j\right|^2}{2}, \text{ where } \left|\vec{v}_j\right| \text{ is the norm of vector } \vec{v}_j. \tag{5.23}$$

To conserve energy in our elastic collisions, we have to enforce

$$m_a \left|\vec{v}_{ai}\right|^2 + m_b \left|\vec{v}_{bi}\right|^2 = m_a \left|\vec{v}_{af}\right|^2 + m_b \left|\vec{v}_{bf}\right|^2. \tag{5.24}$$

This equation, coupled with Equation 5.22 still gives us an under-constrained system to solve [44] for the final momenta of the colliding bodies. We use one common solution to this system, given by Benenson et al. [4]. In this solution, we first find the axis of collision $\vec{c}$. In our case, we can use the surface normal of the elastic body at the collision location $\vec{x}_c$. We then decompose each momentum vector $\vec{p}_j$ into two vectors $\vec{p}_{j_{||}}$ and $\vec{p}_{j_\perp}$, which are the components that are respectively parallel and perpendicular to $\vec{c}$. In other words,

$$\vec{p}_j = \vec{p}_{j_{||}} + \vec{p}_{j_\perp}, \quad \widehat{p}_{j_{||}} \cdot \widehat{c} = 1 \; or \; -1, \quad \vec{p}_{j_\perp} \cdot \vec{c} = 0. \tag{5.25}$$

The solution to the system that we use is to swap the parallel components of the momentum vectors, like so:

$$\vec{p}_{af_{||}} = \vec{p}_{bi_{||}}, \quad \vec{p}_{bi_\perp} = \vec{p}_{ai_\perp}, \tag{5.26}$$

$$\vec{p}_{bf_{||}} = \vec{p}_{ai_{||}}, \quad \vec{p}_{bf_\perp} = \vec{p}_{bi_\perp}. \tag{5.27}$$

Essentially, we assume that the particles are of equal mass and that energy is conserved. These assumptions give us enough constraints to solve the system. However, our assumption that masses are equal is not always correct.

With our spring-mass representation of elastic bodies, fluid particles generally do not collide with the point-masses of the elastic mesh. They instead collide somewhere along the edge (i.e., spring link) between point-masses. To complicate matters, each point-mass can have a different mass value (and some may even be infinite to represent very heavy objects, or fixed locations). To handle this intricacy, we compute two different collisions: one with respect to each of the two point-masses attached to the spring. So, for a point-mass e_j that is part of the boundary edge that collided with the fluid particle, we get the final velocities of the fluid particle ($\widetilde{v}_{F_{jf}}$) and the boundary point-mass ($\widetilde{v}_{E_{jf}}$), assuming that they had collided independently. We get similar values $\widetilde{v}_{F_{kf}}$ and $\widetilde{v}_{E_{kf}}$ from computing a collision with the same fluid particle, but the other point-mass that is attached to the same colliding boundary edge. We then use the collision point $\vec{x}_c$ (as determined in section 5.2.2) to generate an interpolation coordinate s, which we define as follows:

$$s = \frac{||\vec{e}_j - \vec{l}_c||}{||\vec{e}_j - \vec{l}_c|| + ||\vec{e}_k - \vec{l}_c||}. \tag{5.28}$$

So, s will be close to 0 when the collision occurs near endpoint e_j, and will approach 1 as the collision location moves closer to e_k. We then use s to linearly interpolate the resulting velocity changes to the particle, like so:

$$\vec{v}_f = (1-s)\widetilde{v}_{F_{jf}} + (s)\widetilde{v}_{F_{kf}}. \tag{5.29}$$

We also use s to interpolate between the intermediate endpoint velocities ($\widetilde{v}_{e_{jf}}$ and $\widetilde{v}_{E_{kf}}$) and the initial velocities ($\vec{v}_{E_{ji}}$ and $\vec{v}_{E_{ki}}$). Remember that s represents the normalized portion of the spring's length that is between the collision location $\vec{x}_c$ and endpoint $\vec{x}_{E_j}$. So, we interpolate the final velocities for the end-points from their initial velocities and the intermediate ones as follows:

$$\vec{v}_{E_{jf}} = (1-s)\widetilde{v}_{E_{jf}} + (s)\vec{v}_{E_{ji}}, \quad \vec{v}_{E_{jf}} = (s)\widetilde{v}_{E_{jf}} + (1-s)\vec{v}_{E_{ji}}. \tag{5.30}$$

With all the interpolation above, we do risk violating the condition that kinetic energy is conserved. However, we have not observed any issues resulting form this approximation. A similar approach has also been done in 3D by Müller et al. [80], as well as Bridson et al. [7].

Once we have the new velocities, we then project our particles to their new locations as a result of that change in velocity. We do not know the exact location of the collision, but rather only a linear approximation thereto. Similarly, we do not know exactly when in time the collision occurred, though we could derive an estimate based on the s interpolation parameter. However, as will see in section 5.2.2, it is advantageous to give the particle and the elastic boundary as much separation as possible. So, we instead assume that the collision always occurs at the beginning of the time-step, and project both the particle and the elastic boundary along their new velocities for the full length of the time-step.

5.2.2 Collision Detection

To treat the collisions between a fluid particle and a solid boundary, we first must be able to detect the collision, and identify when and where it occurred. The scientific community has very precise means of detecting collisions. These collision events are characterized as a sharp discontinuity in the particle's velocity. So, the most numerically accurate way to handle a collision is to perform special event location to determine the position and time of the collision event, as described by Shampine et al. [98]. Briefly, event location consists of determining a very precise estimate of the time of the collision event (within the accuracy of the integration method). Since time-steps can be arbitrarily large, we have to interpolate between the discrete times at which our steps occur, and could use a root-finding algorithm such as Newton's method or a bisection algorithm to find the precise time of the event. We then stop integrating the differential equations at that event time, and compute the new velocities of the colliding objects (but leave the positions alone). Finally, we restart integration with the new initial values for the velocities at the time of the collision event. This type of approach is unacceptably slow for systems in which dozens or hundreds of events are occurring per time-step. This precise kind of event location effectively puts huge restrictions on the size of our time-steps.

Alternatively, in the computer graphics community, it is common to perform individual integration steps as if there were no boundaries and then fix the trajectories of the objects after each step. This type of approach is used by Teschner et al. [110], Mao and Yang [61], and Müller et al. [77]. Processing the collisions at the end of each step is not as accurate as event location methods, but the trade-off in speed is more than worth it. For simplicity and speed, we use linear interpolations of the objects' positions at each time-step to determine when and where collisions occurred, even though our time-stepping methods may be of higher order. The use of a linear approximation for collision handling purposes allows us to treat the collisions more efficiently, while the use of high-order integration schemes allows us to use larger step sizes in our simulations.

In our initial attempts, we considered a particle's path as it intersects with the boundary of the elastic body. One approximate way to test if a collision occurred would be to search for intersections between the particle's path and the boundary edge's locations at both the start and end of the time-step. However, it is possible that particles can slip through a boundary undetected. A prime example would be a particle with no velocity. Thus, its path will not intersect with anything.

A slightly better way to detect particle collisions would be to consider the region that is swept by the boundary edge over the time-step. If we interpolate linearly, this region is a quadrilateral (in 2D) that could possibly be degenerate (i.e., the edges of the boundary region could intersect each other). These degenerate cases do not pose problems to our method because we detect collisions on all edges independently. So, if a particle enters this region in the same time-step, then it may have collided, but possibly not. In these cases, we cannot be sure if a collision occurred without further refining the time-step. As a less expensive alternative, we just assume that a collision occurred whenever a particle is in the swept region, and consider the location of collision $\vec{x}_c$ to be either where the particle's path intersected the region, or whichever point on the region's perimeter is closest to the particle's position. However, we always consider the axis of collision $\vec{c}$ to be the surface normal of the boundary, rather than the normal at the perimeter of the region. Figure 5.5 explains the process further.

Our method above has one main weakness. If particles are moving slowly compared to the boundary, then the boundary might move right over the particle without any intersections at either edge of the time-step. For this reason, we have to consider the boundary to be an area (in 2D) or a volume (in 3D). Then, if a particle is inside of the boundary region, it must have collided in that time-step. However, we have to decide which face of the boundary collided with the particle. We do so with some ambiguity, as to do this accurately is too slow. Instead, we project the particle's velocity in a straight line (both backwards and forwards). We find all faces of the boundary region that are on that projected line. We then choose the closest boundary edge that is on that projected line. However, if the particle has absolutely no velocity, then we merely choose the closest boundary edge in any direction from the particle. Figure 5.6 shows an example of our final method. As shown in the diagram, projecting the particle's velocity allows us to choose the real colliding edge more reliably. In that figure, we see that a particle has slipped into a boundary in the course of

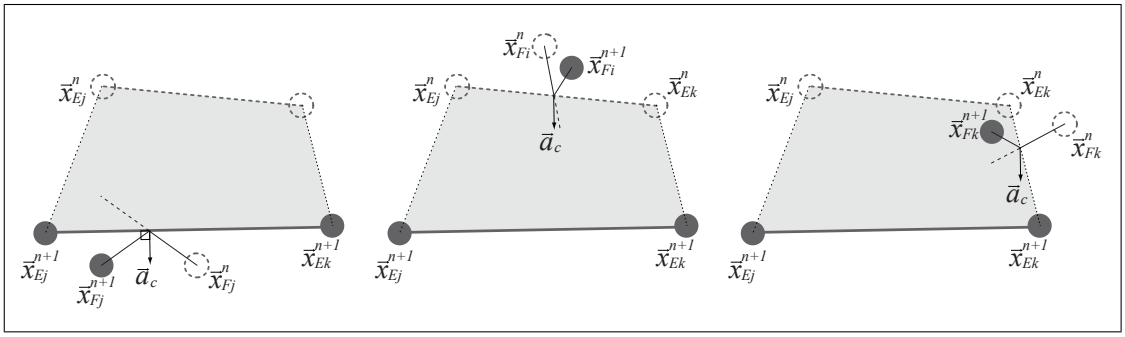

Figure 5.5: Detecting collisions by intersecting a particle's path with the region swept by a boundary. The left image shows the detection of a fluid-elastic collision at the start of the time step, as an intersection between the particle's path and the boundary's final position is found. The resulting changes in velocity are computed with respect to that starting time. The center image shows another collision detected and handled at the end of the time step, in a similar manner to the previous case. The right image shows a collision being detected and handled some time during the step. In this case, the particle's path intersects with the path of one of the solid boundary points.

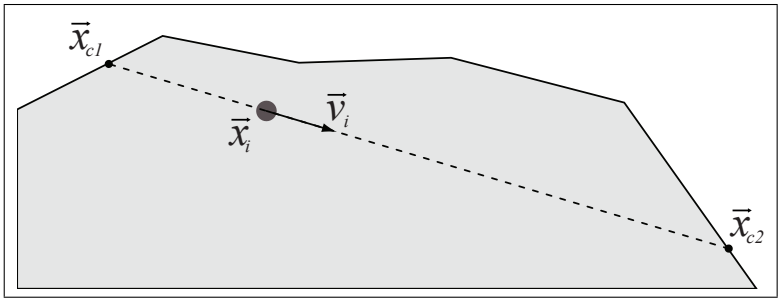

Figure 5.6: Collisions detected by finding the closest boundary that is in the path of a particle. In this diagram, the boundary edge directly above l_i is the closest edge, but is not along the directional heading of the particle. Thus, we consider the two boundaries on points $\vec{x}_{c1}$ and $\vec{x}_{c2}$ to be candidate edges. We then choose the closest of the two, which is $\vec{x}_c 1$, to be the edge with which the particle collided.

a time-step. If we merely push it to the closest boundary edge, then we will choose the edge directly above the particle. That is clearly the wrong edge for the collision. However, if we narrow our search to those that are along the particle's path, we choose the correct edge.

The possibility of choosing the wrong face still exists. This can happen when the particle is motionless, and its path takes it near a vertex or edge between multiple boundary faces. Figure 5.7 illustrates this example. The same thing can happen when a particle's directional heading changes drastically during the time-step. These cases are rare, and have little effect on the overall animation, so we ignore them.

There is another reason to treat boundaries as regions, rather than just infinitely thin faces. In the naïve system described above, particles that collide with the face will get pushed so that they are moving away from the boundary at one step. However, in the following time-step, various forces may push the boundary face ahead of the particle. A new collision will be computed, but to our above method, it will appear as though the particle was coming from the boundary's back side. Thus, the collision algorithm will push

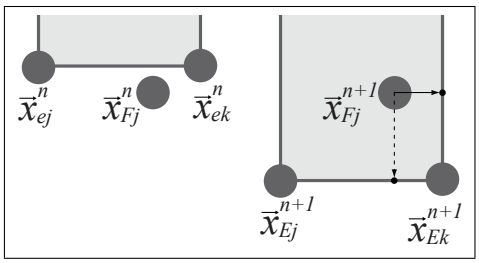

Figure 5.7: An example of our collision detection algorithm choosing the wrong edge. In this example, the particle at position l_j^n slips into the boundary volume near a boundary mesh node. On the left is the configuration of the system before the collision. On the right, we have the configuration after a time-step, where the boundary has moved down past the stationary particle. Since the particle has no velocity, our algorithm will choose the nearest boundary face to compute a collision. In the above case, the nearest face will not be the true face with which the particle collided.

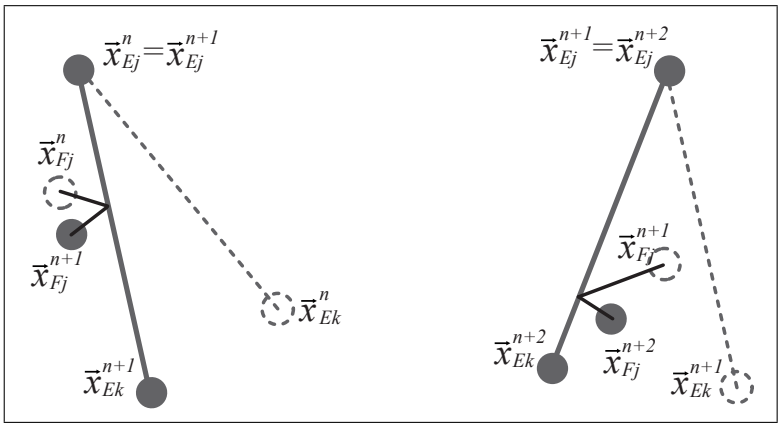

Figure 5.8: Collisions being processed twice in opposite directions on two different time-steps. On the left, a particle collides with an infinitely thin boundary in one time-step. On the right, the boundary is pushed past the particle (because of other forces acting on it) in the next time-step. In this second time-step, a naive collision-handling algorithm will (wrongly) treat the particle as if it has hit the boundary from behind.

the particle in the opposite direction from that desired. Figure 5.8 illustrates the situation. By treating the boundaries of elastic objects as areas or volumes, we may have particles that are just inside the boundary region for many consecutive time-steps, and are constantly being pushed away by the boundary, but the face of boundary itself is being pushed by other forces (e.g., elastic, gravity, etc.). This situation is analogous to a hard ball hitting a spongy surface: the ball will be in constant contact with the surface while being pushed away by that surface. In our system, the particle will stay in close proximity to the surface, and eventually be ejected from the boundary region. The particle will either fly away or roll down the face of the boundary, depending on the forces acting on the boundary faces.

Treating our boundaries as regions also has implications to our numerical integration. Essentially, we must ensure that our time-steps are such that no particle will go half-way through a boundary region. More on this will be discussed in section 5.3.

One last issue is that particles can be trapped between multiple boundaries. These cases will be a source of instability. Effectively, a particle that is trapped in this manner will zig-zag between the two boundaries at each step. The error in our collision-handling method will accumulate very quickly, causing the particle and any movable boundary to achieve unrealistic velocities. We attenuate the effects of these cases by detecting the event that a particle is trapped between more than one boundary, and then rotate the particle's velocity so that it is parallel to one of the boundaries, but has the same magnitude as before. For our purposes, a particle is considered to be trapped if it collides with more than one boundary in the same time-step, or if more than one boundary is within some small radius of the particle's position. Figure 5.9 illustrates the problem, and our solution to it. The top part of that figure shows a particle ricocheting between two boundaries, and the bottom part of that figure shows how we correct the problem.

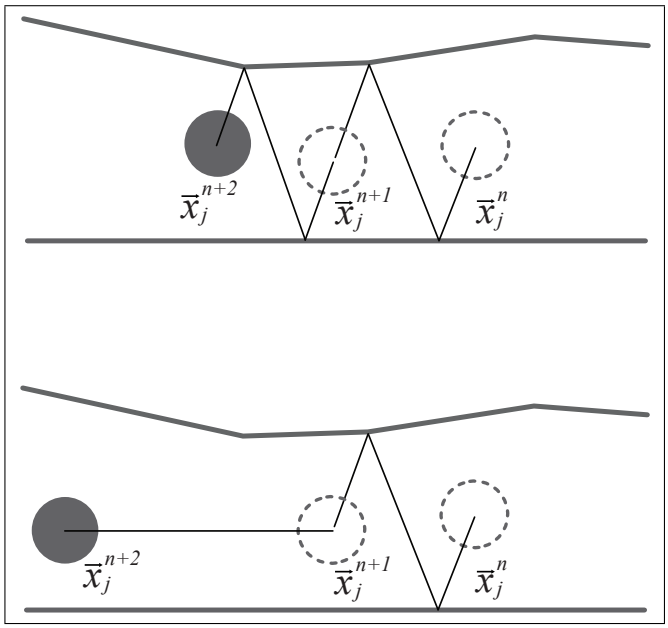

Figure 5.9: A particle being trapped between two boundaries. Both the top and bottom two images show two time-steps of a particle's trajectory. The top image shows a particle zig-zagging between two boundaries, causing multiple collisions on each time-step. The bottom figure shows a corrected case where we detect the trapping in the first time-step, and reorient the particle's trajectory to be along one of the boundaries.

Our method for handling trapped particles also has some implications on the realism of our simulation. By contrast, Müller et al. [80] do not generally allow two boundaries to stay in contact with each other. This is because their method computes pressure forces between the fluids and the solid objects. Thus, if two boundaries were closing in on each other, then any fluid particles between them would prop open a small gap. In reality, two solid objects would indeed trap a small amount of fluid between them. However, since real fluid particles are much smaller (and in greater number) than our SPH simulations can ever hope to handle, the volume of water that would be trapped in reality is negligible compared to that of an SPH particle. Thus, by allowing the two solids to stay in contact, as our method does, we are closer to what

would occur in reality. The only limitation of our method is that if a particle is trapped between a number of boundaries, it may not be able to escape and will be trapped in an infinitely small space. This effectively reduces the volume of the fluid that is in play. With this caveat in mind, we find these cases rare, and the loss of volume is acceptably small.

5.3 Discretization of Time

Normally, SPH researchers tend to prefer explicit methods, like the Forward Euler method [61], or a second-order Runge-Kutta method [54], or even the explicit leapfrog method that was used by Desbrun and Gascuel [26]. As was mentioned in Section 5.1, the stability of an SPH system greatly depends on the gas constant κ_p, which can be thought of as the inverse of the compressibility of the fluid. Essentially, this constant is proportional to the pressure forces that fluid particles exert upon each other. A high gas constant will result in a slow simulation because it will make the system more numerically stiff. Thus, an explicit Euler scheme is too slow for any but the most trivial of applications.

Higher-order methods, like the standard RK2 and RK4 decrease the computational cost of our SPH system. This raises the question: why not go to higher orders still? We investigate the use of a less-common fifth-order Runge-Kutta method developed by Dormand and Prince [28]. Monaghan [73] shows that fully implicit and semi-implicit methods do have their advantages, especially in multi-phase flows with high amounts of drag.

Desbrun and Gascuel [26] also mention a spatially adaptive time-step size, meaning that they integrate each particle independently, so that areas of greater stability are integrated more liberally. We have not tried these types of methods, mostly because we are unaware of any research into how these methods would fare against a spring-mass representation of deformable solids. Also, it is not clear how to implement localized step sizes with higher-order methods, though multi-step methods like BDFs [38] could offer some interesting possibilities, because they naturally allow for high-order extrapolation. However, we defer this issue to Section 9.2 where we discuss future work.

We do have room for globally adaptive step sizes. Desbrun and Gascuel [26] talk of a stability condition for each particle, based on the Courant-Friedrichs-Lewy condition for grid-based simulators that is described in Section 4.1.2. However, the given particle-based stability condition seems to favour first-order explicit methods. We are also unaware of any work showing that the condition Desbrun and Gascuel [26] actually controls the stability of the simulation. Instead, we opt to use standard numerical error estimation techniques, as described in Section 2.4. These error estimation techniques allow us to ensure the stability of the simulation, and as an added bonus also control accuracy. Again, the fifth-order Dormand-Prince method is excellent for error estimation purposes. This method has an embedded pair as described briefly in Section 2.4.6, so we can get two different methods of different orders from the same stage values, merely by placing different weights on the stages. Note that the standard second-order Runge-Kutta method also

has an embedded lower-order method, which is merely the forward Euler method. Thus, the second-order Runge-Kutta method can also be used for error estimation.

We saw in Section 2.4 that elastic bodies are generally best simulated with implicit methods. Fluids are a different sort of deformable mass, though whose point-masses are not statically connected in the same way as a spring-mass system. Explicit methods are commonly used with SPH systems by graphics researchers [61, 54, 26, 77].

One implication of our collision handling model is that the time-step of the system should be small enough that no particle will pass half-way through an object, or else the boundary edge that will be chosen for the collision will be the one that is on the opposite side of the object. This restriction on time-steps is due to the geometry of the model rather than of stability or numerical error. In practice, particles in our animations never reached such a high velocity that boundary thickness was a problem. The size of the time-steps was dictated by our error estimators, and not by issues of boundary thickness. However, for extremely thin boundaries, problems could arise. In these cases with thin boundaries, the time step should be limited to:

$$\Delta t < \frac{w_b}{2|\vec{v}_{max}|}, \tag{5.31}$$

where w_b is the smallest width of the boundary and $\vec{v}_{max}$ is the maximum velocity of any of the fluid particles. Various estimates for w_b could be used. In our 2D case, we could use the minimum height of any triangle in the mesh. However, boundaries that form wedges (i.e., diminish to a width of zero) will always be problematic. In these wedge-like cases, trade-offs between realism and simulation speed must be made.

CHAPTER 6

JELLYFISH BIOLOGY

In this chapter, we discuss the physiological aspects of jellyfish that are related to the organism's locomotion. Our interest in jellyfish is restricted to the adult phase of their life cycle, as this phase is when the jellyfish is most accomplished at swimming. We will see that jellyfish achieve locomotion by jet propulsion. The jellyfish has muscles throughout its flesh that contract in order to expel water downward from the cavity within it. By Newton's third law, if the organism is forcing water downward, then the water also exerts a force upward on the jellyfish.

Since we are simulating only a 2D slice of the jellyfish, we are interested in the biological aspects of the organism that can be modeled in 2D. However, the organism is not perfectly symmetric, and so the 2D slice will not be able to faithfully capture all of the 3D aspects of a real jellyfish. Thus, we also discuss aspects of the organism's physiology that could be used to extrapolate a more convincing 3D model from a 2D slice.

6.1 Basic Zoology of Jellyfish

Jellyfish have several distinct stages of biological development. Figure 6.1 shows the development cycle of a jellyfish. This development begins when two full-grown medusae reproduce sexually: a male fertilizes the egg of a female. The egg develops into a plantula larva, which has no control over its movement, and merely drifts until it attaches itself to the ocean floor. A settled plantula forms a scyphistoma polyp. The polyp develops into a strobila, which asexually produces ephyrae. An ephyra grows into mature medusa that can reproduce sexually. For the purposes of this thesis, we are only concerned with jellyfish in the medusan stage, because they have the most control over their locomotion. Also, the medusan stage is most well-studied by the biology community.

Many species of jellyfish have been classified. Figure 6.2 shows the taxonomy branch of jellyfish families that are indigenous to the Canadian Atlantic region alone [99]. All jellyfish have been classified under the phylum of Cnidaria, which also includes corals, hydras, and other aquatic animals that have stinging cells called nematocysts. The various species of jellyfish span two different classes, and several orders of the biological taxonomy, as can be seen in Figure 6.2.

Different species of jellyfish vary widely in their physical configurations, even among species in the same family. Figure 6.3 shows a small sample of jellyfish species in their medusan form that are all drastically

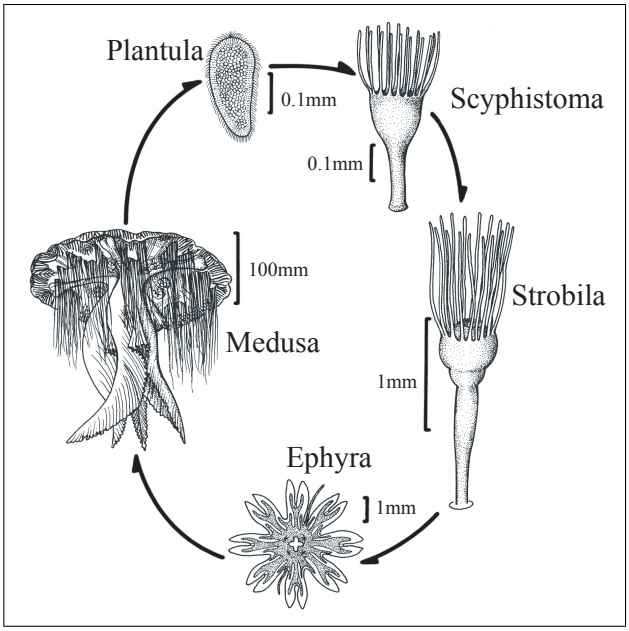

Figure 6.1: The various stages of a scyphozoan jellyfish's life cycle, as taken from Shih's work [99].

different in their geometry. Our proposed model is best suited to certain types of jellyfish that are symmetric, such as the species in the Mitrocomidae family. Examples of symmetric species of which we are interested are the Halopsis ocellata and Halicreas minimum, shown in Figure 6.3.

6.2 Jellyfish Physiology

Our model of jellyfish need only have a simple model of the organism's anatomical features. The prevalent parts of the medusan anatomy is are shown in Figure 6.4. In order for the jellyfish to achieve locomotion, the organism contracts its umbrella, expelling fluid from the subumbrellar cavity and producing thrust. The organism also has tentacles along the aperture of the umbrella. These tentacles are mostly passive, but introduce a component of drag into the system.

Jellyfish deflate their umbrella by contracting muscles that spread across the umbrella hull. Aria [2] provides some high-level information about the distribution of muscle fibres that lie underneath the epidermis of the umbrella. Figure 6.5 shows the muscular structure for one species of jellyfish. Gladfelter [37] gives us a more precise make-up of the tissue within the umbrella itself, which we show in Figure 6.6. Note that the chief muscle involved in locomotion is the circumferential muscle that lines the subumbrellar wall. When this muscle contracts, it pulls the umbrella inward, creating a fluid jet at the aperture of the umbrella, and giving thrust to the organism.

