Real-time and Efficient Rendering of Deformable Bodies

LAZUNIN, Vladimir

(開始ページ / Start Page)
1
(終了ページ / End Page)
83
(発行年 / Year)
2016-09-15
(学位授与番号 / Degree Number)
32675甲第609号
(学位授与年月日 / Date of Granted)
2016-09-15
(学位名 / Degree Name)
博士(理学)
(学位授与機関 / Degree Grantor)
法政大学 (Hosei University)
(URL)
https://doi.org/10.15002/00013343

Doctoral Dissertation Reviewed by Hosei University

Real–time and Efficient Rendering of Deformable Bodies

Vladimir Lazunin

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor professor V. Savchenko for guiding me in my research. I also want to express my gratitude to professors N. Koike, T. Koike, S. Fujita and T. Wakahara for their invaluable and timely feedback regarding this thesis.

Chapter 1

Overview of existing methods of deformable body simulation and 3D visualization

1.1 General Introduction

Deformable bodies form a wide category of objects important for numerical computation in general, and computer graphics, in particular. In fact, most, if not all real life objects can be qualified as "deformable" – however, when it comes to computational models, the deformability aspect is not as widely used. The reasons are many: for some cases elasticity is not big enough to be important, and therefore using only rigid body dynamics is justified. In many cases, however, elasticity is essential, but still cannot be used due to forbiddingly high computational cost. Some real life objects are treated differently, depending on a