Most other organisms, especially mammals and birds, etc., have opposing pairs of muscles: one muscle that causes the contraction of a body part, and another muscle that returns the body part to its original

Superkingdom Eukaryota, Kingdom Animalia, Subkingdom Metazoa, Phylum Cnidaria			
Class	**Order**	**Suborder**	**Family**
Hydrozoa	Anthomedusae	n/a	Corynidae Tubulariidae Hydractiniidae Rathkeidae Bougainvilliidae Pandeidae Calycopsidae
	Leptomedusae	n/a	Dipleurosomatidae Melicertidae Laodiceidae Mitrocomidae Campanulariidae Eutimidae Aequoredae
	Trachymedusae	n/a	Ptychogastriidae Halicreatidae Rhopalonematidae
	Narcomedusae	n/a	Aeginidae
	Siphonophora	Cystonectae	Physaliidae
		Physonectae	Agalmidae Physophoridae
		Calycophorae	Hippopodiidae Diphyidae
Scyphozoa	Coronatae	n/a	Periphyllidae
	Semaeostomeae	n/a	Pelagiidae Cyaneidae Ulmaridae

Figure 6.2: Taxonomy of jellyfish, as taken from Shih's work [99].

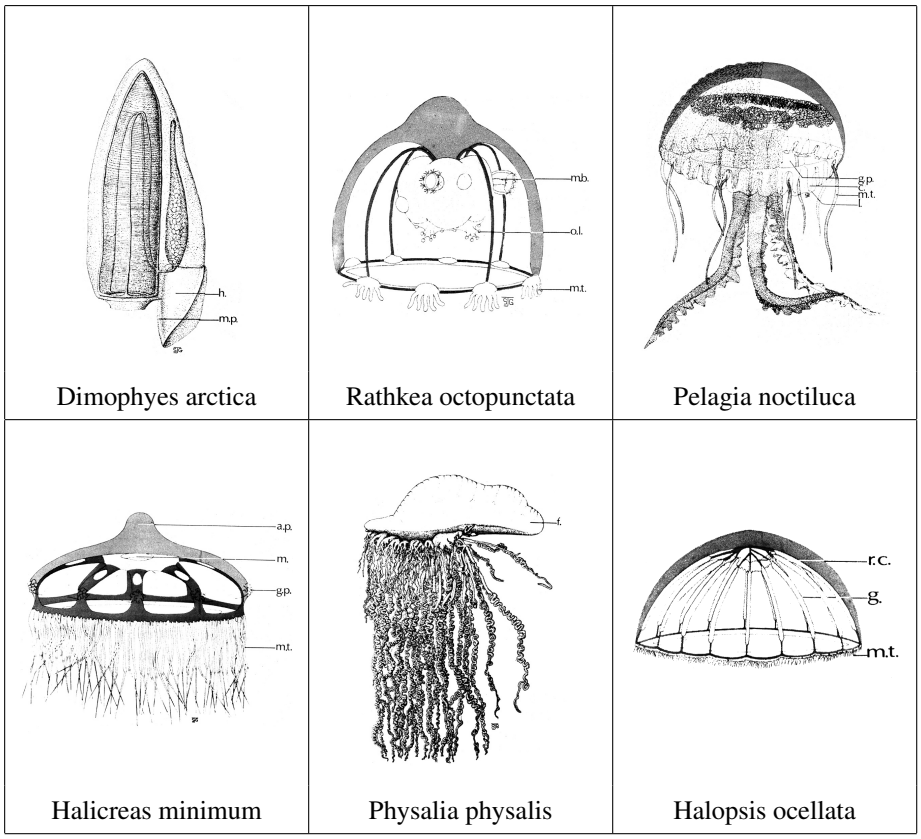

Figure 6.3: Some of the various species of jellyfish [99].

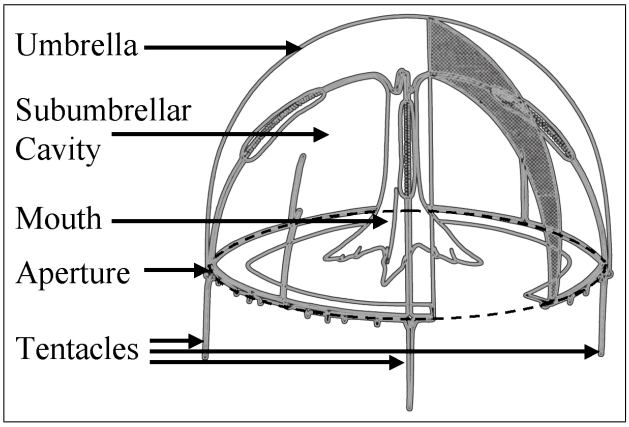

Figure 6.4: General anatomy of a jellyfish [99].

configuration. The combined action of the opposing pairs, working out of phase of each other, are used to effect locomotion. However, jellyfish lack this kind of muscle organization. The circumferential muscle has no opposing muscle. When the circumferential muscle contracts, it effectively shortens its rest length and elastic forces pull the umbrella inward. When the muscle is relaxed, its rest length is increased and the umbrella is pushed outward. In this manner, the contraction phase of the muscle must store enough energy to overcome the drag of the surrounding fluid and return the jellyfish back to its original shape when the muscle relaxes. Megill [65] goes to great lengths to show that enough energy is indeed stored in the contraction phase to power the refill phase of a jellyfish.

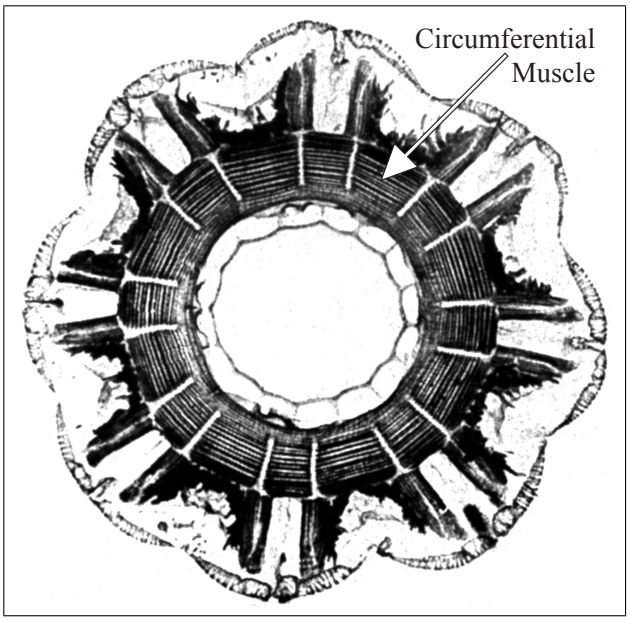

Figure 6.5: The subumbrellar surface of a medusan umbrella, with the tentacles removed to better expose the muscles [2].

Aside from the circumferential muscle, the remainder of the umbrella is loosely categorized into two regions of gelatinous bio-matter: the *bell mesoglea* and the *joint mesoglea*. We refer collectively to these two regions as *mesoglea*. Both of these regions are essentially passive. However, as illustrated in Figure 6.6, the bell mesoglea does have some sparsely placed muscle fibres embedded within it. These muscle fibres are oriented axially with respect to the umbrella (i.e., they are normal to the surface of the umbrella). Megill [65] states that these radial muscles are used to change the symmetry of the umbrella and thus affect a course change for the organism as it contracts and expands.

The joint mesoglea is so-called because it has a much lower elastic modulus (i.e., is less stiff) than the bell mesoglea. Thus, when the circumferential muscle contracts, pulling the umbrella to a smaller radius, the joint mesoglea squishes and deforms into sharp ridges or “joints”. This effect is illustrated in Figure 6.7.

Note that the compression pattern given by Megill [65] in Figure 6.7 shows the jellyfish expanding into two sinusoidal waves throughout the quarter section of the organism, suggesting that a jellyfish has eight

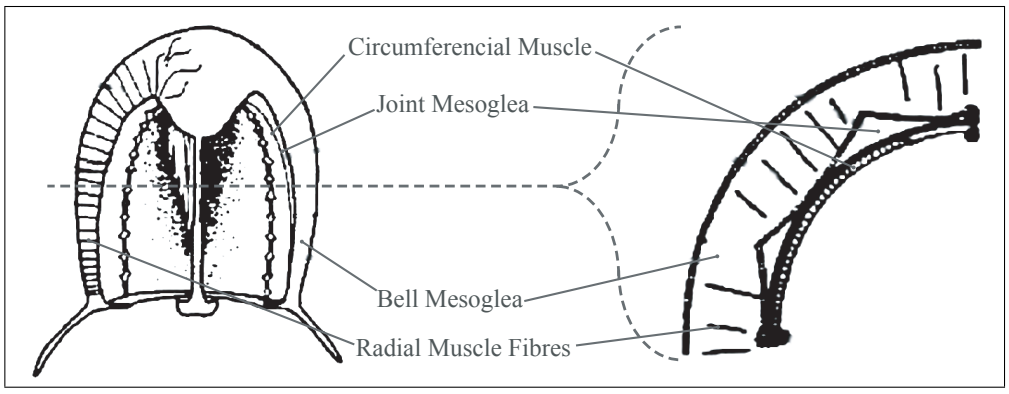

Figure 6.6: Horizontal and vertical cross-sections of a medusan umbrella, as originally described by Gladfelter [37], and adapted from a similar diagram by Megill [65].

such waves around its aperture. However, from looking at Shih's survey of species in the Canadian Atlantic region [99], we see that some species commonly have six, even twelve, or many more such segments, and these segments are not always sinusoidal, but often are parabolic.

Because of the joint mesoglea, the elastic forces that occur because of deformation are not a linear function of the displacement, as our linear springs were in Section 2.1. Instead, Megill [65] shows empirical evidence that the joint mesoglea give a non-linear force with respect to the amount of compression strain on the umbrella. Figure 6.8 shows Megill's stress-strain profile for an intact jellyfish. Note that the elastic response of the mesoglea is essentially linear except for extremely large stresses, where the mesoglea becomes more stiff and thus compresses less. Despite the non-linearity that exists in real jellyfish, we assume pure linear springs, for the sake of simplicity of our model.

Lastly, Spencer [101] noted that the entire circumferential muscle is not completely synchronized. The neuronal impulses that tell the muscle to contract must travel across the nervous system of the organism, and a minute delay is incurred at each neuronal synapse. The effect of this synaptic delay is that fibres which are further away from neuronal sources will receive the impulses at a slightly later time. Spencer tells us that jellyfish's nervous system is such that it divides the umbrella into quadrants, and that the source of the muscle activation impulse is initiated simultaneously around the perimeter of each quadrant, and travels inward across the the muscle sheet. Figure 6.9 illustrates the relative delay in muscle activation across the umbrella quadrant. In practice, we find that this delay pattern actually has little effect on the results of our model, but we include it in our model for completeness.

6.3 Modes of Locomotion

Jellyfish have some control over their own movement. They are active participants in their environment, in that they do intentionally try to swim to certain depths or engage in fishing activities, for example. However, to a large degree, they are at the mercy of their environment, and their movement is chiefly affected by

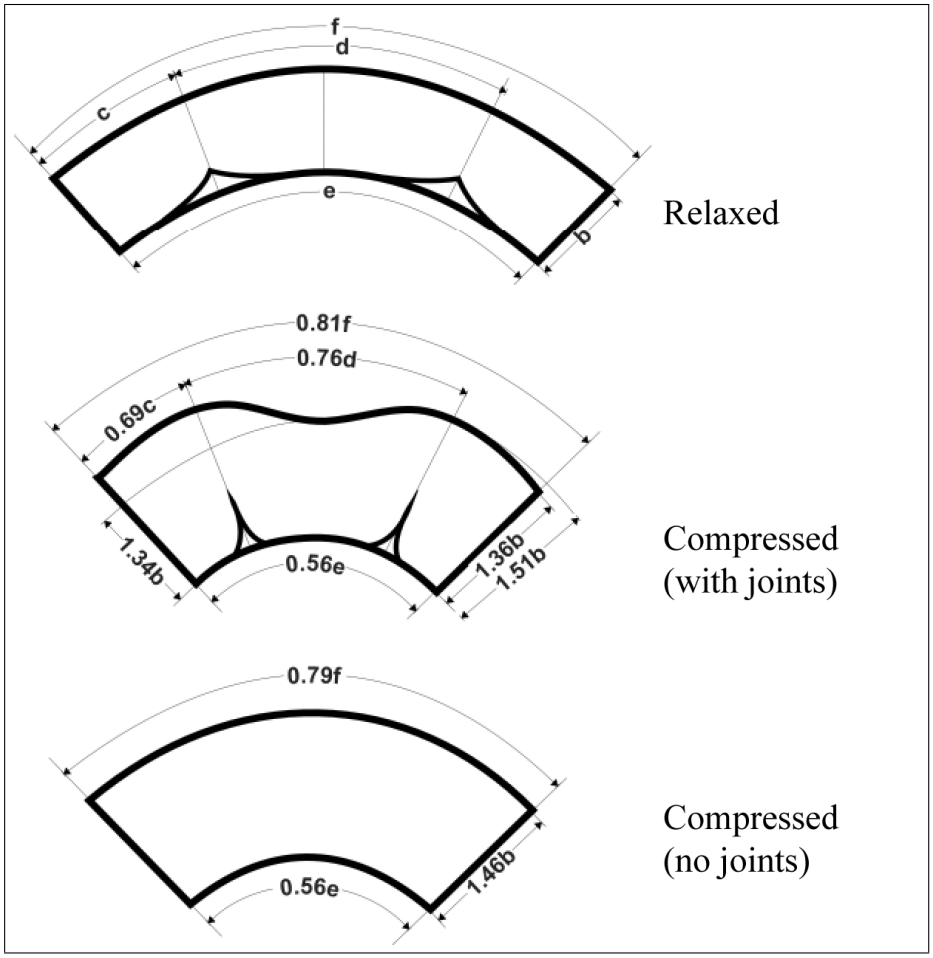

Figure 6.7: The deformation of the umbrella under the contraction of the circumferential muscle, as shown with a horizontal cross-section. The top image shows the umbrella at its rest configuration. The middle image shows the deformation that occurs with the presence of the joint mesoglea. The bottom image shows the hypothetical deformation that would occur if no joint mesoglea existed and the entire umbrella was bell mesoglea. All images are as described originally by Gladfelter [37], though these particular versions of the diagrams were redrawn by Megill [65].

ocean currents and tides. They do control their local movement, especially vertically with respect to the umbrella. Since many species of jellyfish are very sensitive to light, they often retreat to lower depths during the day and rise back toward the surface at night. A jellyfish achieves vertical locomotion by means of jet propulsion. Some jellyfish can control their tilt so that they thrust themselves in arbitrary directions [2, 65].

Megill [65] describes several different gaits, or modes of locomotion, of the jellyfish. First, and most important to our discussion, is the *resonant* gait, which is characterized by contractions at regular time intervals. By far, most of the biology research is on the resonant gait, and this is the gait that we aim to reproduce in our animations. Megill [65] spends much time studying the energetics of the resonant gait. Specifically, he finds an analytical approximation to the resonant frequency that the organism would have, given its elastic properties and an approximate notion of the dampening provided by the surrounding

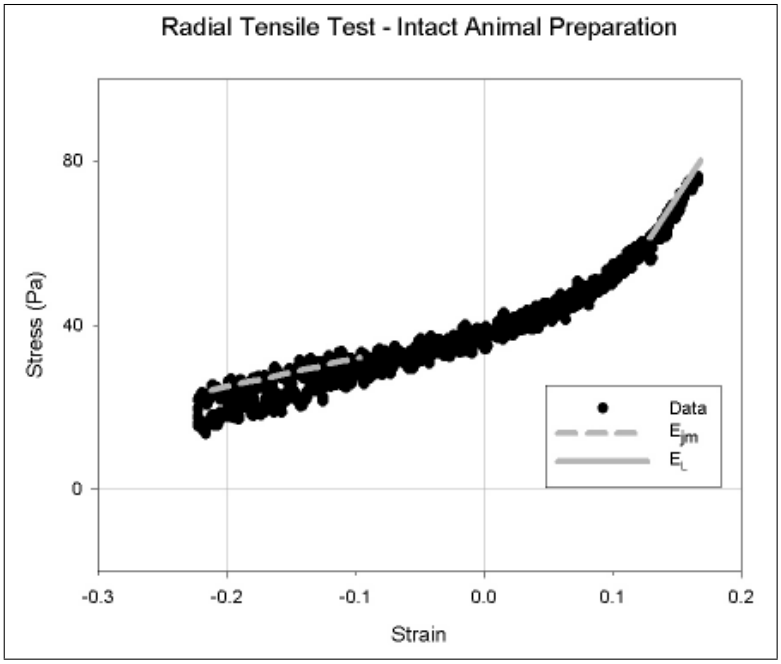

Figure 6.8: The stress-strain relationship of an intact jellyfish umbrella, as given by Megill [65]. Notice the rise in stress at the extreme end of the strain domain, indicating that the elastic forces are increasing superlinearly with respect to a given level of strain.

sea water. This resonant frequency is the optimal frequency at which the organism would oscillate in its contraction cycle. Indeed, jellyfish that contract at the resonant frequency would use the minimum amount of stored energy to achieve jet propulsion, thus requiring less food and rest time. In fact, the term "resonant gait" comes from the fact that the organism is oscillating at or near its resonant frequency when in this gait.

For our purposes, we are not so concerned about conserving the metabolic energy of the organism, but we do want our jellyfish to keep up appearances, so to speak. Thus, we want to induce a contraction frequency that resembles those of real jellyfish in their resonant gait. Based on measured data, Megill [65] finds that the resonant frequency of jellyfish to be in the range of 0.8 Hz to 1.4 Hz, and larger umbrellas will have a lower resonant frequency, which is to be expected. Megill's numbers are close to data captured by other researchers, but do not agree exactly [37, 22]. Figure 6.10 shows some empirical data on the morphology of jellyfish, as collected by Dabiri and Gharib [22]. From this figure, a contraction frequency of approximately 0.8 Hz is seen. However, the jellyfish that they studied were approximately four times the diameter of Megill's subjects. We make no assumptions about which data is correct, or that our model will be true to either result. Thus, we experiment with different contraction frequencies to find one that yields the optimal thrust for our system.

Megill [65] also notes other gaits that are used for a variety of reasons. A *transient* gait is similar to the resonant gait, except that the organism pauses longer between contractions, generally between 0.2 Hz to 0.8 Hz [65]. The net velocity achieved by the organism is lower, and so the transient gait is for low-speed locomotion. An even slower gait is that of *sink-fishing*, which entails the organism slowly sinking with its

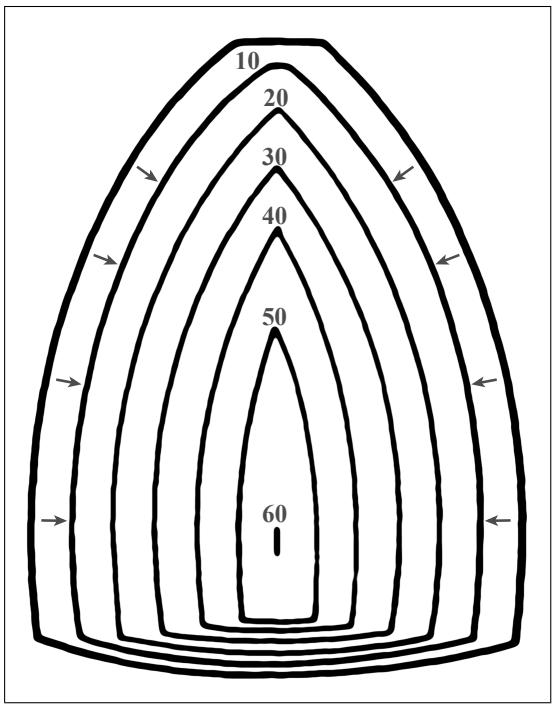

Figure 6.9: The profile of relative delay between for parts of the circumferential muscle, visualized as contoursSpencer [101]. All times are reported in milliseconds.

tentacles outstretched in order to capture prey. In the sink-fishing gait, the organism occasionally contracts to slow its decent and reposition itself. A final gait, referred to as *hopping*, involves the jellyfish standing on the bottom of the ocean with its tentacles, and occasionally contracting to push itself laterally or to stir up mud from the ocean floor. We ignore these less common gaits and concentrate on the resonant gait of jellyfish.

Sullivan et al. [105] provide some empirical data on the mechanics behind jellyfish movement and the reaction of the sea water surrounding the organism. They qualitatively show the flow of fluid around the organism as it contracts. Figure 6.11 shows the flow field that Sullivan et al. [105] measured. Although this data is not directly useful in building a model, it gives us some intuition as to what to expect in our own simulations.

Lastly, Dabiri and Gharib [22] give us empirical data on the relative position, velocity, and acceleration that is achieved by jellyfish. Figure 6.12 is a plot of their observations. Although we do not expect to get exactly these same results, we can aim for locomotion patterns that are qualitatively similar.

6.4 Previous Models of Jellyfish Motion

None of the biological models of locomotion have more than a trivial notion of real-world physics, especially fluid mechanics. However, several mathematical approximations have come from the experimental biology community. We discuss these methods briefly, as they may be useful for developing a procedural model of

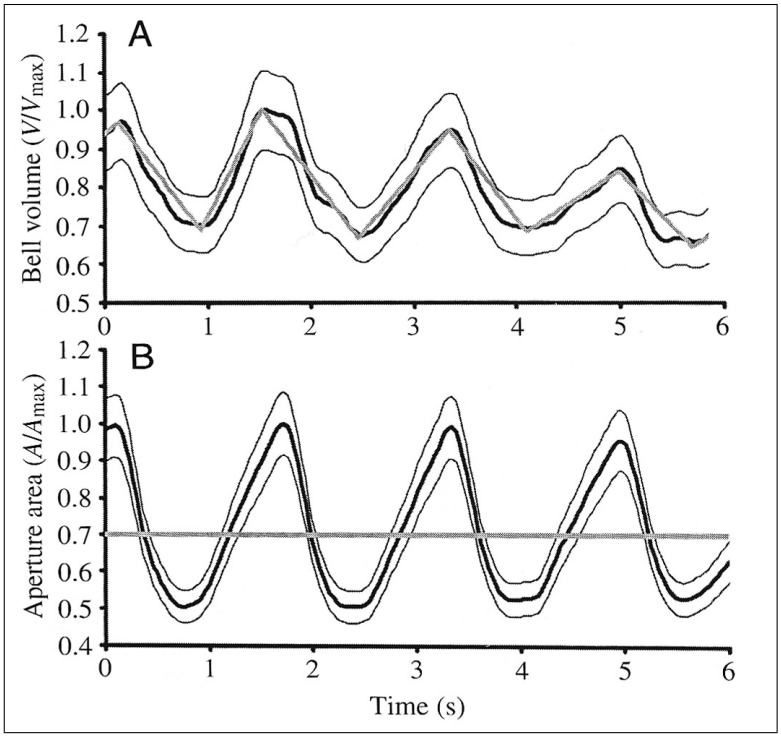

Figure 6.10: The geometry of a jellyfish as it evolves over time during locomotion in the resonant gait, as provided by Dabiri and Gharib [22]. The top graph shows the umbrella volume over time, whereas the bottom graph shows the aperture area over time. Both graphs are normalized with respect to the resting configurations of the organism. Both graphs show a thick solid line, which is the measured data. The thin solid lines are the uncertainty in the measurements.

jellyfish which is less scientifically rigorous, yet potentially less computationally expensive, than our own.

6.4.1 Approximate Numerical Models

The first and most simplistic model of jellyfish motion was developed by Daniel [23]. His one-dimensional medusan thrust model yields the thrust force F_T of the jellyfish that is supplied by a single contraction of the organism's umbrella. Each contraction pushes water away from the jellyfish, and by Newton's third law, the water must also push up on the jellyfish. The definition of thrust is:

$$F_T = v\frac{dm}{dt}, \tag{6.1}$$

where m is the mass of the fluid that remains in the umbrella, and v is the average velocity of the mass that is being expelled or thrust away from the jellyfish. So, in essence, Equation 6.1 is saying that the force is proportional to the mass of the water being expelled and the velocity at which it is being expelled. We can derive a thrust model for our jellyfish in terms of the subumbrellar volume $V(t)$ at time t, the area $A(t)$ of its aperture, and the density ρ of the surrounding water. If the velocity of the expelled fluid is

$$v = \frac{1}{A(t)}\frac{dV(t)}{dt}, \tag{6.2}$$

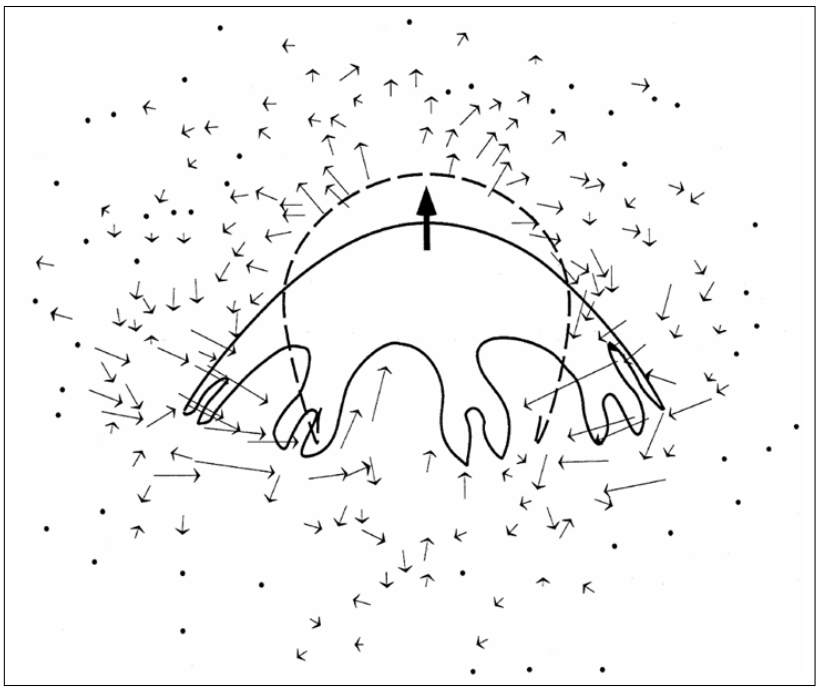

Figure 6.11: The measured flow of the fluid surrounding a jellyfish as it contracts, as provided by Sullivan et al. [105].

and the mass of the expelled fluid is

$$\frac{dm}{dt} = \rho \frac{dV(t)}{dt}, \tag{6.3}$$

then our equation for the thrust force of the jellyfish is

$$F_T = \frac{\rho}{A(t)} \left\{ \frac{dV(t)}{dt} \right\}^2. \tag{6.4}$$

A jellyfish's organic density is roughly equal to that of the surrounding water [25]. As such, the organism can be treated as neutrally buoyant. Megill [65] implies in his discussion of the sink fishing gait that jellyfish are in fact slightly more dense than water, and thus have a small negative buoyancy. However, buoyancy does not appear in Daniel's work. He describes his model with the following nonlinear differential equation:

$$V(t)\Big[1 + \Big(\frac{3V(t)}{2}\sqrt{\frac{\pi}{A(t)}}\Big)^{1.4}\Big]\frac{du}{dt} + \Big(\frac{12\pi^{0.25}A(t)^{0.65}}{2^{0.7}}\Big)u^{1.3} = F_T(t), \tag{6.5}$$

where $u(t)$ is the translational velocity for which we want to solve. The derivation of this equation is given by Daniel [23]. We can easily manipulate Equation 6.5 into an ordinary differential equation in terms of $\frac{du}{dt}$.

Dabiri and Gharib [22] solved Equation 6.5 numerically using the common fourth-order Runge-Kutta [43] method with a constant steps size. However, Dabiri and Gharib endeavoured to show that the model described in Equation 6.5 is very sensitive to its parameters, and thus an accurate measure of subumbrellar volume is needed for such an approach. In fact, their work was to show the high degree of sensitivity of Daniel's model. Even with empirical data for the volume and aperture area of the umbrella, they find that Daniel's model produces an acceleration profile that quickly diverges from empirically measured accelerations.

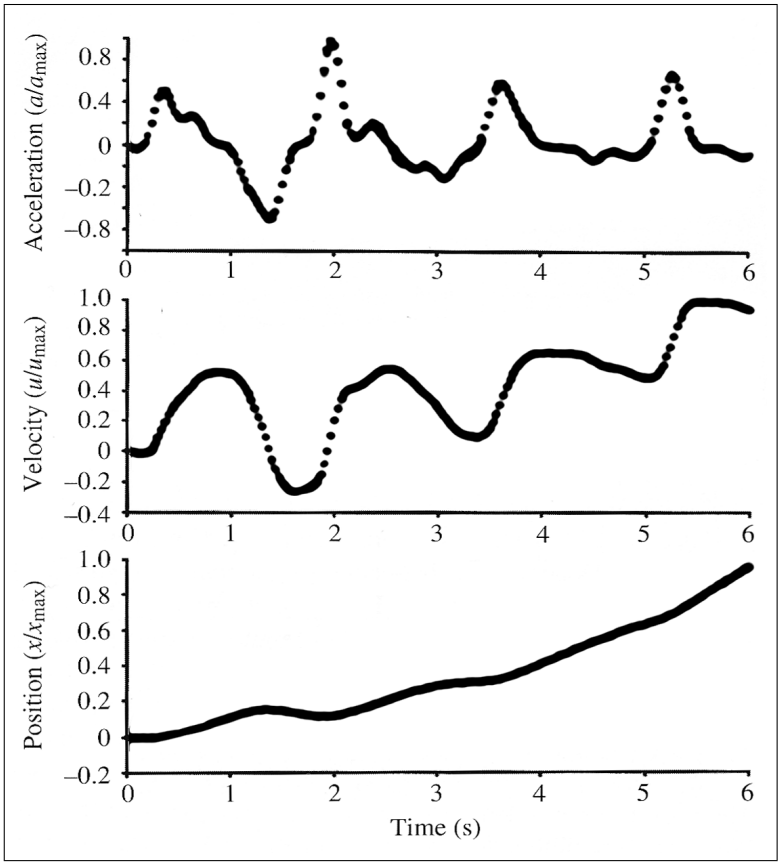

Figure 6.12: The relative position, velocity, and acceleration achieved by a jellyfish as it evolves over time during locomotion in the resonant gait, as provided by Dabiri and Gharib [22]. All values are normalized to be unitless.

6.4.2 Reduced Analytical Models

Megill [65] also studied the biomechanical properties of jellyfish, and determined the approximate resonant frequencies of the organism, given the properties of the organism and the surrounding sea-water. However, his model treats the organism as spring whose motion is dampened by drag forces. Because of the joint mesoglea, the spring is non-linear, rather than linear. Megill used a numerical means of solving the system, as he was not able to find an analytical solution. He considers not only the passive elastic forces of the mesoglea, but active forces from the circumferential muscle. His model also includes drag and treats the mass of not just the jellyfish, but the surrounding fluids as well.

Megill's work reports that the resonant frequency of his reduced model decreases when the size of the umbrella is larger. Presumably, the mass of the organism will also have an effect on its frequency, though no such effect is reported. This result is intuitively correct, and is analogous to how a large tuning fork resonates at a relatively low frequency. Figure 6.13 shows the amount of force needed to maintain a certain contraction amplitude with a given frequency. The figure has several plots for different sized umbrellas.

Megill's theoretical model predicts resonant frequencies that differ considerably from the contraction

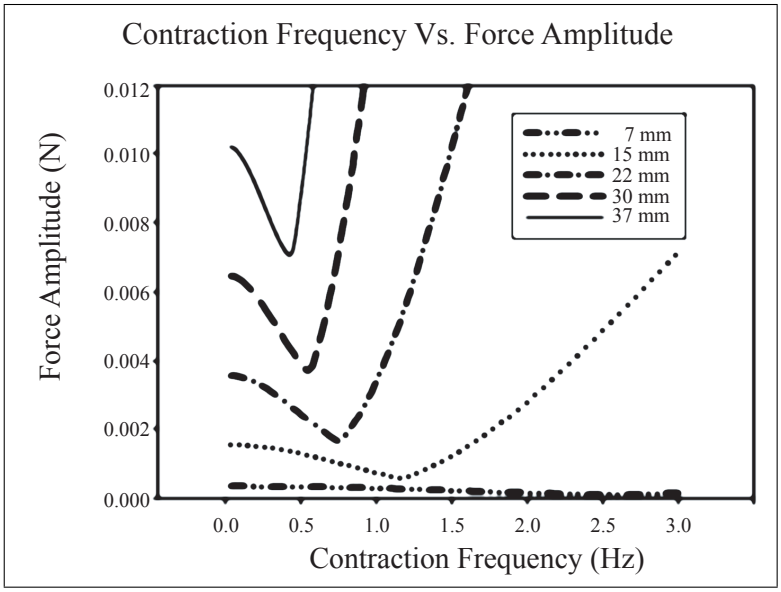

Figure 6.13: The relationship between the contraction frequency of a jellyfish and the amount of force needed to maintain that frequency at a predetermined amplitude. These results are given by Megill's theoretical model of jellyfish [65]. This graph shows the amount of force that the circumferential muscle must exert for several different umbrella heights.

frequency of real jellyfish in their resonant gait that were observed by Dabiri and Gharib [22]. Specifically, Figure 6.10 shows the contractions of a Chrysaora fuscescens with umbrella heights between 50 and 100 mm that were captured by Dabiri and Gharib. It has a contraction frequency of approximately 0.8 Hz. Although Megill's results only show an umbrella as tall as 37 mm, his trends clearly show that a larger umbrella would have to exert a large amount of force to maintain a resonant frequency of 0.8 Hz. Whether this discrepancy is due to methodologies, or perhaps a difference in the studied species, we are not sure. Without any further confirmation of either result, we are left to experiment with a contraction frequency that will yield the most thrust for our model. More will be discussed on this matter in Section 8.2.1 when we present the results of our jellyfish model.

CHAPTER 7

NUMERICAL MODEL OF JELLYFISH

We build a numerical model of jellyfish that accounts for the biological aspects of the organism, which we introduced in Chapter 6. We wish to mimic the physiology of jellyfish in our model as best we can, so that our animations of jellyfish are qualitatively similar to observed behavior of the real organism. We use spring-mass systems that were described in Chapter 2 to represent the body of the jellyfish, and we account for the sea-water that surrounds the organism with the fluid simulation methods discussed in Chapters 4 and 5. In our model, we exploit the axial symmetry that is exhibited by some species of jellyfish. We simulate a 2D slice of the organism, and then extrapolate the results of that slice to 3D. In fact, Dabiri and Gharib [22] exploited the symmetry of the organism in a similar fashion, although their model takes the process further and only considers one half of the slice, which they mirror to get the other half. The results of our 3D extrapolation will be a volume whose surface is quite smooth and artificially perfect. Thus, we add variation to the umbrella and the tentacles by adding various noise functions to the extrapolated 3D surface. The process is illustrated in Figure 7.1. The following sections outline the details of our model.

Our final animations of jellyfish may be generated at arbitrary resolutions, yet our physics model is relatively coarse. Thus, we need a means of building a rendering model that resamples the geometry our physics model so that it is at a resolution that is appropriate for our rendering needs. We do this resampling on the 2D slice data, before extrapolating and adding noise, so that these two steps are performed on a model that is an appropriate resolution for our rendering configuration.

7.1 2D Simulation Model

We start by building a spring-mass system to represent a jellyfish. We have three physical aspects of the jellyfish's anatomy that we wish to capture in our model. We want to represent the umbrella's mesoglea itself. To represent the umbrella's mesoglea, we could build a mesh of linear springs to represent the volume. The left part of Figure 7.2 shows a mesh-like representation of the mesoglea. The mesh is composed of contiguous quadrilaterals whose edges are springs. Each quadrilateral has diagonal springs to keep the volume from collapsing. The diagonal springs are allowed to cross, and are independent of each other. This mesh representation of the mesoglea may seem sturdy, but it invites issues of numerical instability into our simulations. The instabilities of a purely linear mesh seem to be due to the the large number of short springs

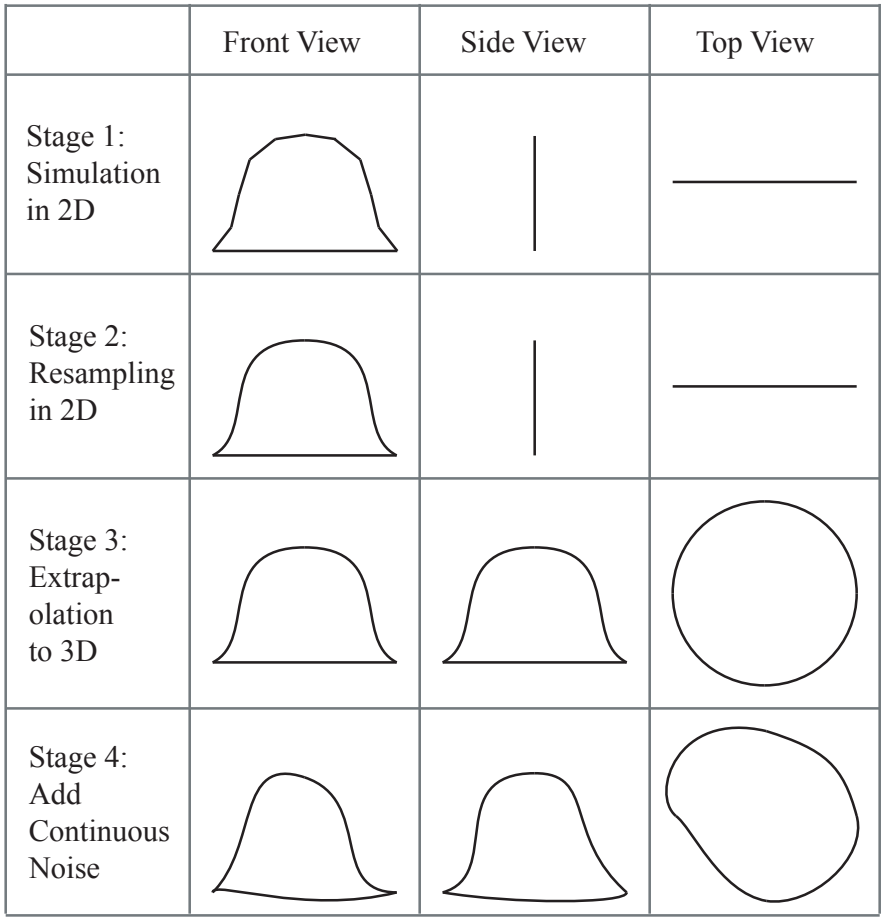

Figure 7.1: Steps in exploiting the symmetry of a jellyfish.

and point-masses. Not only do the extra springs increase the number of force calculations that must be done, but they make the numerical system more complicated and harder to predict. If forces are too strong, then the chances of the mesh collapsing are increased with such a complicated network of points.

An alternative means of enforcing the structure of the umbrella is to use angular springs. We can use a single string of points to represent the gross geometry of the umbrella. Each point in the string is connected to its neighbors by both linear and angular springs, to enforce relative distances and orientations. Doing so simplifies the dynamics of the system. The right side of Figure 7.2 illustrates the simplified system involving angular springs. Experimentally, we find that the simpler model involving angular springs and fewer linear springs is more structurally stable than a model that relies solely on linear springs. However, if we were using an SPH-based fluid simulator, then our collision detection method described in Section 5.2.2 would require that the elastic model have some finite volume. Thus, the model involving purely linear springs would be preferred because the spring-mass model would then have a finite volume against which collisions could be more reliably detected.

Figure 7.2 also shows linear springs going longitudinally across the umbrella. We use these springs to represent the circumferential muscle that lines the inner surface of the umbrella. Each spring represents a

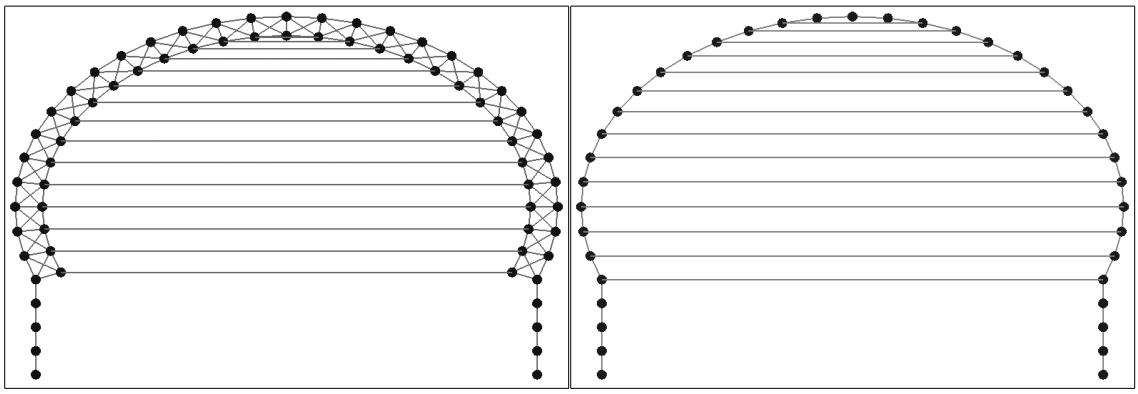

Figure 7.2: Different spring-mass representations of an jellyfish umbrella. The left image uses a relatively dense point mesh, connected solely with linear springs. The right is a simpler network of points that are all connected with both linear and angular springs along the umbrella. linear springs also go between the two sides of the umbrella, and are used to represent the circumferential muscle on the underside of the umbrella. Lastly, strands of points on either side of the umbrella are used to represent the tentacles of the organism. These points are also connected with linear and angular springs (for both the left and right images).

cylindrical area of the circumferential muscle that is oriented longitudinally along the umbrella. We reduce the rest lengths of these springs to cause the umbrella to contract. However, these springs are used only as an approximation of how the full 3D umbrella would react. In reality, the circumferential muscle fibres will pull tangentially on their region of the umbrella, but the net force of all of these muscle fibres will be normal to the inner surface of the umbrella. Our springs that cross the two sides of the umbrella approximate that normal force. Thus, our subumbrellar springs are used to approximate the net normal forces from the circumferential muscle. The subumbrellar springs should have no effect on the fluid within that region of the 2D slice. Luckily for us, the immersed boundary method described in Section 4.3.2 exerts forces only at the location of the point-masses, and the springs themselves have no direct effect on the fluid. With our SPH system in Chapter 5, we merely have to ensure that we ignore any collisions between these springs and the fluid particles.

We have to choose coefficients of elasticity κ_l and κ_θ to define the resistance of our model to deformation. We can turn to literature in experimental biology to suggest appropriate coefficients. Megill [65] measured the elastic strength of the bell mesoglea (including the uncontracted radial muscle fibres). He found that the mesoglea has an elastic modulus of approximately 1186 Pa, and similarly the joint mesoglea to have an elastic modulus of 130 Pa. The modulus for the muscle fibres is significantly higher, at around 400,000 Pa. The elastic modulus λ is related to the elastic coefficient κ_l for linear springs in the following way:

$$\lambda = \frac{\kappa_l}{V} = \frac{\kappa_l}{l_r A}, \tag{7.1}$$

where V is the volume of the elastic body being represented by the spring. In our case, this volume is the rest length l_r of the spring and the cross-sectional area A that the spring is to represent. If the spring is tangential

to the umbrella (i.e. representing the mesoglea), then we approximate the area with the square of the umbrella's thickness ϑ. If the spring joins the two sides of the umbrella (i.e., representing the circumferential muscle), then we use the square of the average distance between this spring and its two nearest neighbours.

Figure 7.2 shows one final physical feature of the jellyfish that we wish to simulate, namely the tentacles. These features show up as the two strands of points on either side of the umbrella in Figure 7.2. We connect these strands of points with both linear and angular springs. We could not find a general notion of the size and makeup of jellyfish tentacles. This sort of information can vary widely among different species, as shown in Figure 6.3. Thus, we artistically choose the physical parameters of the tentacles. We use the same elastic modulus as the bell mesoglea, but a much smaller volume ($1/100^{th}$) than the umbrella springs.

In order to contract the umbrella, we modify the rest lengths of the subumbrellar springs. We attempted to define a simple closed-loop controller like the one described in Section 1.2.5. This controller would assign each subumbrellar spring a rest length based on a goal state and a time at which we wanted to reach that goal state. The idea was to have the controller make the organism fully contracted (for example) by a certain time. If a spring is behind schedule to make the contraction happen on time, then we decrease the rest length of the spring. So, if the target state is a goal contraction amount of γ_g at time t_g, and the current time and contract amount are t_c and γ_c, then we change in the spring's length l_r as follows:

$$\Delta l_r = \alpha\Big(\frac{d\gamma_c}{dt} - \frac{\gamma_g - \gamma_c}{t_g - t_c}\Big) \tag{7.2}$$

Where again α is a sensitivity parameter that tunes the controller's response to error. Extremely large values of α cause large oscillations in the spring's rest length, and consequently motion is erratic and the elastic forces could become arbitrarily large. With moderate values of α, the controller keeps the end-points of the spring moving at essentially a constant velocity when they are in transition between goal states. This constant velocity looks too artificial. Small values of α give the points a smoothly curved position profile somewhat similar to what is observed in real jellyfish, as was shown in Figure 6.10. However, because the controller is too forgiving about discrepancies between the goal state and the current state, the points are chronically late for their arrival times. Despite the simplicity of our controller, we abandoned the use of controllers in general because the biology literature does not provide evidence that the organism uses controllers of this type to induce muscle contractions. Instead, we suspect that jellyfish contractions are governed by a cyclical pattern that is quite predictable [101, 65]. Rather than using closed-loop control processes, we turn to open-loop controllers. Specifically, we make use of analytical functions to determine the rest-lengths of the springs within our jellyfish model.

We are simulating the resonant gait of the organism, and both Megill [65] and Dabiri and Gharib [22] show us that the motion of jellyfish using this gait is highly cyclical. However, we have no notion of how to control the contraction of the circumferential muscle to produce the resonant gait. We are left to artistically define a periodic function that controls the rest length of the subumbrellar springs. From looking at Figure 6.10, one might suspect that a suitable contraction function would be a skewed sine wave. Our initial trials involved simple functions like sine waves and triangle waves. However, the resulting

jellyfish morphology was far from the captured data in Figure 6.10. Later attempts on our part to define a suitable contraction function involved a short Fourier series: we summed up the results of several sine waves that had different amplitudes and phase offsets. Despite the fact that these combined sine waves created a skewed wave as we had intended, this contraction function did not produce morphology that resembled Figure 6.10. We then began to consider parametric curves which allow for more control over the behaviour of the contraction function. One such curve that we tried is the modified *trochoid* that was used by Fournier and Reeves [34] to model ocean waves. This trochoid resembles deformed sine waves, and has parameters specifically to control the skew and the relative widths of the peaks and troughs of the wave. However, these trochoid curves also did not prove exceptionally fruitful. As a final approach, in an effort to make our contraction function easier to manipulate, we later defined a curve based on Hermite splines, with artistically chosen points and velocities to be interpolated.

Briefly, a Hermite spline is an interpolant that has four geometric constraints: the two end-points $\vec{x}_b$ and $\vec{x}_e$ between which we want to interpolate, and two "velocity" vectors $\vec{v}_b$ and $\vec{v}_e$ which are derivatives of the curve at the two end-points. We want to compute an interpolated point $Q_H(s)$ at some parametric location $s \in [0,1]$. The trivial cases are $Q_H(0) = \vec{x}_b$ and $Q(1) = \vec{x}_e$. The definition of the Hermite interpolant is:

$$Q_H(s) = (2s^3 - 3s^2 + 1)\vec{x}_b + (-2s^3 + 3s^2)\vec{x}_e + (s^3 - 2s^2 + s)\vec{v}_b + (s^3 - s^2)\vec{v}_e. \tag{7.3}$$

The derivation of Equation 7.3 are given by Foley et al. [33]. For shorthand, we will denote Equation 7.3 as

$$Q_H(s) = f(s, \vec{x}_b, \vec{x}_e, \vec{v}_b, \vec{v}_e). \tag{7.4}$$

The Hermite interpolant has only first-order continuity, and only between the two given points that it interpolates. We can force a curve to go through several points $\vec{x}_0, \vec{x}_1, ..., \vec{x}_n$ by joining several Hermite curves $Q_{H_0}(s), Q_{H_1}(s), ..., Q_{H_{n-1}}(s)$ together, such that $Q_{H_i}(s) = f(s, \vec{x}_{b_i}, \vec{x}_{e_i}, \vec{v}_{b_i}, \vec{v}_{e_i})$. To ensure continuity of the curve, we have to set $\vec{x}_{b_i} = \vec{x}_i$ and $\vec{x}_{e_i} = \vec{x}_{i+1}$. If we wanted to maintain continuity of the first derivative as well, we would have to define end velocities $\vec{v}_0, \vec{v}_1, ..., \vec{v}_n$ such that $\vec{v}_{b_i} = \vec{v}_i$ and $\vec{v}_{e_i} = \vec{v}_{i+1}$. However, if we do not need the extra order of continuity, then we can define the velocities arbitrarily. Whatever the continuity of our curve, it is being used to indirectly manipulate the force (and thus the acceleration) of our particles. Thus, we maintain two extra orders of continuity in our particle positions. Because of this extra continuity, the Hermite spline is continuous enough for defining our contraction function. However, we will need other interpolants with higher continuity when we define our rendering model in Section 7.2. Figure 7.3 shows the contraction function for our subumbrellar springs that we generate with Hermite spline patches, along with the geometric constraints that we use to generate the curve.

Whatever contraction function we choose, we normalize the function to be between 0 and 1. We then define a parameter γ that represents the amount by which the umbrella is to contract. As was seen in Figure 6.7, a good choice for γ is 0.56. Then, if l_{r_u} is the uncontracted rest length of the spring, we set the contracted rest length to be

$$l_r = l_{r_u}\big(\gamma + (1-\gamma)Q_H(s)\big). \tag{7.5}$$

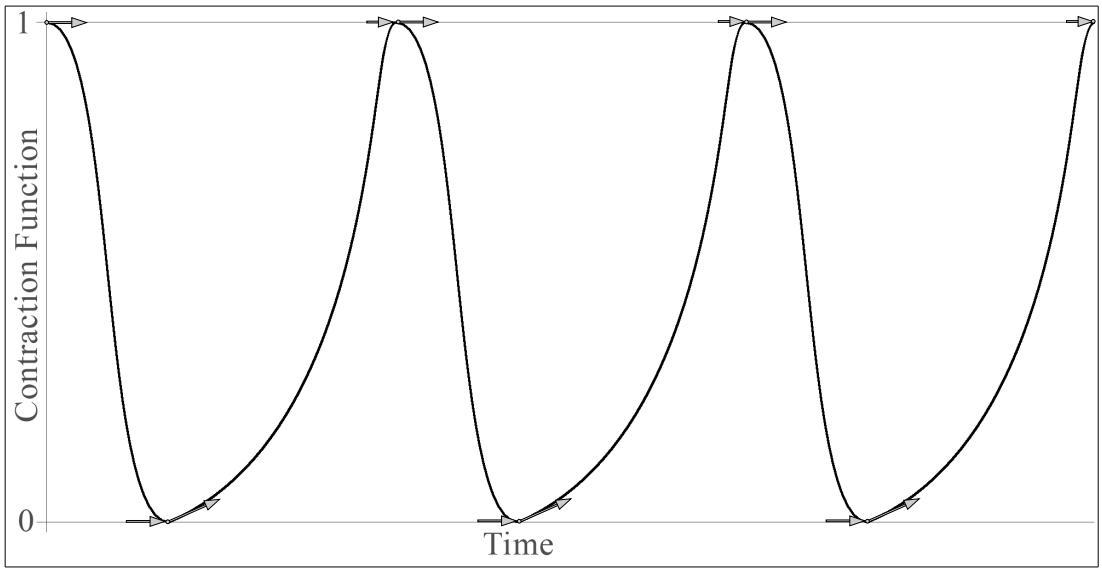

Figure 7.3: Three cycles of our simple contraction function based on Hermite spline patches.

When the jellyfish contracts, it will propel itself forward. However, when it expands, it also pulls itself backward. For jellyfish to achieve a net positive acceleration over the course of a contraction cycle, the organism must incur less drag on the expansion phase of the cycle than it does in the contraction phase. We are unaware of any biology literature that would suggest exactly how this drag reduction is achieved by the organism. However, we have our hypotheses. First, let us look at Figure 7.4, which is a short image sequence of an actual jellyfish that expands, and then contracts. In the first three images of Figure 7.4, the jellyfish is expanding, and its umbrella shape (with respect to a side view) is relatively curved. However, in the fourth frame of Figure 7.4, the jellyfish is contracting, and the umbrella appears less curved and resembles a cone shape. This conic resemblance is especially striking toward the aperture of the umbrella.

Figure 7.4: A series of frames from actual jellyfish footage. The first three frames show the expansion phase of the resonant gait, and the last two frames show contraction phase. Clearly, the umbrella is more bowl-shaped during expansion, and more cone-shaped during contraction.

In order to achieve the sort of morphology that we see in Figure 7.4, we have to slow the expansion of springs that are at the bottom of the umbrella, relative to those at the top. We give each subumbrellar spring a unique contraction function in which springs close to the aperture expand more slowly than the those closer to the apex of the umbrella. Figure 7.5 again shows our spring-mass model of the jellyfish, but with the

corresponding contraction functions for each spring.

We also take into account the relative delay in muscle activation that occurs in different regions of the umbrella. This delay is caused by the time needed for synaptic impulses to travel across the umbrella, and was described in detail in Section 6.2. We can simulate this delay by giving a phase offset to our contraction function for that spring, effectively shifting the function horizontally along the time axis. Since we are simulating an entire cylindrical area of the umbrella with a single spring, we merely give a single offset to the entire spring, and the value of that offset is the average delay of the region that the spring represents. The neuronal impulse travels with a constant velocity, and so the delay profile with respect to a thin longitudinal region of the umbrella is triangular. We approximate the phase offset of a particular spring to be half of the maximum delay in that cylindrical region of the umbrella.

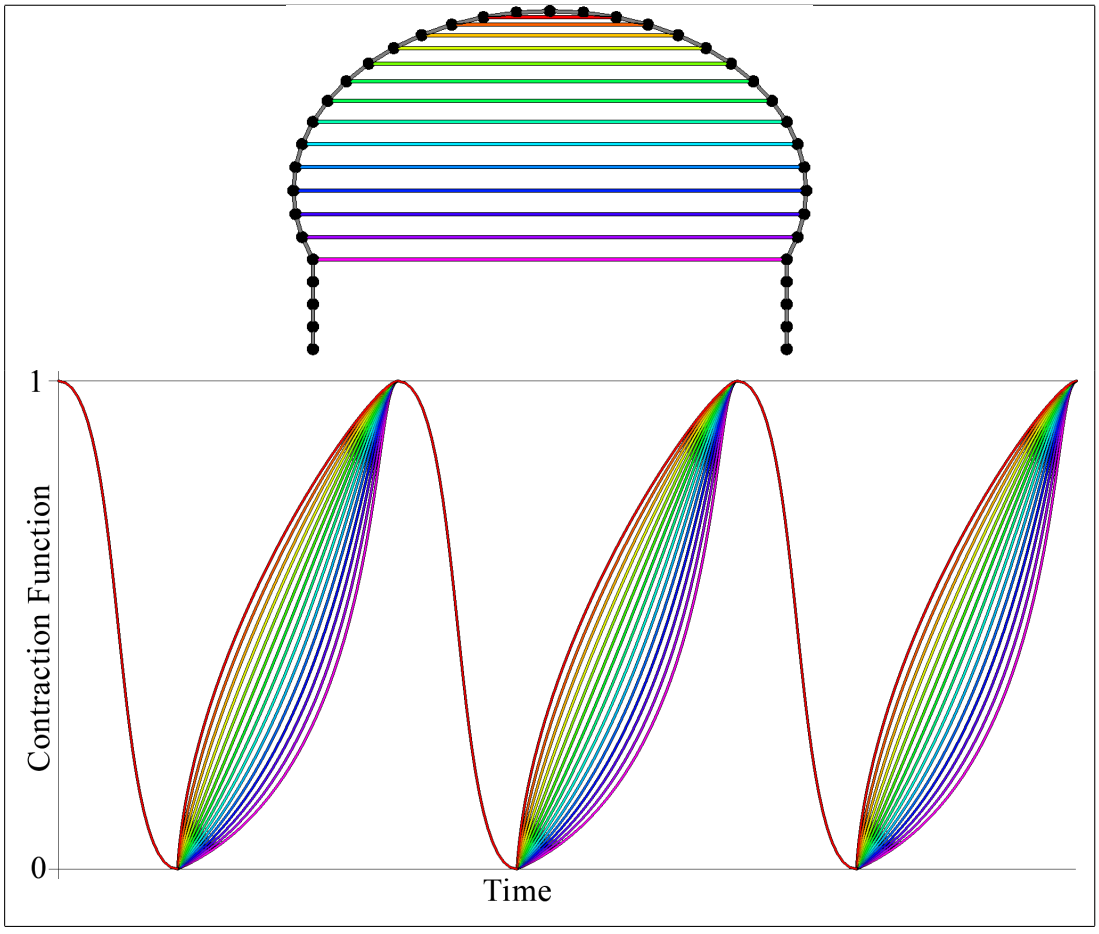

Figure 7.5: Our spring-mass network for a jellyfish slice, and the corresponding contraction functions that we use. The subumbrellar springs in the left image are colour-coded, and the contraction function for a spring is graphed in the right image with the same colour. Note that the graph of the contraction functions has regions where only a red function was plotted. In these parts of the graph, all of the contraction functions coincide with the same values and overlap each other.

Finally, we need to account for the sea water that surrounds the organism. We found the SPH-based approach that was described in Chapter 5 to be too computationally expensive. Instead, we use grid-based methods. We immerse our spring-mass model of the jellyfish in a square fluid volume that is ten times the jellyfish's diameter in width. For boundary conditions, we initially thought that periodic boundaries would be ideal because this type of configuration would more closely approximate the open sea. However, we found that numerical error in our simulation caused a current to occur, and the periodic boundaries seemed to accentuate that drift. We instead chose to use free-slip boundary types, so that the flow would be prevented across the boundaries, but tangential flows along the boundary would be allowed.

7.2 2D Rendering Model

We want to distinguish between the geometry of our physical model that was described in Section 7.1 and the geometry that we want to render. Our physics model is rather coarse, and if we were to simply render a close-up view of the springs as straight lines, we would be able to see artifacts due to the low-order continuity between adjacent springs. The opposite argument applies when we are rendering jellyfish that are far away from the camera of our rendering system, and the distance between points would be a fraction of a pixel's width. In either case, we will want to resample the physics model to be at a resolution that is appropriate for the rendering configuration. We also need a way to account for the thickness of the umbrella and the tentacles, since we are modeling both as piecewise linear curves. For all these reasons, we have created a separation between our physical model and the rendering model which we actually rasterize to the screen. This kind of separation between the simulation and the rendering is not new to the graphics community, and has been investigated by Müller and Gross [78], as well as by Bouthors and Nesme [5].

We begin by adding a finite thickness to our model of the umbrella. In our simulation model that involved both linear and angular springs, the points of the umbrella represent the subumbrellar surface, but we have no representation of the exumbrellar surface. We can generate the exumbrellar side by pairing each subumbrellar point $\vec{x}_i$ with another $\tilde{x}_i$ that is shifted outward, away from the subumbrellar cavity. The actual direction to which the point is shifted is the surface normal $\vec{n}_i$ of the umbrella at that location. We approximate these normals numerically by using the two adjacent points $\vec{x}_{i-1}$ and $\vec{x}_{i+1}$. More sophisticated means of recovering surface normals have been developed [39], but we find our linear approximation to be adequate for our purposes. We get the following expression for our normal:

$$\vec{n}_i = (\vec{x}_{i+1} - \vec{x}_{i-1}) \times \vec{z}, \qquad \text{where } \vec{z} = (0,0,1). \tag{7.6}$$

Our jellyfish umbrella will have a thickness of ϑ, and Megill [66] gives us some idea of how thick the umbrella of the organism should be. He reports a thickness that is 5.4% of the height of the umbrella. Our expression for $\tilde{x}_i$ is then

$$\tilde{x}_i = \vec{x}_i + \vartheta \hat{n}_i. \tag{7.7}$$

The hull of the physical model is then defined by the ordered set of points $\vec{x}_1, \vec{x}_2, ..., \vec{x}_n, \tilde{x}_n, \tilde{x}_{n-1}, ..., \tilde{x}_1$.

We can generate an arbitrary resolution model for rendering by interpolating a spline through the above set of points. We stick to a relatively low-order spline to avoid the high-frequency oscillations that can appear when using higher-order splines. The specific spline we use to interpolate between points $\vec{x}_i$ and $\vec{x}_{i+1}$ is the following:

$$Q_c(s) = \vec{P}s^3 + \vec{Q}s^2 + \vec{R}s + \vec{W} \tag{7.8}$$

where

$$\begin{aligned}
\vec{P} &= (\vec{x}_{i+2} - \vec{x}_{i+1}) - (\vec{x}_{i-1} - \vec{x}_i), \\
\vec{Q} &= (\vec{x}_{i-1} - \vec{x}_i) - \vec{P}, \\
\vec{R} &= \vec{x}_{i+1} - \vec{x}_{i-1}, \\
\vec{W} &= \vec{x}_i.
\end{aligned}$$

With this interpolation, we have a complete rendering model for the 2D slice of the jellyfish.

7.3 3D Extrapolation

As seen in Figure 6.3, some species of jellyfish are roughly symmetric axially about the axis that goes through the top of the umbrella to the center of its aperture. We take advantage of this symmetry by simulating a 2D slice of the organism. We choose the slice plane such that the organism's axis of symmetry lines on the plane. The resulting 2D data must then be extrapolated to 3D. However, our 2D slice may not be symmetric itself (i.e., the two sides of the umbrella may be different). Thus, we cannot extrapolate the geometry by merely rotating the results of the 2D slice about the axis of symmetry. We must instead interpolate radially between the two sides.

Our radial interpolation proceeds by considering each pair of points $\vec{x}_i$ and $\vec{x}_j$ that are opposite to each other on the umbrella. On our simulation model that was described in Section 7.1, these opposing points would be joined by a subumbrellar spring. However, since we have changed the sampling of our model for rendering, as was discussed in Section 7.2, we must use opposing pairs that resulted from the resampling. We define an axis of rotation for each pair, which goes through the center of the line segment between the two points $\vec{x}_i$ and $\vec{x}_j$, and is orthogonal to that line segment. Note that each pair of points will potentially have a different axis. Rotating about that axis, we get a disc on which we define our 3D geometry of the umbrella. We can place points uniformly about the circumference of each disc, and then stitch polygons between two adjacent discs to get a surface for the umbrella hull. We use the same disc extrapolation scheme for generating tentacles along the 3D aperture of the umbrella, except that we only connect vertically adjacent points with line segments, rather than creating polygons between neighbours in both directions. Figure 7.6 illustrates our process of extrapolating circular area from pairs of points in our 2D model.

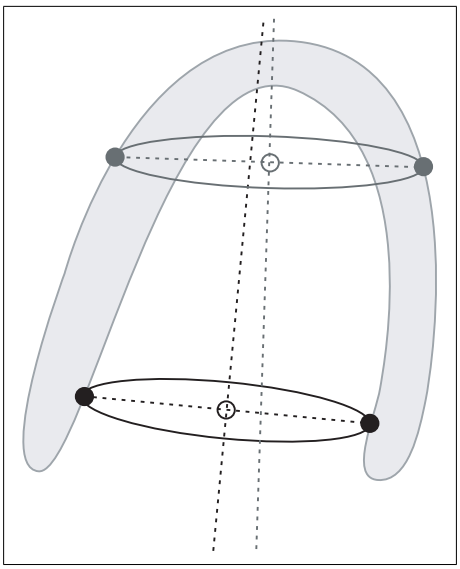

Figure 7.6: An example of point pairs along the 2D umbrella slice that have been extrapolated to 3D discs. For each point pair in the illustration, the line between them is drawn, along with the axis of rotation that bisects the line, and the disc that results from rotating those points about that axis.

The volume that will result from our disc-based extrapolation will be artificially smooth and perfect. Although a given species may be characterized as roughly symmetric, an individual jellyfish will likely not be exactly symmetric. Figure 7.7 shows a couple of example individuals whose umbrellas are approximately symmetric, but with some small details that are not exactly symmetric. Several factors can cause these small-scale asymmetries. For one, variation can be due to features in the underlying geometry of the umbrella itself that are particular to the species of jellyfish. For example, the Halopsis ocellata (as was seen in Figure 6.3) exhibits a ripple pattern across the umbrella's surface. Gladfelter [37] also showed us that jellyfish deform nonuniformly when they contract their umbrellas. This nonuniform elasticity caused sinusoidal ripples to form across the umbrella as it contracts, as was illustrated in Figure 6.7. As well, more unstructured variation can exist in the overall shape of the umbrella, just as different members of the human race have different facial features. In this case of unstructured asymmetry, the variation can be dependent of the phase, amplitude, or frequency of the organism's contraction (or any combination of the three). Figure 7.7 shows two different examples of jellyfish that exhibit small-scale asymmetries.

This type of approximate symmetry has been exploited in other fluid simulations in computer graphics literature, most notably in particle-based animations of explosions by Rasmussen et al. [91]. Like our work, theirs simulated a system that is roughly symmetric, but that has fine details that are not symmetric. Rasmussen et al. [91] also simulated their system as a 2D slice, and then rotated the results, and added variance to the extrapolated 3D data. In their case, a 3D periodic noise spectrum was used to directly modify the 3D velocity field of the fluid. In our method, we only perturb the geometry of our umbrella, and do not attempt to modify the underlying fluid field.

Figure 7.7: Jellyfish with irregular shapes, specifically with small-scale asymmetries. The left image is courtesy of Professor Mark G. Eramian of the University of Saskatchewan, while the right image was taken by the staff at the Florida Keys National Marine Sanctuary [95].

To add variation to the umbrella of our rendering model, we perturb each point $\vec{z}_{i,j}$ on the surface by some scalar distance $c_{i,j}$ in the direction of its surface normal $\vec{n}_{i,j}$. We use an extra subscript in our notation for the points because we are now dealing with points on two different axes. The first axis is the latitude of the point, which is the position relative to its counterparts on the same 2D slice. The second axis is the longitude, which is relative to the position along the 3D-extrapolated disc. Just as in Section 7.2, we compute the surface normals numerically. However, we are now dealing with 3D points, and so the normal is approximated as follows:

$$\vec{n}_i = (\vec{x}_{i+1,j} - \vec{x}_{i-1,j}) \times (\vec{x}_{i,j+1} - \vec{x}_{i,j-1}). \tag{7.9}$$

For periodic variation due to the raw structure of the umbrella, like the corded appearance of the Halopsis ocellata in Figure 6.3, one could define simple trigonometric functions to generate the desired appearance, such as

$$c_{i,j}^{structural} = \vartheta sin(f\sigma_j), \tag{7.10}$$

or

$$c_{i,j}^{structural} = \vartheta |sin(f\sigma_j)|, \tag{7.11}$$

where ϑ is the thickness of the umbrella as was discussed in Section 7.2, f is some artistically chosen frequency that the variation should have, and σ_j is the longitudinal angle of points $\vec{x}_{i,j}$ for all values of i. This angle can be expressed as:

$$\sigma_j = 2\pi \frac{j}{j_{max}}. \tag{7.12}$$

Another periodic variation in the umbrella hull occurs because of the nonuniform elastic properties of the umbrella itself, as was illustrated in Figure 6.7. As the umbrella contracts its overall radius, the exterior surface of the umbrella begins to show a sinusoidal pattern across it. We can account for this variation

by considering the amount of contraction that the umbrella is undergoing at that location. We can get this information from the subumbrellar springs in our physics model by looking at the spring's uncontracted rest length l_r and its current length l_c. Essentially, we want to design a contraction coefficient that is at 1 when $l_c = l_r$, and 0 when $l_c = \gamma l_r$. Again, γ is the maximum amount by which the jellyfish muscles will contract, as was discussed in Section 7.1. This contraction coefficient will be of the form

$$\frac{l_c - \gamma l_r}{l_r - \gamma l_r}. \tag{7.13}$$

We also need an expression for which part of the umbrella experiences the most expansion due to compression of the mesoglea, for which we can use the following:

$$\frac{1}{2} sin(\sigma_j) + \frac{1}{2}. \tag{7.14}$$

The $\frac{1}{2}$ constants are in place to normalize the function between 0 and 1. The overall displacement of the umbrella hull can then be described as:

$$c_{i,j}^{compressed} = \vartheta\Big(\frac{l_c - \gamma l_r}{l_r - \gamma l_r}\Big)\Big\{0.34 + 0.17\Big(\frac{1}{2} sin(\sigma_j) + \frac{1}{2}\Big)\Big\}. \tag{7.15}$$

The constants $0.34 = 1.34 - 1$ and $0.17 = 1.51 - 1.34$ come from Figure 6.7. Note that only points on the exumbrellar surface are affected in this manner. We do not modify the points along the inside of the umbrella, at least not because of the compression of the mesoglea.

As for asymmetries that are not periodic, we do not have any statistics on the types of shapes and patterns that jellyfish exhibit. One might imagine that this variation is due to genetic differences between individuals of the same species. The asymmetries may also be caused by action on the part of the radial muscle fibres that were described in Section 6.2. Without a better understanding of why these differences occur, we are left to mimic this variation artistically by adding noise to the surface of the umbrella. However, we cannot simply add independent random displacements to the surface points, or the umbrella would appear to have sharp discontinuities. Instead, we make use of continuous noise functions, which ensure that the noise values have at least a couple of orders of continuity. We choose to use Perlin's noise function [84, 86, 85], which yields a second-order continuous scalar field $Q_P(x,y,z)$ over a 3D space. Perlin's noise function can most easily be described as a regular grid of random values and their derivatives, which are interpolated with a second-order spline. However, much care must be taken when choosing the random values to be used, as is discussed in more detail in the literature [86, 85]. Perlin's noise function is often associated with fractal geometry, in that multiple octaves of the same noise function are often added together in such a way that the higher octaves have decreasing amplitude. In our work, we find that just one octave of Perlin's noise function creates enough variation for our purposes. Several versions of Perlin's noise function exist, and we use his most recent version [85]. Other continuous noise functions, such as the wavelet noise function that was developed by Cook and DeRose [18], could conceivably be used instead of Perlin noise.

To make use of Perlin noise, we have to parameterize the points $\vec{x}_{i,j}$ on the umbrella hull. We do so by defining the discrete distance that each point has from the the peak of the umbrella (i.e., how many points

away from the peak it is). This distance is with respect to either the peak of the inside of the umbrella or the outside. Thus, if i_{out} is the discrete latitude of the outside peak of the umbrella, and i_{in} is the latitude of the inside peak, then the discrete distance d_i of a point $\vec{x}_{i,j}$ is simply:

$$d_i = \frac{min(||i - i_{out}||, ||i - i_{in}||)}{i_{max}}. \tag{7.16}$$

We define a similar distance metric for the longitudinal axis of our umbrella, except that it considers the discrete distance from the 2D slice, as below:

$$d_j = \frac{min(j_{max} - j, j)}{j_{max}}. \tag{7.17}$$

The third component that we give our noise function is time. We want the geometry to evolve slowly over time.

We control the frequency of each parameter to the noise function independently. This is done because different species display different frequencies of noise in the lateral and longitudinal axes. Again, we do not have data on what these frequencies should be, but instead we artistically chose parameters to qualitatively resemble the look of a target species. Lastly, we also attenuate the amplitude of the noise function for points that are near the peak of the umbrella (both on the exumbrellar and subumbrellar sides of the volume). The discs that we extrapolate from our slice data are smaller near the peak of the umbrella than the discs that are closer to the aperture. Thus, the noise function appears to have a higher frequency on the discs with smaller radii. By attenuating the noise function near the top of the umbrella, we remove these high-frequency artifacts.

Our final expression for the displacement of the umbrella points due to non-periodic variation is as follows:

$$c_{i,j}^{unstructured} = d_i * Q_P\big(2d_i, 8d_j, 0.01t\big). \tag{7.18}$$

We apply this displacement to both umbrella points, and also tentacle points, so that the tentacles themselves exhibit some variation. Figure 7.8 shows our model after 3D extrapolation, both with and without noise added to the surface.

7.4 Final Rendering

Once we have a 3D model with the appropriate spatial resampling, we then render a frame of the animation. We use the OpenGL graphics library with conventional graphics hardware to render the final geometry of the jellyfish. The methods in Section 7.3 yield a 2D array of points that are roughly arranged along latitudinal and longitudinal lines of the umbrella. We can render this geometry as a series of quadrilaterals using conventional scan conversion techniques [33]. We can even decompose the quadrilaterals into triangles, which is generally more efficient to render [116]. Although real jellyfish are generally translucent, we have not investigated the optical properties of the organism, nor have we implemented any rendering systems that

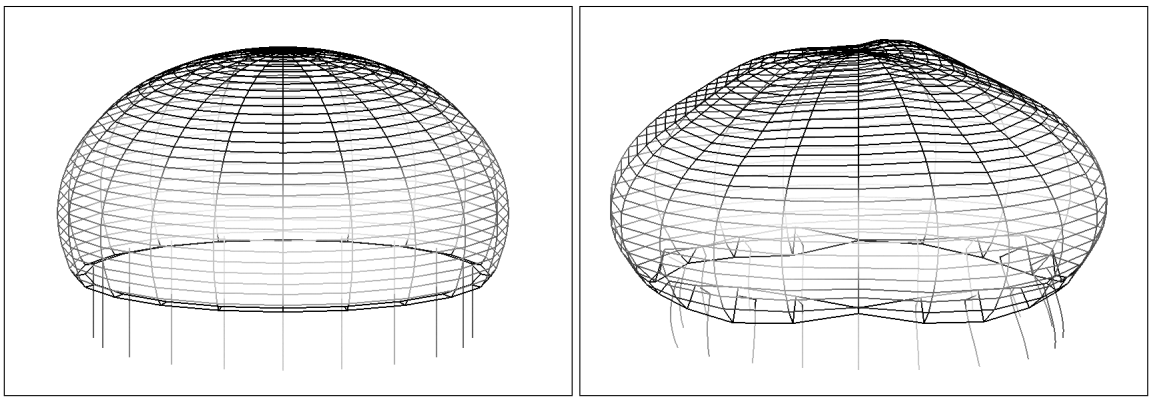

Figure 7.8: Wireframes of our 3D jellyfish model. The left image is the result of extrapolating to 3D without adding variation to the model. The right image is the same model with noise added to it.

handle translucent volumes. We instead treat the surface of the organism as opaque. Thus, we can render the quadrilaterals in arbitrary order, and a depth buffering algorithm [106] is used to determine which portions of each triangle are in the foreground and thus are visible. However, we can make more efficient use of the OpenGL rendering pipeline by rendering adjacent quadrilaterals consecutively [116]. Thus, we render our 3D hull in horizontal strips.

To add a sense of depth and shading to the scene, we illuminate the model with the conventional three-term lighting model [33], which is built into OpenGL. We merely provide the graphics interface with sets of quadrilaterals, and the surface normal of the quadrilaterals. These normals are again computed numerically, as they were in Section 7.3. However, we cannot use the same normals that were computed in Section 7.3, because they apply to the perfectly round umbrella hull. Since we have added noise to the 3D model, the surface orientation will generally have changed. We simply need to compute new numerical approximations to the surface orientations.

CHAPTER 8

RESULTS

The method we have described comes with some strengths and weaknesses. The type of simulation we employ is not physically accurate, but it does yield plausible animations of jellyfish locomotion that contain all of the main aspects of the organism's locomotion. We are also afforded some extra latitude in simulating jellyfish because most humans rarely observe jellyfish in motion. Thus, our audience is generally less adept at recognizing the visual aspects of this organism's locomotion. By contrast, animators of human characters must be extremely vigilant to capture the subtleties of our motions. We as a society spend a large part of our day observing and interacting with other humans, and so we have an incredible understanding of our own style of locomotion, at least in a visual sense.

Aside from the jellyfish model itself, we show some testing and performance metrics of our methods. In Chapters 2, 4, and 5, we discussed a number of methods to simulate physical systems. In this chapter, we compare the results of those methods in general, and expose the advantages and disadvantages of each method. We discuss numerical issues that we discovered specifically with our model of jellyfish. We also investigate ways of improving the speed and reducing the stiffness of our jellyfish animations, though at the cost of physical accuracy.

8.1 Numerical Results

With regard to the methods that we use, we investigate the numerical issues that arise and our means of handling those issues. The following sections discuss our experimental findings with respect to the numerical properties of the methods that we tried. We discuss all aspects of our system: the spring-mass simulation of elastics, integration of the particle-based SPH fluid method, and the grid-based Eulerian and semi-Lagrangian fluid methods.

8.1.1 Spring-Mass Systems

Chapter 2 explained basic elasticity theory and the use of spring-mass systems for simulating elastic bodies. Specifically, we simulate an elastic body with a series of point-masses, which are connected with a mesh of springs. We discuss two types of springs: linear springs that aim to enforce a certain distance between two points, and angular springs that attempt to enforce a certain angle between three points. Although

Chapter 2 did discuss spring-mass systems and numerical methods for simulating those systems, most of the discussion was well-known to the scientific community. Also, our work with jellyfish does not utilize spring-mass systems by themselves, but in combination with fluid simulators. . Figure 8.1 shows a short animation of an elastic bar swinging from a single point. This animation was simulated with our spring-mass system, and easily runs in real-time on modest hardware, so long as we use a somewhat sophisticated integration method like the backward Euler or Dormand-Prince methods. Note how the shape of the elastic strand bows in alternating directions, as a real rope would as it swung from a fixed point.

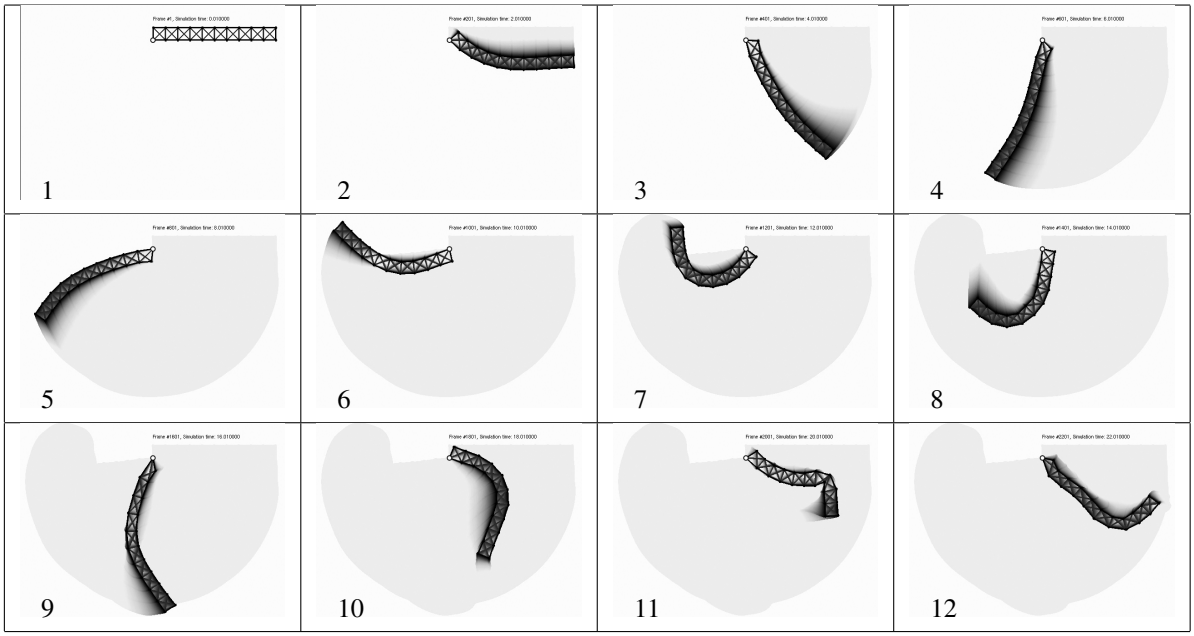

Figure 8.1: Sequential frames from a simulation of a simple spring-mass system. The system has one suspended point which is shown as a white circle from which the mesh swings under the force of gravity. Blur lines of the elastic body are shown to give a sense of relative velocities. The initial length of the elastic strand is 0.45, and the initial width is one tenth of that (0.045). The mass of each particle is $m_i = 24$ and the linear spring constant for each spring is $\kappa_l = 75m_i = 1800$. A gravity vector of $\vec{g} = (0, -0.05, 0.0)$. The Dormand-Prince method is used with a constant step size of $\Delta t = 0.01$, which is really governed by the interval at which we capture frames more than accuracy or stability constraints.

However, the spring-mass system by itself is relatively uninteresting. We wish to have the elastic body interact with other objects. First, we need our simulations to interact with solid objects. Using the collision detection and handling techniques that were discussed in Sections 5.2.1 and 5.2.2, we get the results shown in Figure 8.2. We also need our elastic object to interact with other moving objects. Specifically, for our particle-based fluid simulator described in Chapter 5, we need to deal with collisions between the elastic body and fluid particles. Again, we compute the collisions as described in Sections 5.2.1. Figure 8.3 shows our elastic body colliding with a single particle.

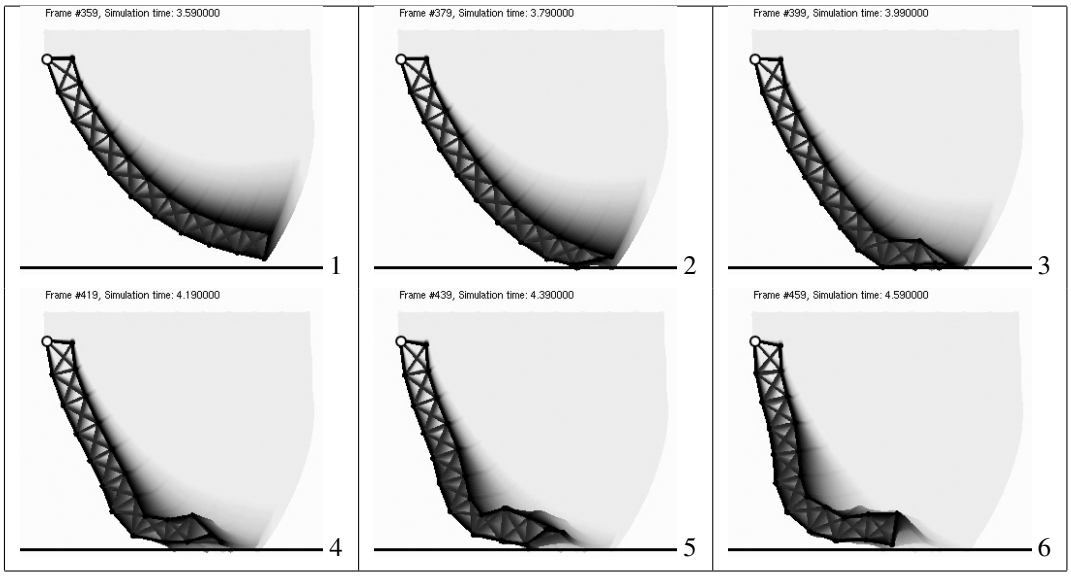

Figure 8.2: Sequential frames from a simulation of a simple spring-mass system colliding with a rigid boundary. The initial conditions and parameters of this simulation are the same as those in Figure 8.1. The only difference is that we add a static boundary 0.45 units below the lower face of the elastic strand, with which the strand collides. We focus on the frames of the animation near the time of the first collision of the two bodies.

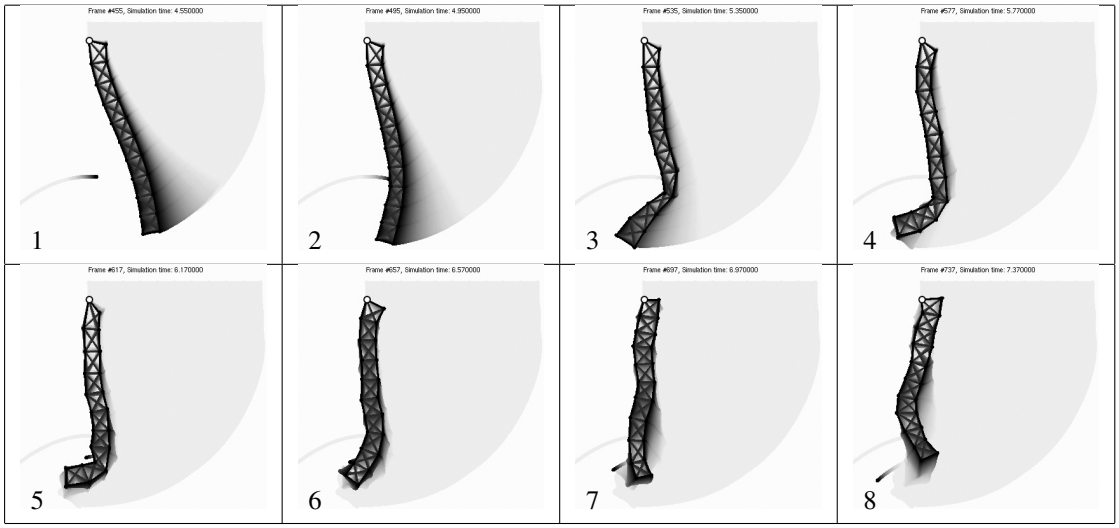

Figure 8.3: Sequential frames from a simulation of a simple spring-mass system colliding with a moving point-mass. The initial conditions and parameters of this simulation are the same as those in Figure 8.1. We focus on the frames of the animation near the time of the first collision of the two bodies. The initial position of the solid particle is $\vec{x}_i = (-0.48, -0.93, 0.0)$, relative to the fixed pivot point of the elastic strand. The particle's initial velocity is $\vec{v}_i = (0.11, 0.22, 0.0)$, and with our simulation its velocity at the collision is $\vec{v}_i = (0.0601, -0.0025025, 0.0)$.

8.1.2 SPH Fluids and Spring-Mass Elastics

In Section 8.1.1, we showed the results of a spring-mass system interacting with moving solid particles. We can then have the particles behave like a fluid volume by applying the SPH-related forces that are described in Chapter 5. We account for the viscosity and pressure forces that each fluid particle will exert on neighbouring particles. Figure 8.4 shows the effect of adding these fluid forces to the particles. Remember from our discussion of fluid pressure forces in Section 5.1 that the compressibility of the fluid is governed by the gas constant κ_p. Large values of κ_p make the fluid less compressible, but also make the system more numerically stiff. This stiffness forces smaller time-steps and makes the system more expensive to simulate. The results in Figure 8.4 were generated using relatively small gas constant ($\kappa_p = 0.0005$). This small value for κ_p makes the fluid extremely easy to compress, though the simulations can be carried out quite efficiently on modest hardware: we can simulate approximately 5 frames of simulation per second. We can get away with a small gas constant with such a simple system for several reasons. First, the elastic body in Figure 8.4 is passive, and does not exert extremely strong forces on the fluid. Second, we are not terribly concerned with the physical accuracy of this system. Much of the force of the elastic strand goes into compressing the fluid in the direction of its surface normal, instead of creating a flow that is tangential to the elastic surface. Eventually, the density of the fluid will be such that its pressure forces will push the elastic strand away, albeit with a delayed reaction. Although this is not physically accurate, it still makes for a dynamic animation that appears to show the interaction of the two bodies.

However, in the case of our jellyfish, the umbrella is quite active, in that it exerts relatively large forces onto the fluid. These strong forces have two large effects on the computational cost of our simulation. First, the large elastic forces will compress the fluid a significant amount before the fluid density builds high enough to have a significant counteracting force. Thus, the gas constant κ_p needs to be quite high in order to give the necessary force magnitude. Another issue that is pointed out by Martin et al. [63] is that the radius of the smoothing kernel must be small enough to capture sharp changes in the density field of the fluid. Consequently, the number of particles has to be quite high, which also increases the numerical stiffness of the system and thus forces smaller time-steps, aside from adding more variables to be simulated.

To experiment with the stability properties of an SPH fluid simulator interacting with an active elastic body, we designed a small test model of a mosquito larva that wriggles its way through a body of water. Figure 8.5 shows several consecutive frames of the resulting animation. Note that the system is quite numerically unstable. The positions of the point-masses tend to flicker wildly between consecutive steps.

If we use error controlling techniques like those described in Section 2.4.6 to dictate our time-steps, then we do get diminishing amounts of jitter artifacts, though they are not completely eliminated. Still, we show the run-time performance of those simulations in Figure 8.6. We tried simulations with gas constants between 4×10^{-2} and 4×10^{1}, and found that the run-time for each numerical method grows linearly with the gas constant. With a gas constant of 4×10^{1}, simulating just 10 seconds of the system with the fastest method, the fifth-order Dormand-Prince method, took 670 seconds of CPU time. To complicate matters

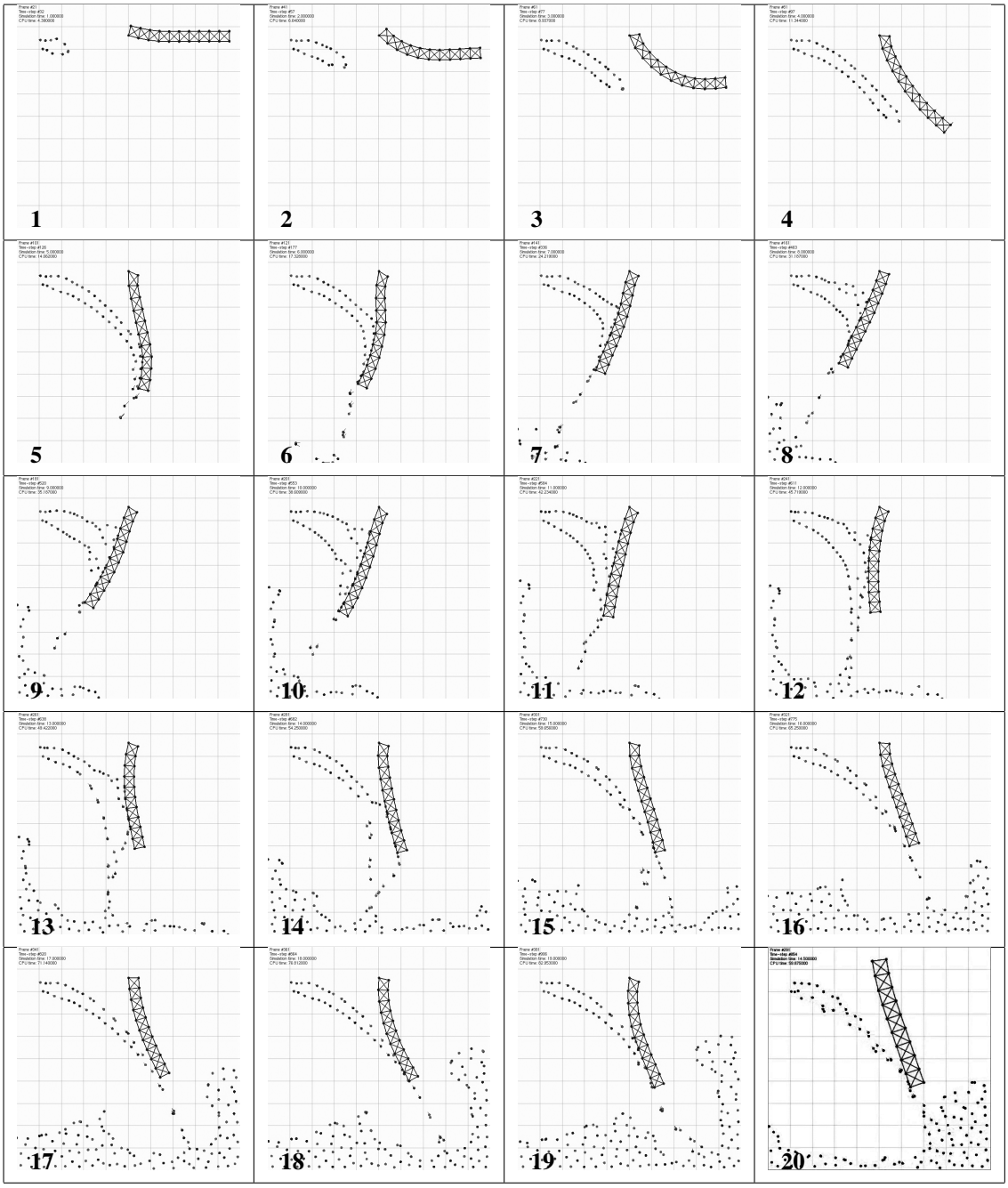

Figure 8.4: Sequential frames from a simulation of a simple spring-mass system colliding with a stream of SPH particles. The spring-mass system has the same initial conditions and physical parameters as the one in Figure 8.1. However, the numerics of this simulation are more complicated, and smaller time-steps are required. Thus, we use a step controller that makes use of the Dormand-Prince method's embedded pair to estimate error. We allow relatively generous error tolerances of $10^{-2} + 10^{-1}\left(\tilde{y}(t+\Delta t)\right)$, where $\tilde{y}(t+\Delta t)$ is the numerical solution that is calculated by the time-step. The scene is enclosed in a square static boundary with a width and height of 1 unit. Fluid particles are released every 0.05 units of time, and from a position of (0.1, -0.2, 0.0) relative to the top-left corner of the static boundary. Each fluid particle is given an initial velocity of $(0.125, 0.0, 0.0)$, and a mass of $m_i = 4$. A smoothing radius of $h = 0.1$ was used.

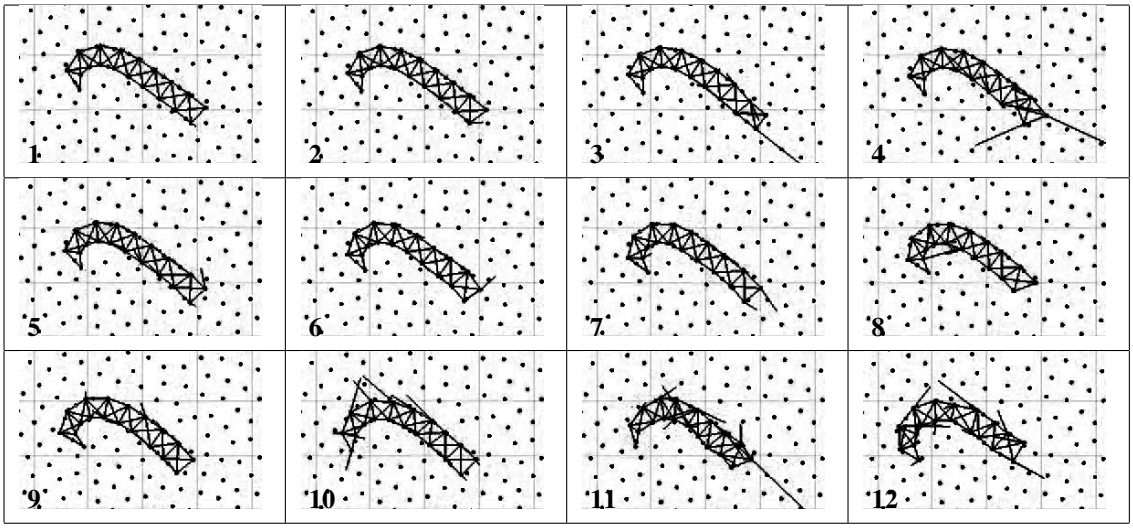

Figure 8.5: Consecutive frames from a contracting spring-mass model of a mosquito larva interacting with SPH particles. The system is quite structurally unstable, and so discontinuities in the positions of the spring-mass mesh points are evident. As well as the springs between each point of the larva model, velocity vectors are also drawn for each point to also show the discontinuity of the velocity values as well. This simulation is contained in a 1×1 static boundary square, and the fluid is comprised of 625 particles, arranged in a uniform triangle pattern. The rest density of the fluid is $\rho_0 = 2000$, and the mass per fluid particle is determined to be $m_i = 2.8$. The viscosity of the fluid is $\nu = 8.9 \times 10^{-3}$ and the gas constant used is $\kappa_p = 40$. For the elastic structure, each solid particle had a mass of $m_i = 1$, and linear springs had an elastic constant of $\kappa_l = 400$. The solid object is 0.4 units long, with a height equal to the smoothing radius (so particles on either side of the object would not interact with each other). A smoothing radius of $h = 0.05$ was used.

further, simulating a highly incompressible fluid like water requires a gas constant that is several orders of magnitude higher. If the run-time performance continues to grow linearly with higher gas constants, then we can expect run-times to also be several orders of magnitude higher. This is far too slow for our purposes. Thus, we abandon particle-based fluid methods for simulating incompressible fluids, and move instead to grid-based methods.

8.1.3 Grid-based Fluids and Spring-Mass Elastics

For the purposes of simulating very active elastic bodies as they interact with incompressible fluids, we find that a grid-based approach is most effective, although the system can easily become numerically unstable if proper care is not taken. As discussed in Chapter 4, we have a couple of different choices in how to treat the interaction between a point-based solid representation and a grid-based fluid representation. We have not attempted to implement the pressure gradient method that was discussed in Section 4.3.1, as it is best suited for rigid bodies [8], and also can allow significant amounts of fluid to leak through the solid boundary [41]. We instead make use of Peskin's immersed boundary method that was discussed in Section 4.3.2. This

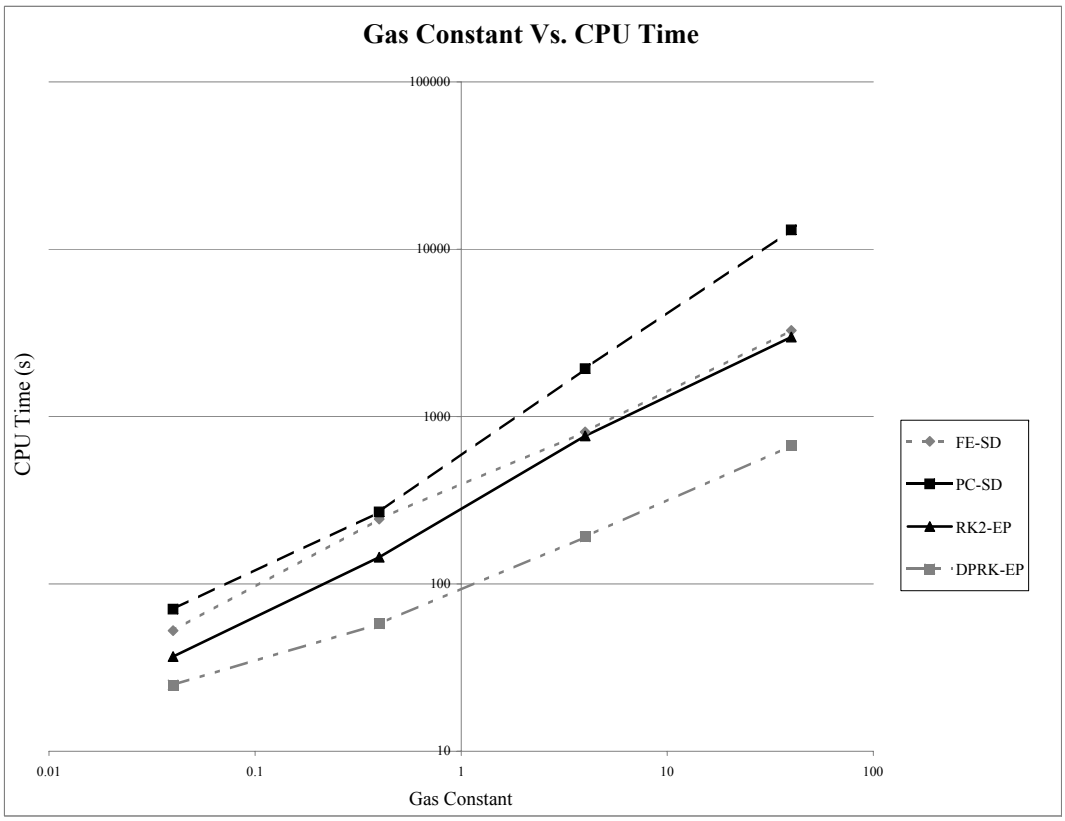

Figure 8.6: The relative performance of different integration methods simulating ten seconds of an SPH system with different gas constants. We test the forward Euler (FE), predictor-corrector(PC), the common second-order Runge-Kutta (RK2), and the Dormand-Prince Runge-Kutta (DPDK) methods. When available, we use embedded pairs (EP) to peform error estimation and step controlling; otherwise, we use step-doubling (SD). Both axes are in logarithmic scale.

method is not foolproof either, and does allow small amounts of fluid to leak through the elastic solid [104]. However, the amount of leaked fluid reported in immersed boundary literature is much smaller than that of pressure gradient methods. The other advantage to the immersed boundary method is that it was specifically designed to model the interaction between elastic bodies and incompressible fluids.

We start our discussion of grid-based fluid methods by contrasting the methods themselves, without the added complication of elastic body interactions. A common benchmark test of a fluid simulator is the so-called *driven cavity test*. In 2D, this test involves a regular square area filled with fluid, which we refer to as the fluid cavity. The concept of the driven cavity is that the top of the box has a conveyor belt that drives the top region of the fluid with a constant velocity. To simulate this driven cavity, the top side of the fluid cavity is an in-flow boundary, and the direction of the inward flow is tangential to the boundary surface. The other three sides of the cavity are solid boundaries, either no-slip or free-slip boundary conditions as was discussed in Section 4.1.3. Figure 8.7 shows the flow pattern of the driven cavity test for both the semi-Lagrangian and classical Eulerian methods, and also for several viscosity values. We consider the Eulerian

approach to be correct because its results are similar to those of previous literature in computational fluid dynamics [40]. The semi-Lagrangian method yields results that are qualitatively different than the Eulerian method. However, the semi-Lagrangian method still produces a divergence-free velocity field, and this method has also been widely used in the graphics community because it still produces convincing animations of fluids [8, 14, 32, 102, 60, 31].

When we add elastic bodies into the mix, Stockie and Wetton [104] give us some ideas of how to test the immersed boundary method. The test is intended to see how much of the fluid leaks from within an enclosed elastic membrane. The initial configuration of the test is a set of points arranged in an oval with linear springs connecting adjacent points. The springs are initially stretched so that they will pull inward. However, the volume of the fluid will prevent the area of the oval from changing. Thus, the elastic membrane will pull itself into its minimum energy configuration, which is the shape that has the minimum perimeter for the given area of the fluid within. The shape of this configuration is a circle. As a loose test, we ensure that our membrane does approach this circular shape. Figure 8.8 shows the evolution of the elastic membrane as it pulls to a circular configuration.

Stockie and Wetton [104] also used their oval membrane test to get a more quantitative sense of the accuracy of the method by measuring how much the fluid area changes as the simulation progresses. Since the fluid within the membrane is under elastic pressure, Stockie and Wetton measured the accuracy of the method by how much of the fluid volume is retained as time progresses. We also make the same comparison, and show our results in Figure 8.9. We get numbers that are similar to Stockie and Wetton [104] for the classical Eulerian fluid solver. With a first-order explicit time-stepping scheme, Stockie and Wetton lost 2.8% of their volume. With our implementation, we only lose 2.575% with the same method. However, if we use the semi-Lagrangian method, we lose 3.67% of the volume. We find that the levels of leakage by using the semi-Lagrangian method to be a small price to pay for its computational efficiency.

For graphics purposes, we are more concerned with plausibility and speed than with accuracy. We also compared the run-time of our implementation of the immersed boundary method with both classical Eulerian and semi-Lagrangian fluid solvers. With both methods, we use a first-order explicit time-stepping scheme like that described in Section 4.1.2. Stockie and Wetton [104] suggest that we could reduce the computational cost of our simulation by an order of magnitude by using a semi-implicit time-stepping regime. However, we find that we drop our cost by two orders of magnitude simply by switching our Eulerian fluid method with the semi-Lagrangian method, though at the cost of accuracy. Figure 8.10 shows the relative amount of CPU time needed to simulate our jellyfish model with the two different fluid solvers.

8.1.4 Stability of the Immersed Boundary Method

The stability of our system greatly affects the speed at which we can simulate the system, and diminishes the accuracy and believability of our animations. We encountered two main types of stability issues in our work. First, instability of the structure of the spring-mass system itself can cause the elastic body to do

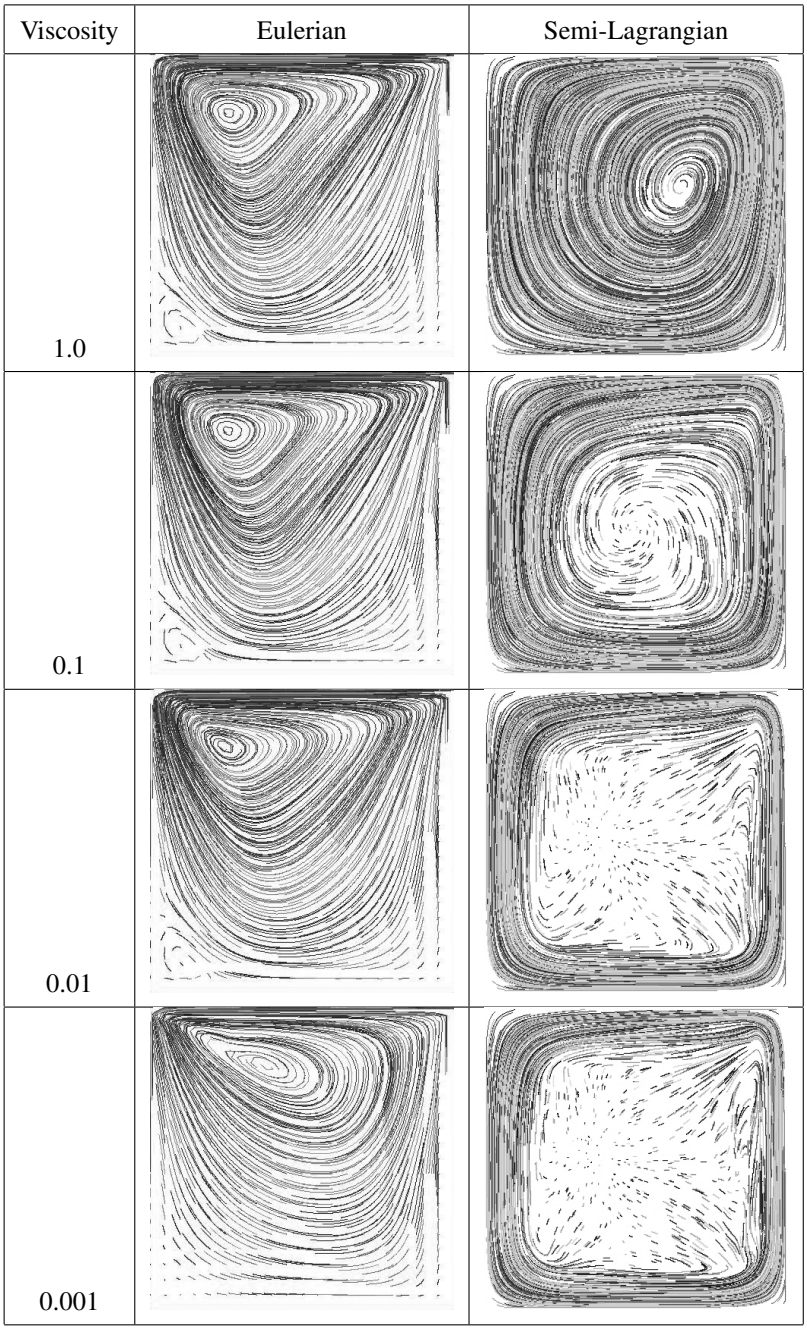

Figure 8.7: Comparison of grid-based fluid methods on the driven cavity problem with several different viscosity values. On the left is the result from the classical Eulerian method. On the right the semi-Lagrangian method. The Eulerian method is deemed to be most accurate, as it generates flows that are similar to those of physical experiments [40]. Note the secondary vortex that appears in the lower left corner of each cavity with the Eulerian method. The semi-Lagrangian method does not capture these types of flows. In all cases, simulation was done on a 1×1 grid divided into 30×30 grid cells, where the top edge was given a horizontal velocity of 1. A constant time-step of $\Delta t = \frac{1}{30}$ was used.

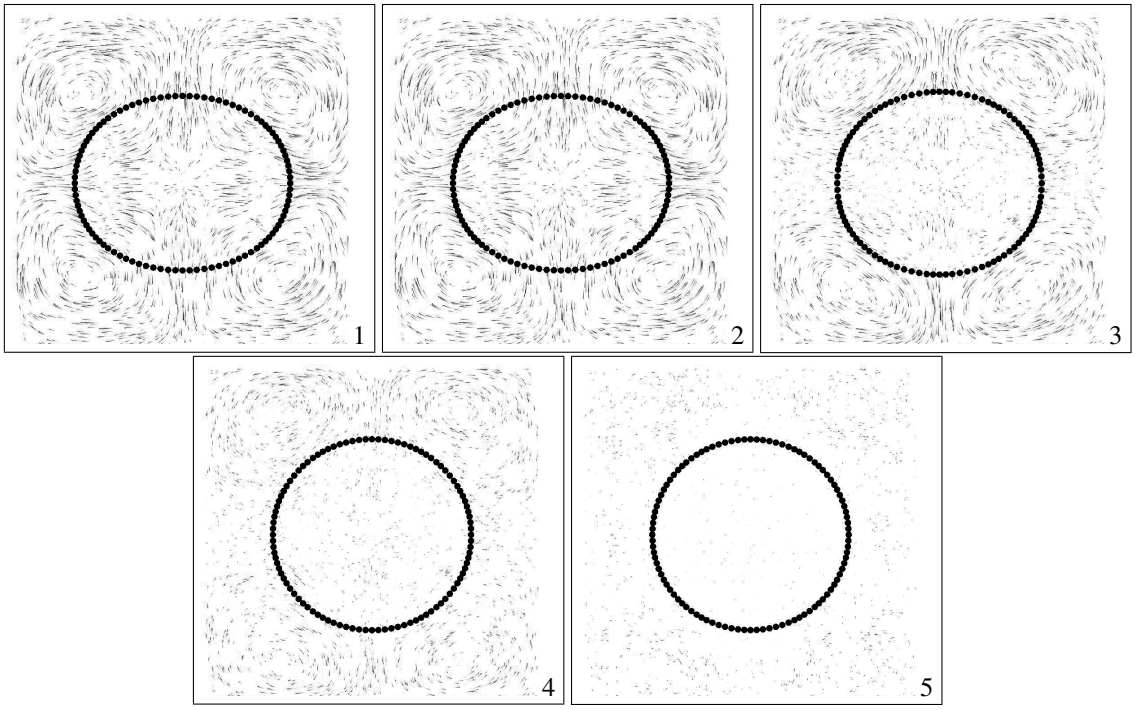

Figure 8.8: Sequential configurations of the oval membrane test that is prescribed by Stockie and Wetton [104].

unrealistic things. For example, if the springs in our system are too weak, then the mesh may collapse into itself or become inverted, or undergo artificially large deformation. We consider our simulation to be structurally unstable when it undergoes an artificially high amount of deformation. This structural instability can occur because the spring-mass mesh is too low a resolution to properly capture the force profile over the elastic body, and thus external forces (i.e., from the surrounding fluids) can overpower the elastic forces. By contrast, we consider the simulation to be structurally stable if the elastic body can maintain its desired shape and structure throughout the simulation. One way to increase the structural stability would be to increase the spatial resolution of the elastic mesh, though at an increased computational cost. Note that decreasing the time-step of the numerical integration does not generally increase the structural stability of the system because the elastic forces are relatively small and thus the errors due to the integration method are also quite small. We can also combat structural instability by increasing the strength of the springs via the spring constants κ_l and κ_θ that were discussed in Sections 2.1, though by doing so, we change the material properties of the elastic body, and thus lose physical accuracy. Figure 8.11 shows the evolution of a jellyfish slice as it becomes structurally unstable. Notice how the spring-mass mesh is being greatly deformed, yet the fluid flow is smooth and continuous.

Another type of instability with which we grapple is numerical in nature. We discussed numerical stability in general in Section 2.4.7. To improve the numerical stability of our simulations, we need to reduce the error incurred by the integrator, either by using a better integration method, or by reducing the

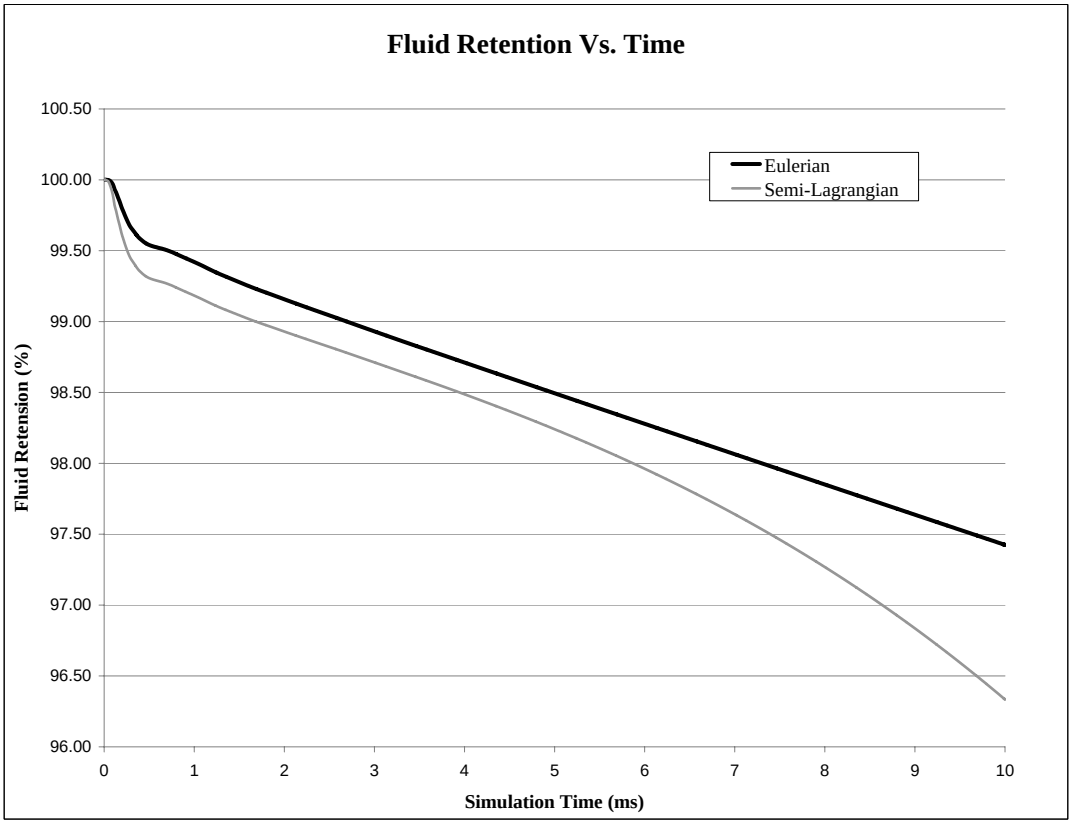

Figure 8.9: Percentage of fluid volume that is retained with the immersed boundary, for both Eulerian and semi-Lagrangian fluid methods, as measured with the oval membrane test by Stockie and Wetton [104]. The volume values are normalized against the starting volume of the membrane. The fluid cavity is again a 1×1 square divided into a 30×30 grid. The fluid is inviscid (i.e., $\nu = 0$). The linear springs have an elastic constant of $\kappa_l = 10$. A constant time-step of 0.002 was used. The elastic oval's initial dimensions are 0.35×0.25.

time-step of our integrator. Note that we could also reduce the spring constants κ_l and κ_θ, and again doing so changes the mechanical properties of the elastic body. Figure 8.12 shows the evolution of a jellyfish slice as the entire simulation becomes numerically unstable. Notice how the spring-mass mesh is able to keep its general shape, but the fluid flow is chaotic and discontinuous.

Since we only have a limited number of numerical methods at our disposal, and we have requirements on the run-time performance of the simulator, yet we do not need a precisely accurate simulation, then the only degrees of freedom that we can manipulate are the elastic constants of the springs and the viscosity of the fluid. However, clearly the elastic constants have to be carefully set to satisfy both the numerical stability and structural stability of the overall system. In practice, we had to make sacrifices on our performance requirements to find a suitable configuration of these two stability requirements. For the fluid grid itself, we have the stability requirements that were discussed in Section 4.1.2. However, we also need constraints that ensure the stability of the spring-mass mesh. Stockie and Wetton [104] discuss stability regions for

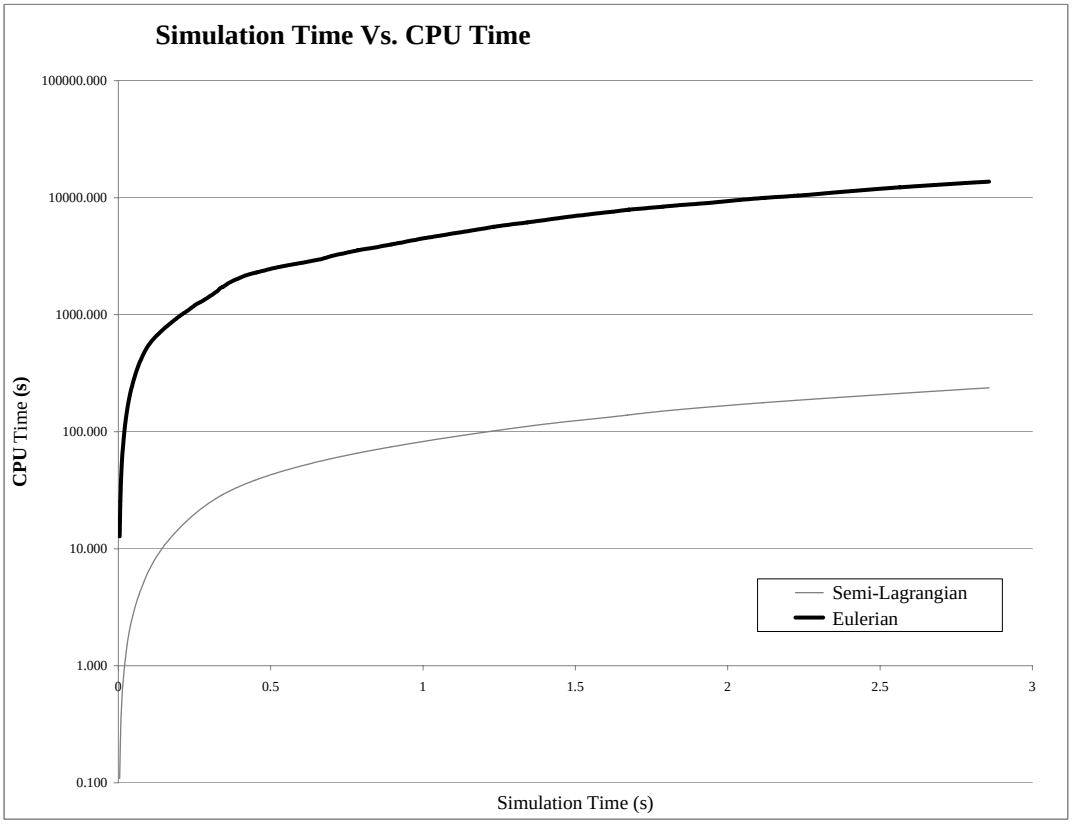

Figure 8.10: Comparison of the CPU time needed to simulate our jellyfish using the semi-Lagrangian and classical Eulerian methods.

simulating the Stokes equation for fluid flow, but which does not account for the advection term of the Navier-Stokes equations. Also, their work does not give us a concise expression from which we can build a stability condition for the elastic body. Such a condition would likely limit the amount of work that each spring could do in each time step. We were unable to derive such a condition. Instead, we experimented with the space of parameters for a configuration that gives us an efficient simulation, while still allowing us to generate motion that is similar to observed jellyfish.

8.2 Jellyfish Results

With regard to the jellyfish itself, we would like to have some reassurance that our simulations are at least qualitatively similar to those of real jellyfish. In the following sections, we discuss how our model compares to empirical data and analytical models from the biology community. We look at the morphological aspects of the jellyfish as it contracts, as well as the thrust that it achieves due to contraction. We also discuss how the contraction frequency that we choose affects the thrust achieved our simulated organism.

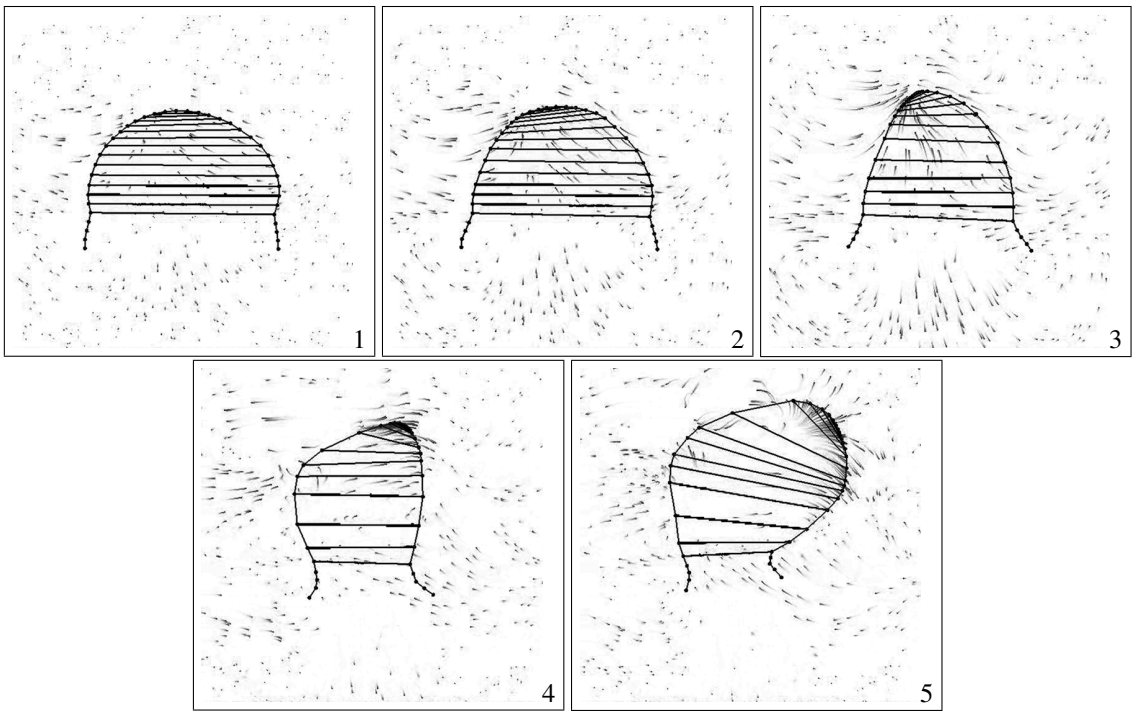

Figure 8.11: Several frames of a jellyfish simulation in which the elastic coefficients are so low that the mesh becomes structurally unstable. Marker particles have been traced in the fluid's velocity field to show the flow of the fluid over time. For this simulation, the fluid cavity is 80 *cm* in both width and height, divided into 60×60 cells. The fluid's viscosity is $\nu = 0.1304$. The jellyfish's umbrella has a radius of 4 *cm* and a height of 2.8*cm*. The tentacles of the jellyfish are 1.4 *cm* long. The elastic modulus the jellyfish in this simulation is $\lambda = 1.186 \times 10^4$ *Pa*. The jellyfish contracts at a frequency of 0.7 *Hz*. The semi-Lagrangian method was used to simulate the fluid grid, along with a step controller that is based on the stability constraints described in Section 4.1.2.

8.2.1 Contraction Frequency

The frequency of the jellyfish's contractions has a direct effect on the thrust that is achieved by the organism, as well as the morphology of the organism. In Section 6.3, we discussed the contraction frequency of the organism. Megill [65] showed us that the resonant frequency of the organism depends on the diameter of its umbrella, as was illustrated in Figure 6.13. At this resonant frequency, the organism will achieve the optimal thrust for the amount of muscular force exerted by the organism. As was discussed in Section 6.4.2, Megill derives an approximate analytical model of the jellyfish. However, his findings differ from the empirical data captured by Dabiri and Gharib [22]. Without further confirmation of either result, we decided to experiment with different contraction frequencies to determine which ones were optimal for our model. We also suspect that our model has different physical properties, and would not have the same resonant frequency anyway. We have not investigated the thrust for various umbrella diameters, but we have tried different contraction frequencies for the same diameter to see if we get results that are similar to those of Megill's work. Figure 8.13 shows the translational motion of a simulated jellyfish with an umbrella diameter of 40 mm, but

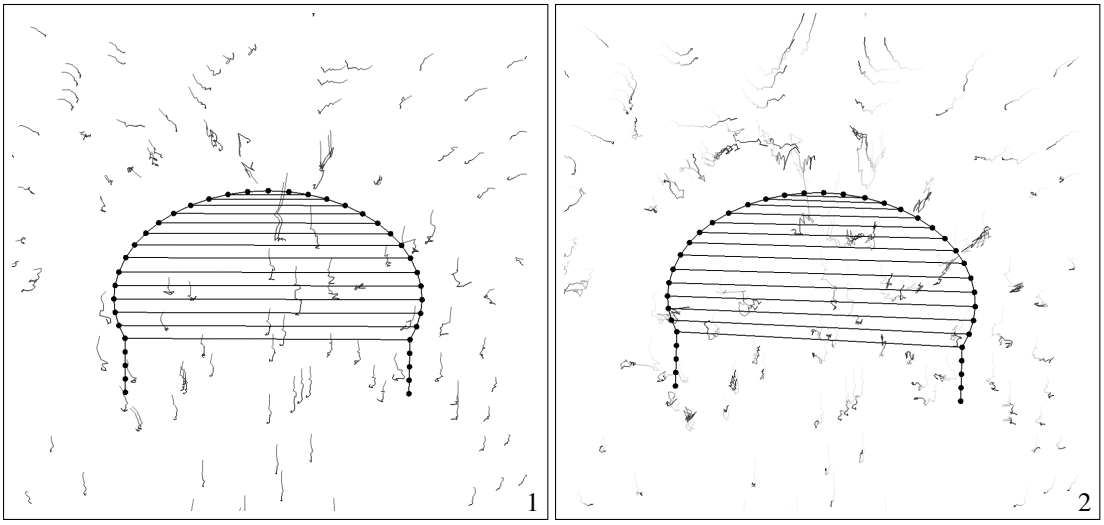

Figure 8.12: Two frames of a jellyfish simulation in which the elastic coefficients are so high that the entire system becomes numerically unstable. Marker particles have been traced in the fluid's velocity field to show the flow of the fluid over time. This simulation was carried out in the same manner as Figure 8.11, except that the elastic modulus of the jellyfish's umbrella is $\lambda = 1.186 \times 10^6$ *Pa*.

with different contraction frequencies.

We ran several more simulations to explore the relationship of contraction frequency and thrust in our system. Figure 8.14 shows the distances traveled by a small jellyfish for several different contraction frequencies. We also get a large maxima for a frequency of 0.7 Hz, which coincidentally agrees with Megill's predicted resonant frequency for an umbrella diameter of 4 *cm*.

We do not have exact data on the thrust that a jellyfish achieves, nor do we have empirical data on the magnitude of the acceleration should be. Dabiri and Gharib [22] do provide us some qualitative ideas as to what shape the acceleration profile of the organism should take, though their data is normalized to be unitless. We showed their findings in Figure 6.10. We also compared the distance traveled by our simulated organisms to the empirical data of Dabiri and Gharib [22]. Figures 8.15, 8.16, and 8.17 show the position, velocity, and acceleration of our simulated jellyfish. Our results do not correspond exactly to those shown in Figure 6.12. Specifically, the translational distances measured by Dabiri and Gharib appear to have an upward curve with a small amount of oscillation. This pattern indicates that the jellyfish was indeed accelerating and decelerating with contraction cycle, but that its general movement was largely governed by a large positively accelerating thrust. By contrast, our results do not achieve the same amount of thrust, and so the movement profile is dominated more by the higher-frequency motions of each contraction. We also note that although the large-scale movement of our simulated organism is at the same frequency as the muscle contractions, another harmonic frequency appears in our results, indicating that our system has another resonant frequency. This frequency appears to be at the next highest octave. We find the resulting animations to be satisfactorily convincing. However, we realize that the visual appeal of our simulations are

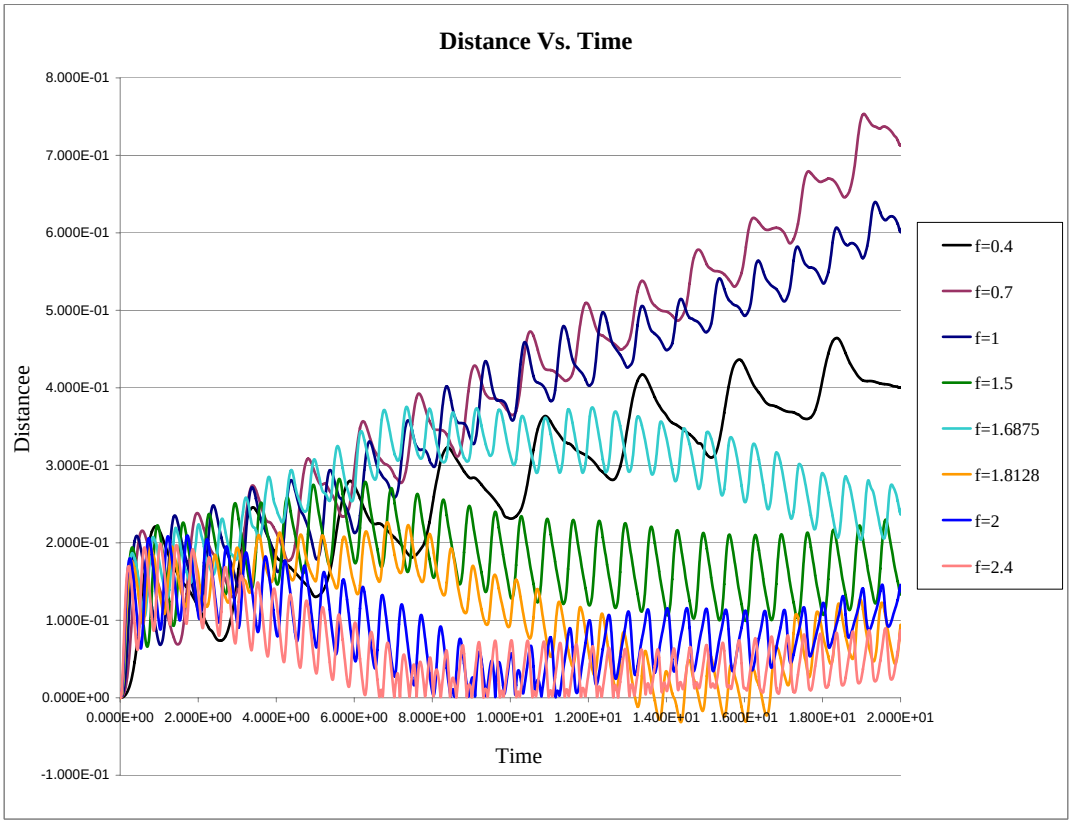

Figure 8.13: Trajectories of several simulated organisms with different frequencies of contraction. All results were simulated with a 4 *cm* organism. This simulation was carried out in the same manner as Figure 8.11, except that the elastic modulus of the jellyfish's umbrella is $\lambda = 1.186 \times 10^5$ *Pa*, and the contraction frequencies were those plotted in this figure.

subjective and depend on the observer. For instance, our animations might be suitable for members of the general population, but we do not expect to convince a well-trained marine biologist.

Aside from pure locomotion, we are also trying to simulate the morphology of the organism. Thus, we would like to reproduce the geometric changes that the umbrella undergoes. Again, Dabiri and Gharib [22] give us empirical data on the matter, which was shown in Figure 6.10. We provide similar data on our jellyfish in Figure 8.18. Again, our results are not an exact match of empirical data, but resemble captured data of real jellyfish.

8.2.2 Extrapolating from 2D Slice Data

The previous sections of this chapter discussed the means of simulating a 2D slice of a jellyfish umbrella. In Section 7.2, we discussed a means of interpolating the geometry of our 2D physics model so that we could render the model at an arbitrary resolution. Section 7.3 then discussed our method of extrapolating this high-resolution 2D rendering model to a 3D surface model. The extrapolation process also introduced

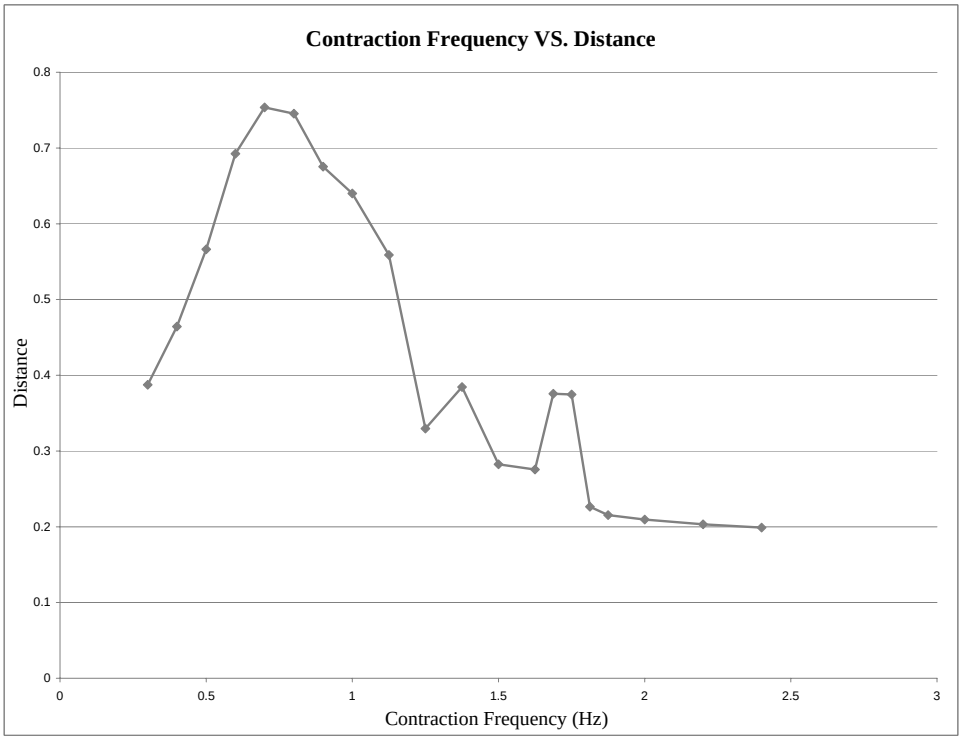

Figure 8.14: Distance traveled with different frequencies of contraction. All results were simulated with a 4 *cm* organism. This simulation was carried out in the same manner as Figure 8.13, except that the contraction frequencies were those plotted in this figure.

some variation into the surface model by perturbing the surface with various types of noise. In particular, three types of noise were considered. First, structural noise accounted for the variation in the surface that was due to features that naturally occur in particular species of jellyfish. Most notably, some species have parabolic extrusions that run laterally down the umbrella in a uniform way. We also considered features that occur due to the nonlinear compression property of the umbrella's mesoglea, as discussed in Section 6.2. Essentially, thick bands of flesh within the umbrella compress more easily than its surrounding flesh, and cause ridges to form when the umbrella is contracted. Lastly, the umbrella may simply have a certain amount of unstructured variation to its geometry. Figure 8.19 illustrates the stages in which we generate a 3D surface model from our 2D physical simulation.

8.2.3 Removal of Fluid Simulation

We were curious as to whether we could speed up our simulation for applications that do not require the interaction between the jellyfish and its surroundings. By removing the fluid solver altogether, we could dramatically increase the running time of our system. In fact, removing the fluid grid and simulating the elastic body by itself does reduce the computational cost by approximately three orders of magnitude. However, the morphology of the organism is less than convincing. The model that we have described thus far

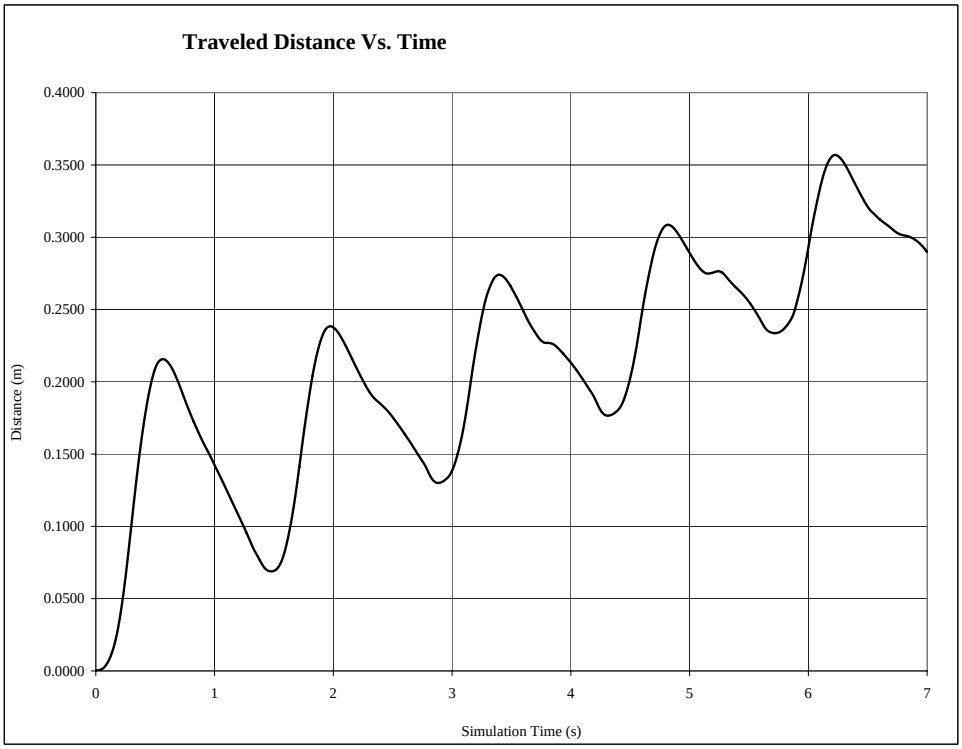

Figure 8.15: The distance traveled by a 4 *cm* simulated jellyfish with a muscle contraction frequency of 0.7 Hz.

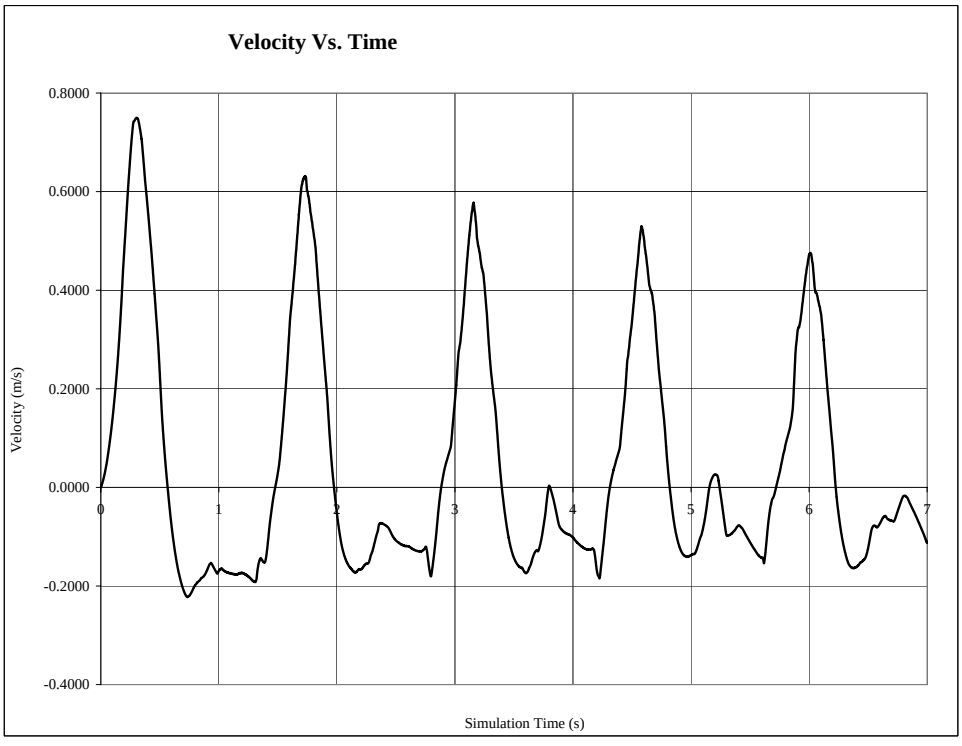

Figure 8.16: The velocity of a 4 *cm* simulated jellyfish with a muscle contraction frequency of 0.7 Hz.

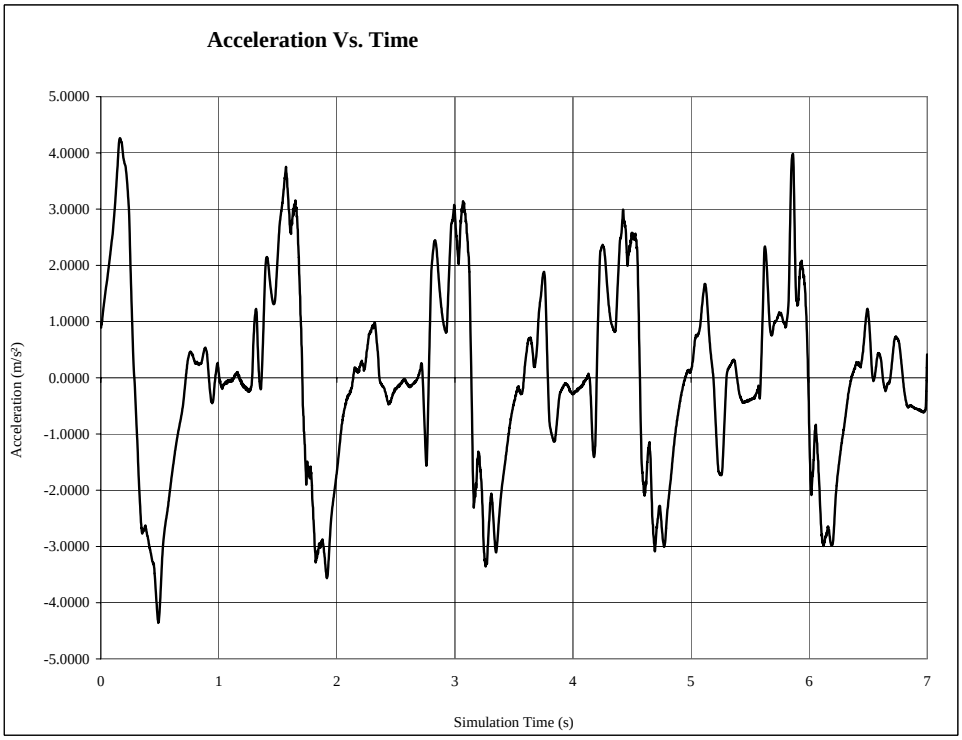

Figure 8.17: The acceleration of a 4 *cm* simulated jellyfish with a muscle contraction frequency of 0.7 Hz.

generates strong elastic forces to overcome the resistance of the surrounding fluid. With the fluid removed, the model overshoots its positions during the swimming motion. We could induce some artificial fluid resistance by adding drag forces that were described in Section 2.3.1.

Also, without the full fluid simulation, the umbrella does not bulge out during contraction as it does when the fluid is present. We also attempted to account for this by adding an approximate notion of fluid pressure to the system. We defined a scalar pressure magnitude that is proportional to the change in the umbrella's volume:

$$p = \alpha \frac{\partial V}{\partial t} \tag{8.1}$$

where α is some artistically chosen scaling factor. However, even after trying a wide range of values for α, we still find it difficult to get realistic motion from our organism. At this point, we abandoned the quest for a fluidless simulation model for jellyfish. The primary advantage to simulation would be the interaction of the model with other objects. However, the lack of a fluid solver prevents detailed interaction. Thus, we may as well use some other animation technique that is more efficient, such as ad-hoc procedural techniques or key-framing.

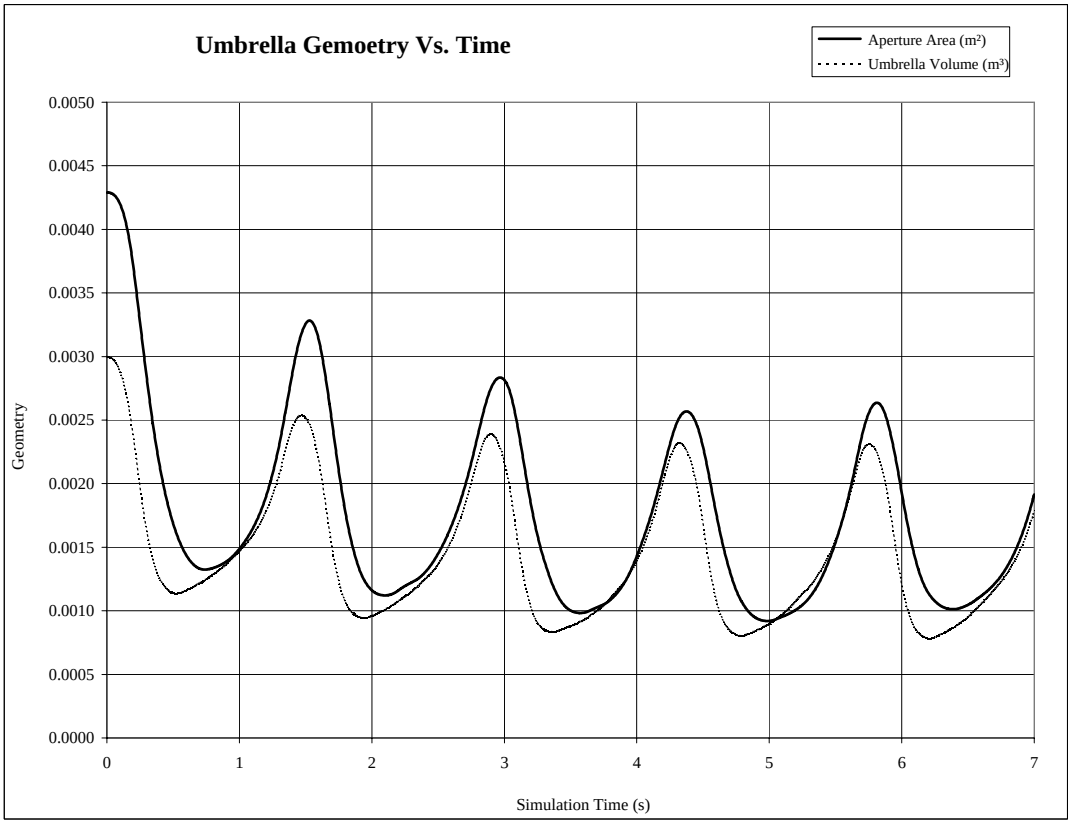

Figure 8.18: The morphology of a 4 *cm* simulated jellyfish with a contraction frequency of 0.7 Hz. The simulation is carried out in the same manner as Figure 8.14, except that we use a fixed contraction frequency.

8.2.4 Final Renderings

With all of the pieces in place for our animation, we are ready to render images of our model. Figure 8.20 shows a sequence of images from one of our animations. The parameters of our system, and the values that we used in Figure 8.20, are given in Figure 8.21. We light the scene with simple direction light sources [116]. With these types of lights, we assume that the light source is sufficiently far away that the light rays are essentially parallel to each other, and also that any attenuation effects due to distance is negligible.

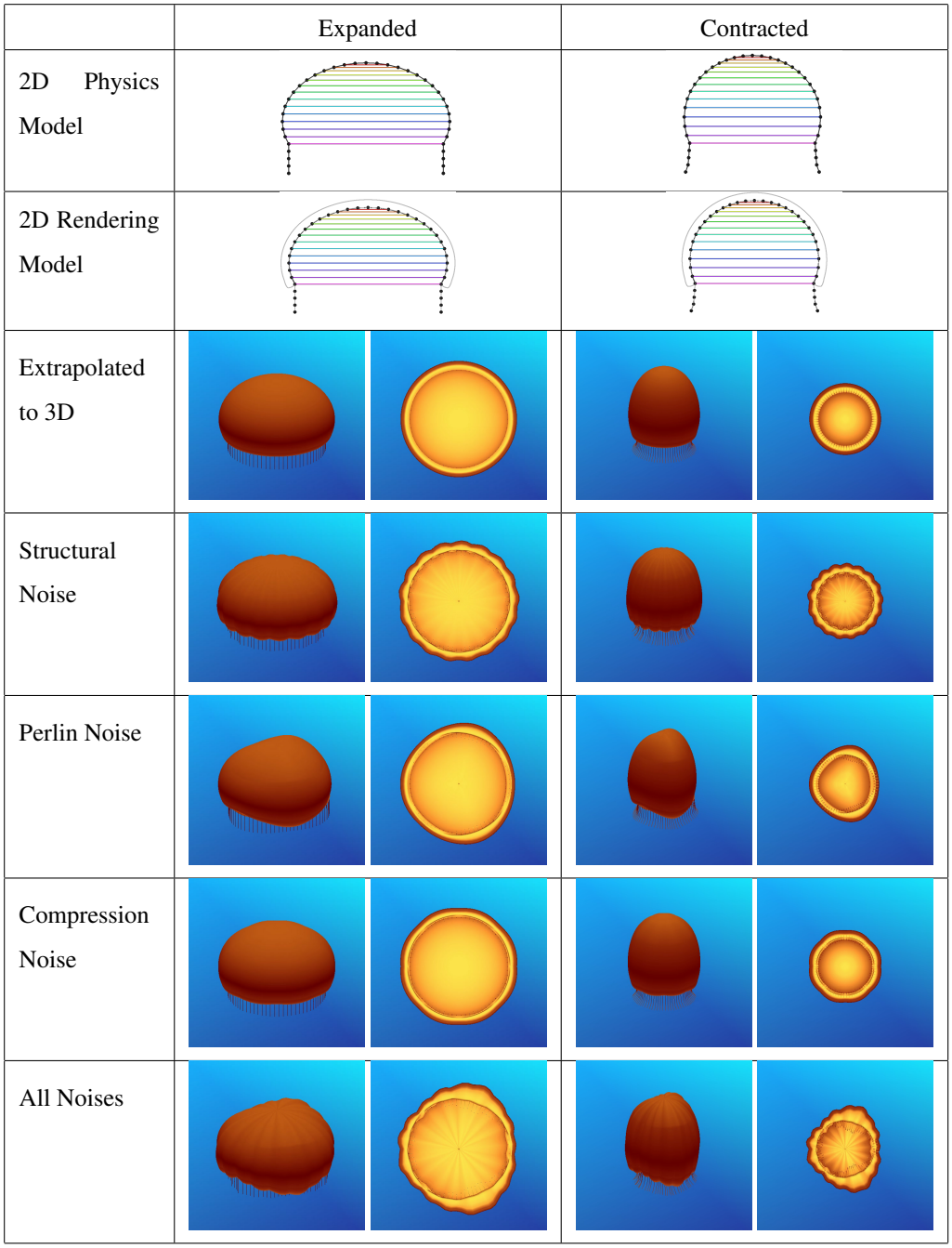

Figure 8.19: Results of each stage of our animation system. As explained in Chapter 7, we first simulate a 2D slice of the organism with a coarse physics model, and then interpolate within that coarse mesh to get a smoothed rendering model. We then extrapolate the smoothed result to 3D and add various types of noise to the geometry. With regard to the noise that we apply to the 3D umbrella, we define the total noise to be a linear combination of the other noise values $c_{ij}^{sum} = 0.01875\ c_{ij}^{structural} + 0.125\ c_{ij}^{compressed} + 0.45\ c_{ij}^{unstructured}$. We define $c_{ij}^{structural}$ as in Equation 7.11, as opposed to Equation 7.10, and use a frequency of $f = 9$, as discussed in Section 7.3. The thickness of the umbrella is $\vartheta = 0.02\ cm$

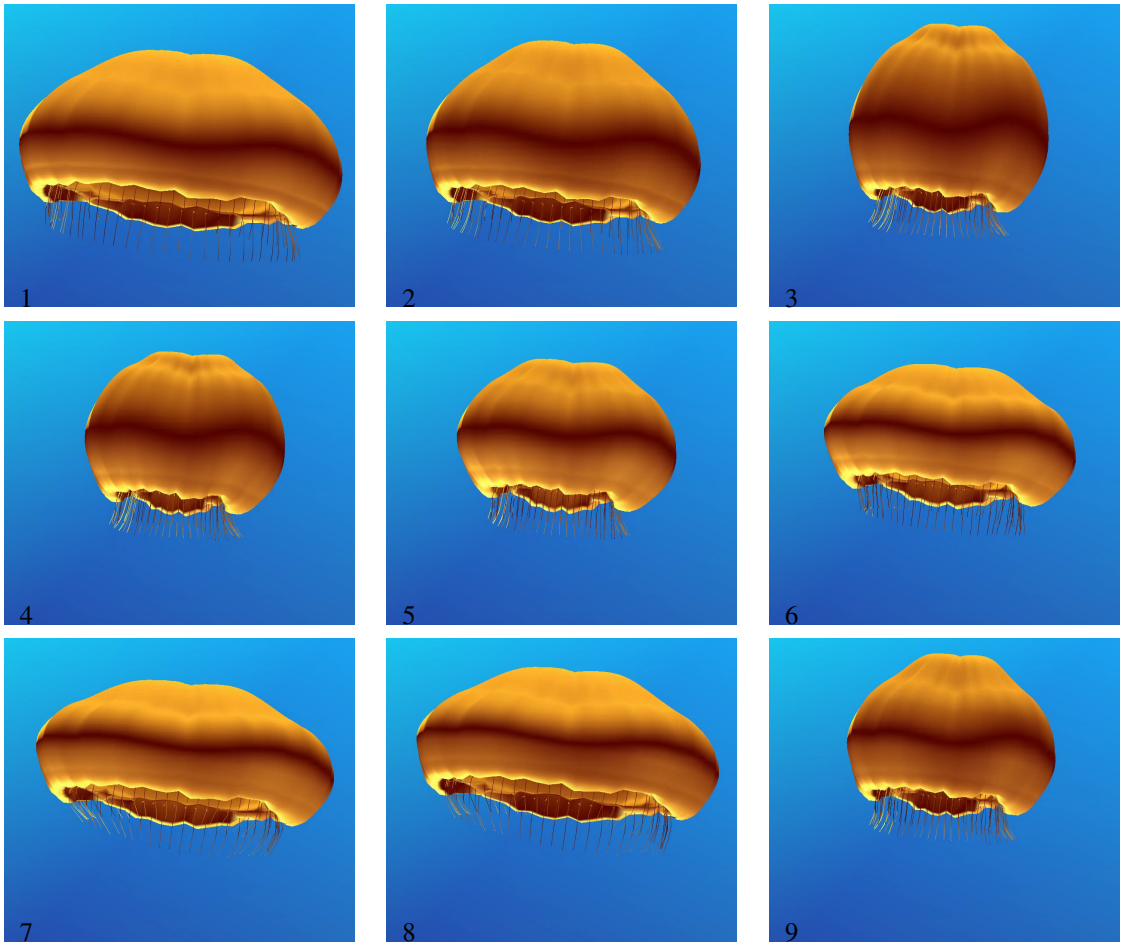

Figure 8.20: A final sequence of images from a fully rendered jellyfish model. The 2D simulation was carried out in the same manner as Figure 8.14. The parameters that govern the 3D model's variation are the same as those in Figure 8.19.

Fluid Grid Dimensions:	$80\ cm\ \times\ 80\ cm$
Fluid Grid Resolution:	$60\ cells\ \times\ 60\ cells$
Fluid Viscosity:	$\nu = 0.1304$
Umbrella Diameter:	$4\ cm$
Umbrella Height:	$2.8\ cm$
Umbrella Thickness (ϑ):	$0.02\ cm$
Umbrella Contraction Frequency:	$0.7\ Hz$
Tentacle Length:	$1.4\ cm$
Jellyfish Elastic Modulus (λ):	$11,860\ Pa$
Structural Noise Frequency:	9
Structural Noise Factor:	0.01875
Compression Noise Factor:	0.125
Unstructured Noise Factor:	0.45

Figure 8.21: The parameters of our model, and the values used to generate the results shown in Figure 8.20.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

Our model of jellyfish still leaves many possible avenues of future work. Still, we develop an efficient means of animating jellyfish that also allows for interaction with other objects in the scene. Our methods are not accurate enough for scientific exploration of the creature, and certainly it is possible to point out discrepancies between the results of our work and of real jellyfish. However, our model generates morphology and thrust profiles that resemble those of real jellyfish, and we believe that our model would be suitable for viewing by the general public. The following sections give an overview of our methods, as well as discuss the results of our work and suggest possible directions that future work could take.

9.1 Method Overview

We simulate an approximate model of jellyfish numerically, which accounts for the elastic forces of the organism as it contracts its muscles, as well as the reaction of the sea water that surrounds the organism. We choose simulation as our animation technique because it allows us to model the direct interaction of the jellyfish and its environment. We conduct our simulations in a 2D vertical slice of the jellyfish and exploit the axial symmetry of the organism. However, doing so limits the species that we can simulate, as not all of them are axially symmetric. We also concentrate on the resonant gait of an organism, in which its contractions are regular and at a rate that is approximately the natural resonant frequency of the submerged organism. Additionally, we restrict our interest to fully grown adult jellyfish, as the organism most actively swims in this phase if its development.

To simulate the 2D slice of the organism, we use a particle-based approach to represent the elastic body of the jellyfish. Specifically, we build a spring-mass system that consists of a combination of linear springs to enforce distances between particles, and angular springs to enforce relative orientations of particles. This combination of spring types is used to represent the umbrella hull and the tentacles of the organism. Both of these structural elements are considered passive, in that they do not actively manipulate the structure of the jellyfish. We model the muscles that line the under part of the umbrella as a series of linear springs across the subumbrellar cavity. To introduce movement into the system, we shorten the rest lengths of these subumbrellar springs to simulate the contraction of muscles within the umbrella.

To simulate the interaction between our elastic jellyfish model and the fluid that surrounds the organism,

we use a grid-based fluid simulator in conjunction with the immersed boundary method. This method is specifically designed to couple a grid-based fluid simulator with a point-based elastic body representation. For the fluid model itself, we have a choice between a couple of different simulation methods. We find Stam's semi-Lagrangian method [102, 103] to be an excellent compromise between computational efficiency and physical accuracy. Little is to be gained by using the more expensive classical Eulerian fluid simulation technique. Combining a grid-based fluid model with a point-based elastic model is by no means a trivial task, and much work goes into reconciling the differences between the two spatial representations. We also tried the Smoothed Particle Hydrodynamics (SPH) approach to simulating fluids, which is a particle-based method. However, we found SPH to be prohibitively expensive when simulating incompressible fluids such as water.

With the immersed boundary method, we do not have an exact stability criteria for the system. We do use standard stability criteria for grid-based fluid methods to control the stability of the fluid part of the simulation. However, the stability of the elastic body is not explicitly controlled. Thus, we have to fine-tune the parameters of the system in order to achieve stable yet efficient simulation. We can always decrease the size of our time-steps to obtain a stable simulation, but at an increased computational cost. We can also decrease the elastic strength of our spring-mass system, or alternatively increase the viscosity of the fluid. However, changes to the physical parameters of the system will have negative effects on the accuracy of our simulation. We must carefully balance realism with efficiency in our model.

The 2D model that we simulate is quite coarse, and so we generate a higher resolution model by threading a cubic spline interpolant around the points of the simulation model. With this higher resolution slice model, we extrapolate to a 3D surface by defining discs that go through either side of the jellyfish's umbrella. These discs will be artificially smooth, and so we add several forms of variation to the discs. First, we add parabolic ridges that run vertically down the surface of the umbrella. These ridges are specific to certain species of of jellyfish. For example, the Halopsis ocellata in Figure 6.3 has 16 of these ridges across its umbrella, whereas the Pelagia noctiluca has none. More general to all species, sinusoidal ridges will also appear vertically down the umbrella as the organism contracts. These patterns are the result of the non-linear compression properties of the jellyfish flesh. Thus, we set the amplitude of the sinusoids to be proportional to the amount of contraction that the umbrella undergoes. Lastly, we also add continuous noise to the umbrella's surface to give each individual jellyfish a unique shape. This kind of random variation is observed in real jellyfish, though we are unaware of any concrete model of this variation.

9.2 Future Work

Much work could still be done with respect to jellyfish animation. To model the organism more accurately, we would need more information about the physiology of the organism. The biology community has different interests regarding jellyfish motion than the computer graphics community, and so they have not focused

on the muscle control of the organism. However, much work can also be done by graphics researchers. We could do more to develop our own control mechanisms for the organism, investigate more complicated rendering issues, and behavioral models. We could also use more research in computation fluid dynamics so that we can better control the speed of our simulations, and perhaps develop faster integration methods for the system.

9.2.1 Modeling Possibilities

In our model, we consider only jellyfish that are axially symmetric, so that we can simulate a 2D slice of the organism and then rotate that model about an axis of symmetry that approximately goes through from the tip of the umbrella to the center of the aperture. Not all species of jellyfish have this sort of symmetry. We could of course model asymmetric species as full 3D models. However, simulating the system in 3D would be rather expensive computationally. Instead, we might be able to simulate multiple 2D slices of the organism, and then interpolate axially between the slices. In such a model, we would have to ensure that the models coincide at the intersection of the two simulation planes. To date, we know of no work that approximates a 3D immersed boundary simulation with intersecting 2D models. We anticipate that such a model would have a number of challenges to be overcome. Principal among these challenges is the need to reconcile any differences in the model that occur at the intersection of the 2D slices.

Even within the set of species that are axially symmetric, our modeling efforts have been concentrated on a specific configuration of jellyfish, roughly matching that of the species Halopsis ocellata that was seen in Figure 6.3. One could imagine a more general model of jellyfish in which all sorts of parameters of the organism's geometry could be taken into account. For example, biologists generally classify jellyfish in terms of their umbrella's height or diameter. Figure 6.13 shows that the raw size of the umbrella has an effect on its resonant frequency, as would be expected. At the moment, our work controls the contraction frequency manually, but one could imagine a model that determines the contraction frequency based on the shape of the umbrella.

Our model is also tailored toward simulating the resonant gait of a jellyfish. Megill [65] discusses other gaits that jellyfish exhibit, and we have not attempted to address these other gaits. Future work could investigate these gaits in more detail, and find a more general model of jellyfish motion that incorporates all of these modes of locomotion.

We still know little about how jellyfish control their physiology to achieve jet propulsion. Although much work in this regard could be carried out by the biology community, we could also make use of work in the field of computer animation. Specifically, van de Panne and Fiume [67] designed a means of experimenting with the control processes for simple creatures using sensor-actuator networks. As was discussed in Section 1.2.6, they were able to find a large number of locomotive modes for even simple mechanical models. We could perhaps leverage this kind of control exploration technique to determine a better means of controlling our jellyfish.

9.2.2 Biology Questions

When we extrapolate a 3D model from our 2D simulations, we find it necessary to add noise to the 3D surface to obtain more plausible variation. We identify three types of noise, namely structural noise due to features that are specific to a species of jellyfish, compression noise due to the nonlinear elasticity of the umbrella flesh, and an unstructured sort of noise that seems to occur because of differences in individuals within the same species. We have no model for the unstructured noise and so we attempt to mimic this noise with continuous noise functions. However, a better model of this unstructured noise could be developed so that we could generate more plausible results.

At present, we render the final geometry of the organism as an opaque surface, which assumes that all light is reflected by the surface, and no light travels through the flesh of the organism. In reality, jellyfish are generally transparent or translucent, and so light will indeed travel through the organism. The graphics community has several different approaches to rendering translucent volumes [55, 13, 62]. In fact, some early attempts to model the optics of jellyfish have already been made [30, 27]. However, we are not aware of any biology literature that discusses the optical properties of a jellyfish.

Some species are also bioluminescent [17], in that they are themselves sources of light. Aside from reflecting and scattering light from their environment, these bioluminescent species themselves give off light to their environment. Hammen [46] gives some information on bioluminescence in general, and preliminary work on jellyfish bioluminescence has been done by Dixon and Shen [27]. However, the problem is still largely open in both the biology and graphics community.

Also, some species of jellyfish exhibit a vein-like structure across their hull. For example, the organism on the right side of Figure 7.7 exhibits one such venous structure. Again, we are not aware of any biology research into the types of branching patterns that occur in these venous structures. The graphics community does have its share of methods by which we can generate such patterns in ad-hoc ways [89, 76, 75].

We currently model only a translational thrust of jellyfish. We do not model other locomotive behavior, such as turning. Megill [65] tells us that the radial muscle fibres that are in the umbrella mesoglea are used change the umbrella's symmetry, and this affects the orientation of the umbrella during swimming. However, the exact way in which the muscles act and the results of their action are still unclear. We made an attempt to model turning under the assumption that the radial muscles would cause the umbrella wall to become thinner yet more spread out. We modeled this spreading of the umbrella by increasing the rest length of the linear springs that run along the umbrella wall. We hoped that doing so would significantly change the symmetry of the jellyfish's aerodynamics. However, our attempts had no real affect on the orientation of the organism, even with length changes that radically deformed the umbrella model. We suspect that other behavior is involved in the turning process, and perhaps this movement may need a full 3D model to be captured properly.

9.2.3 Potential Fluid Dynamics Work

Foremost in relation to our work, we presently have no means of determining how large a time-step to take in our simulations. Currently, no stability conditions have been derived for the immersed boundary, although some analysis of its stability properties has been done [104, 53]. Instead, we used the standard stability conditions for the classical Eulerian method described in Section 4.3.2 as a guideline, but we still have to manually tune the step sizes.

We also discussed the use of smoothed particle hydrodynamics in Chapter 5. For our purposes, we found that SPH was simply too slow, and quickly abandoned the method for our particular application. However, the SPH method is widely used in graphics, as well as astrophysics and fluid mechanics. We did find possible improvements to techniques that model elastic spring-mass systems with SPH fluid models. Muller et al. [80] handled solid-fluid interaction by placing fake fluid particles along the boundary of the solid object. However, if the solid object deformed too greatly, then the fake particles will be too sparse to yield accurate pressure forces. Muller et al. [80] found it necessary to redistribute the fake particles along the solid boundary at every step. Since their work does not prevent collisions between the solid and the fluid particles, we instead choose to let collisions be the sole means of interaction. However, when the solid object is thin, fluid particles may never collide with the solid. Pressure forces from the fluid on one side of the solid will repel the fluid particles on the other side. Similar effects occur with other forces like viscosity and surface tension. We could possibly allow for better solid-fluid interaction by ignoring forces from particles that are on opposite sides of the same solid object. However, we may introduce discontinuities at the edges of solids with this line-of-sight sort of force filtering, as two particles come into view of each other.

Failing a line-of-sight approach to limiting particle interaction, we could compute pressure forces that the solid object exerts on the fluid particles as Muller et al. [80] did in their work. However, rather than approximating these forces with a discrete number of fake particles within the solid, we could instead derive a more analytical approach to solving for the pressure forces. If we assume that the solid has a uniform density, then we could integrate the region of the smoothing kernel that intersects the solid object and use that definite integral to compute pressure forces. Since we generally know what kernels we are using ahead of time, we could derive analytical definite integrals for the kernel over such intersection regions. However, if we want a more general approach in which we can simply plug in any kernel, we could precompute look-up tables of these integrals for each kernel.

Other work may involve the use of high-order implicit integration methods with SPH. Specifically, Desbrun and Gascuel [26] mention a spatially adaptive step size, essentially allowing each particle to be integrated with its own step size. However, for particles that require smaller time-steps, we still have to extrapolate the positions and velocities of its neighbouring particles in order to compute the forces at each mini-step. Desbrun and Gascuel [26] used a leap-frog integration method. Higher-order methods could perhaps be used, especially multi-step methods like the explicit Adams-Bashforth or implicit Adams-Moulton families of methods [38]. These methods essentially fit an interpolant through values at previous time-steps.

Thus, they can easily be used to find future particle positions and velocities.

REFERENCES

[1] Finding Nemo motion picture. DVD, 2003. Pixar Animation Studios, Walt Disney Pictures.

[2] Mary N. Arai. *A Functional Biology of Scyphozoa*. Chapman and Hall, London, 1997.

[3] Randall D. Beer, Roger D. Quinn, Hillel J. Chiel, and Roy E. Ritzmann. Biologically inspired approaches to robotics. *Communications of the ACM*, (1):30–38, March 1997.

[4] Walter Benenson, John W. Harris, Horst Stocker, and Holger Lutz. *Handbook of Physics*. Springer-Verlag., New York, 2nd edition, 2002.

[5] Antoine Bouthors and Matthieu Nesme. Twinned meshes for dynamic triangulation of implicit surfaces. In *GI '07: Proceedings of Graphics Interface 2007*, pages 3–9, New York, NY, USA, 2007. ACM Press.

[6] R. W. Brankin, I. Gladwell, J. R. Dormand, P. J. Prince, and W. L. Seward. Algorithm 670: a Runge-Kutta-Nyström code. *ACM Trans. Math. Softw.*, 15(1):31–40, 1989.

[7] Robert Bridson, Ronald Fedkiw, and John Anderson. Robust treatment of collisions, contact and friction for cloth animation. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 594–603, New York, NY, USA, 2002. ACM Press.

[8] Robert Bridson, Ronald Fedkiw, and Matthias Muller-Fischer. Fluid simulation: SIGGRAPHy 2006 course notes. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, pages 1–87, New York, NY, USA, 2006. ACM Press.

[9] Mark Carlson, Peter J. Mucha, and Greg Turk. Rigid fluid: animating the interplay between rigid bodies and fluid. *ACM Trans. Graph.*, 23(3):377–384, 2004.

[10] Johnny T. Chang, Jingyi Jin, and Yizhou Yu. A practical model for hair mutual interactions. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 73–80. ACM Press, 2002.

[11] Jeng-Shi Chen, Fan-Tien Cheng, Kai-Tarng Yang, Fan-Chu Kung, and York-Yih Sun. Optimal force distribution in multilegged vehicles. *Robotica*, 17(2):159–172, 1999.

[12] Wenjie Chen, K. H. Low, and S. H. Yeo. Adaptive gait planning for multi-legged robots with an adjustment of center-of-gravity. *Robotica*, 17(4):391–403, 1999.

[13] Yanyun Chen, Xin Tong, Jiaping Wang, Stephen Lin, Baining Guo, and Heung-Yeung Shum. Shell texture functions. *ACM Trans. Graph.*, 23(3):343–353, 2004.

[14] Nuttapong Chentanez, Tolga G. Goktekin, Bryan E. Feldman, and James F. O'Brien. Simultaneous coupling of fluids and deformable bodies. In *Eurographics Symposium on Computer Animation*, pages 83–89, 2006.

[15] Kwang-Jin Choi and Hyeong-Seok Ko. Stable but responsive cloth. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 604–611, New York, NY, USA, 2002. ACM Press.

[16] Alexandre Joel Chorin and Jerrold Eldon Marsden. *A Mathematical Introduction to Fluid Mechanics*. Springer, New York, 1993.

[17] Judith L. Connor and Nora L. Deans. *Jellies: Living Art*. Monterey Bay Aquarium Foundation, Monterey Bay, CA, 2002.

[18] Robert L. Cook and Tony DeRose. Wavelet noise. *ACM Trans. Graph.*, 24(3):803–811, 2005.

[19] R. Hays Cummins. My best pictures and movies from marine ecology. http://www.junglewalk.com/popup.asp?type=v&AnimalVideoID=759, 1999.

[20] John D. Cutnel and Kenneth W. Johnson. *Physics*. John Wiley and Sons, Inc., New York, 3rd edition, 1995.

[21] John O. Dabiri. Unsteady fluid mechanics of starting-flow vortex generators with time-dependent boundary conditions. Ph.D. Dissertation, California Institute of Technology, 2005.

[22] John O. Dabiri and Mortexa Gharib. Sensitivity analysis of kinematic approximations in dynamic medusan swimming models. *Journal of Experimental Biology*, 206:3675–3680, 2003.

[23] T. L. Daniel. Mechanics and energetics of medusan jet propulsion. *Canadian Journal of Zoology*, 61:1406–1420, 1983.

[24] Thomas Dean and Michael Wellman. *Planning and Control*. Morgan Kaufmann Publishers, San Francisco, 1991.

[25] E. J. Denton and T. I. Shaw. The bouyancy of gelatinous marine animals. In *Proc. Physiol. Soc.*, pages 14–15, 1961.

[26] Mathieu Desbrun and Marie-Paule Gascuel. Smoothed particles : A new paradigm for animating highly deformable bodies. In *Computer Animation and Simulation '96*, pages 61–76.

[27] Brendan Dixon and Walter Shen. Jellyfish. Stanford Computer Graphics Laboratory, http://graphics.stanford.edu/courses/cs348b-competition/cs348b-03/jellyfish/, 2005.

[28] J. R. Dormand and P. J. Prince. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980.

[29] J. R. Dormand and P. J. Prince. Runge-Kutta-Nystrom triples. *Comp. and Math. with Appl.*, 13:937–949, 1987.

[30] Kayvon Fatahalian and Tim Foley. Rendering Jellyfish. Stanford Computer Graphics Laboratory, http://graphics.stanford.edu/ kayvonf/cs348b/jellies/, 2004.

[31] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 2001. ACM Press.

[32] Bryan E. Feldman, James F. O'Brien, Bryan M. Klingner, and Tolga G. Goktekin. Fluids in deforming meshes. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2005*, july 2005.

[33] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, and Richard L. Phillips. *Introduction to Computer Graphics*. Addison-Wesley, Boston, 1993.

[34] Alain Fournier and William T. Reeves. A simple model of ocean waves. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 75–84, New York, NY, USA, 1986. ACM Press.

[35] Torsten Fröhlich. The virtual oceanarium. *Commun. ACM*, 43(7):94–101, 2000.

[36] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375–398, 1977.

[37] W. B. Gladfelter. Structure and function of the locomotory system of polyorchis montereyensis (cnidaria, hydrozoa). *Helgolaender Wiss. Meeresunters*, 23:38–79, 1972.

[38] Ian Gladwell, Larry Shampine, and Skip Thompson. *Solving ODEs with MATLAB*. Cambridge University Press, New York, NY, USA, 2003.

[39] Jack Goldfeather and Victoria Interrante. A novel cubic-order algorithm for approximating principal direction vectors. *ACM Trans. Graph.*, 23(1):45–63, 2004.

[40] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer. *Numerical Simulation in Fluid Dynamics: a practical introduction*. Society for Industrial and Applied Mathematics, Philadelphia, 1998.

[41] Eran Guendelman, Andrew Selle, Frank Losasso, and Ronald Fedkiw. Coupling water and smoke to thin deformable and rigid shells. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers on xx*, pages 973–981, New York, NY, USA, 2005. ACM Press.

[42] James K. Hahn. Realistic animation of rigid bodies. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 299–308, New York, NY, USA, 1988. ACM Press.

[43] Ernst Hairer, Christian Lubich, and Michel Roche. *The Numerical solution of differential-algebraic systems by Runge-Kutta methods*. Springer-Verlag, Berlin, 1989.

[44] David Halliday and Robert Resnick. *Fundamentals of Physics*. John Wiley and Sons, New York, 3rd edition, 1988.

[45] Alf Kjartan Halvorsen. *Model-Based Methods in Motion Capture*. Uppsala University Press, Uppsala Universitet, 2002.

[46] Carl S. Hammen. *Marine Invertebrates: Comparative Physiology*. University Press of New England, Hanover, 1980.

[47] F. Harlow and J. Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Phys. Fluids*, 8:2182–2189, 1965.

[48] Jessica K. Hodgins, Wayne L. Wooten, David C. Brogan, and James F. O'Brien. Animating human athletics. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 71–78. ACM Press, 1995.

[49] T. Igarashi, T. Moscovich, and J. F. Hughes. Spatial keyframing for performance-driven animation. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, page 17, New York, NY, USA, 2006. ACM Press.

[50] Doug L. James and Dinesh K. Pai. Bd-tree: output-sensitive collision detection for reduced deformable models. *ACM Trans. Graph.*, 23(3):393–398, 2004.

[51] Young-Min Kang and Hwan-Gue Cho. Complex deformable objects in virtual reality. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 49–56, New York, NY, USA, 2002. ACM Press.

[52] Tae-Yong Kim and Ulrich Neumann. Interactive multiresolution hair modeling and editing. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 620–629. ACM Press, 2002.

[53] Y. Kim and C. Peskin. Penalty immersed boundary method for an elastic boundary with mass. *Physics of Fluids*, 19(5):53103–53121, 2007.

[54] Peter Kipfer and Rüdiger Westermann. Realistic and interactive simulation of rivers. In *GI '06: Proceedings of the 2006 conference on Graphics interface*, pages 41–48, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[55] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458. ACM Press, 1994.

[56] Serge Lang. *Introduction to Linear Algebra*. Springer, New York, 2nd edition, 1986.

[57] Sung-Hee Lee and Demetri Terzopoulos. Heads up!: biomechanical modeling and neuromuscular control of the neck. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 1188–1198, New York, NY, USA, 2006. ACM Press.

[58] Matt Liverman. *The Animator's Motion Capture Guide : Organizing, Managing,Editing*. Charles River Media, New York, 2004.

[59] S. Lloyd. Least square quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[60] Frank Losasso, Frédéric Gibou, and Ron Fedkiw. Simulating water and smoke with an octree data structure. *ACM Trans. Graph.*, 23(3):457–462, 2004.

[61] Hai Mao and Yee-Hong Yang. Particle-based immiscible fluid-fluid collision. In *GI '06: Proceedings of the 2006 conference on Graphics interface*, pages 49–55, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[62] John Marlon and Marlon John. *Focus on Photon Mapping*. Premier Press, 2003.

[63] T. J. Martin, F. R. Pearce, and P. A. Thomas. An Owners Guide to Smoothed Particle Hydrodynamics. Astronomy Centre, Sussex University, Falmer, Brighton, 1993.

[64] R. Mason and J. Burdick. Propulsion and control of deformable bodies in an ideal fluid. In *IEEE International Conference on Robotics and Automation*, pages 773–780, 1999.

[65] William M. Megill. The biomechanics of jellyfish swimming. Ph.D. Dissertation, Department of Zoology, University of British Columbia, 2002.

[66] William M. Megill, John M. Gosline, and Robert W. Blake. The modulus of elasticity of fibrillin-containing elastic fibres in the mesoglea of the hydromedusa polyorchis penicillatus. *Journal of Experimental Biology*, 208:3819–3834, 2005.

[67] Eugene Fiume Michiel van de Panne. Sensor-actuator networks. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 335–342, New York, NY, USA, 1993. ACM Press.

[68] Gavin Miller and Andrew Pearce. Globular dynamics: A connected particle system for animating viscous fluids. *Computers and Graphics*, 13(3):305–309, 1989.

[69] Gavin S. P. Miller. The motion dynamics of snakes and worms. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 169–173. ACM Press, 1988.

[70] Brian Mirtich and John Canny. Impulse-based simulation of rigid bodies. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 181–ff., New York, NY, USA, 1995. ACM Press.

[71] Neil Molino, Zhaosheng Bao, and Ron Fedkiw. A virtual node algorithm for changing mesh topology during simulation. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 385–392, New York, NY, USA, 2004. ACM Press.

[72] J. J. Monaghan. Smoothed particle hydrodynamics. *Annu. Rev. Astron. Astrophys.*, 30:543–574, 1992.

[73] J. J. Monaghan. Implicit SPH drag and dusty gas dynamics. *J. Comput. Phys.*, 138(2):801–820, 1997.

[74] J. J. Monaghan and J. C. Lattanzio. A Refined Method for Astrophysical Problems. *Computer Physics Reports*, 149:135–143, 1985.

[75] David Mould. Image-guided fracture. In *GI '05: Proceedings of the 2005 conference on Graphics interface*, pages 219–226, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society.

[76] David Mould and Michael C. Horsch. An hierarchical terrain representation for approximately shortest paths. In *PRICAI*, pages 104–113, 2004.

[77] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[78] Matthias Müller and Markus Gross. Interactive virtual materials. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 239–246, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[79] Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. Meshless deformations based on shape matching. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 471–478, New York, NY, USA, 2005. ACM Press.

[80] Matthias Müller, Simon Schirm, Matthias Teschner, Bruno Heidelberger, and Markus Gross. Interaction of fluids with deformable solids: Research articles. *Comput. Animat. Virtual Worlds*, 15(3-4):159–171, 2004.

[81] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer-Verlag, New York, NY, USA, 2002.

[82] Dinesh K. Pai, Shinjiro Sueda, and Qi Wei. Fast physically based musculoskeletal simulation. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, page 25, New York, NY, USA, 2005. ACM Press.

[83] Mark Pauly, Dinesh K. Pai, and Leonidas J. Guibas. Quasi-rigid objects in contact. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 109–119, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[84] Ken Perlin. An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296. ACM Press, 1985.

[85] Ken Perlin. Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, New York, NY, USA, 2002. ACM Press.

[86] Ken Perlin and E. M. Hoffert. Hypertexture. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 253–262, New York, NY, USA, 1989. ACM Press.

[87] Charlie Peskin. The immersed boundary method. *Acta Numerica 11*, pages 479–517, 2002.

[88] William Press, Saul Teukolsky, William Vetterling, and Brian Flannery. *Numerical Recipes in C*. Cambridge University Press, New York, NY, USA, 2nd edition, 1992.

[89] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, New York, NY, USA, 1990.

[90] Marc H. Raibert and Jessica K. Hodgins. Animation of dynamic legged locomotion. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 349–358, New York, NY, USA, 1991. ACM Press.

[91] Nick Rasmussen, Duc Quang Nguyen, Willi Geiger, and Ronald Fedkiw. Smoke simulation for large scale phenomena. *ACM Trans. Graph.*, 22(3):703–707, 2003.

[92] Stephane Redon, Nico Galoppo, and Ming C. Lin. Adaptive dynamics of articulated bodies. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 936–945, New York, NY, USA, 2005. ACM Press.

[93] W. T. Reeves. Particle systems: a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[94] S. Russel and P. Norvig. *Artificial Intelligence*. 1st edition, 1995.

[95] Florida Keys National Marine Sanctuary. NOAA's Coral Kingdom Collection: reef2547. http://www.photolib.noaa.gov/htmls/reef2547.htm, 2007.

[96] Timothy Sauer. *Numerical Analysis*. Pearson Addison Wesley, Boston, 2006.

[97] Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. A vortex particle method for smoke, water and explosions. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 910–914, New York, NY, USA, 2005. ACM Press.

[98] L. F. Shampine, I. Gladwell, and R. W. Brankin. Reliable solutions of special event location problems for ODEs. *ACM Transactions on Mathematical Software*, 17(1):11–25, 1991.

[99] C. T. Shih. *A Guide to the Jellyfish of Canadian Atlantic Waters*. Number 5. National Museum of Natural Sciences, Ottawa, Canada, 1977.

[100] Hyun Joon Shin, Lucas Kovar, and Michael Gleicher. Physical touch-up of human motions. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, page 194, Washington, DC, USA, 2003. IEEE Computer Society.

[101] A. N. Spencer. The physiology of a coelenterate neuromuscular synapse. *Journal of Comparative Physiology*, 148:353–363, 1982.

[102] Jos Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[103] Jos Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, 2003.

[104] John M. Stockie and Brian R. Wetton. Analysis of stiffness in the immersed boundary method and implications for time-stepping schemes, 1999.

[105] B. K. Sullivan, C. L. Suchman, and J. H. Costello. Mechanics of prey selection by ephyrae of the scyphomedusa aurelia aurita. *Marine Biology*, 130(2):213 – 222, December 1997.

[106] Evan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55, 1974.

[107] Gary Taubes. Biologists and engineers create a new generation of robots that imitate life. *Science Magazine*, (1):80–83, April 2000.

[108] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. Elastically deformable models. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 205–214. ACM Press, 1987.

[109] Demetri Terzopoulos, John Platt, and Kurt Fleisher. Heating and melting deformable models (from goop to glop). In *Proceedings of Graphics Interface'89*, pages 219–226, 1989.

[110] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, Laks Raghupathi, A. Fuhrmann, Marie-Paule Cani, François Faure, N. Magnetat-Thalmann, W. Strasser, and P. Volino. Collision detection for deformable objects. *Computer Graphics Forum*, 24(1):61–81, March 2005.

[111] Christian Theobalt, Joel Carranza, Marcus A. Magnor, and Hans-Peter Seidel. Enhancing silhouette-based human motion capture with 3d motion fields. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, page 185, Washington, DC, USA, 2003. IEEE Computer Society.

[112] David Tonnesen. Modeling liquids and solids using thermal particles. In *Proceedings of Graphics Interface'91*, pages pages 255–262, 1991.

[113] Xiaoyuan Tu and Demetri Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. *Computer Graphics*, 28(Annual Conference Series):43–50, 1994.

[114] Munetoshi Unuma, Ken Anjyo, and Ryozo Takeuchi. Fourier principles for emotion-based human figure animation. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 91–96. ACM Press, 1995.

[115] Xiaoming Wei Wei Li, Zhe Fan and Arie Kaufman. GPU-Based Flow Simulation with Complex Boundaries. Technical Report 031105, Computer Science Department, SUNY at Stony Brook, Nov.

[116] Richard S. Wright and Michael Sweet. *OpenGL Superbible*. Waite Group Press, Corte Madera, CA, 2nd edition, 1999.

[117] Jia–chi Wu and Zoran Popović. Realistic modeling of bird flight animations. *ACM Trans. Graph.*, 22(3):888–895, 2003.

[118] Jianmin Zhao and Norman I. Badler. Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Trans. Graph.*, 13(4):313–336, 1994.

[119] Victor Brian Zordan and Jessica K. Hodgins. Motion capture-driven simulations that hit and react. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 89–96, New York, NY, USA, 2002. ACM Press.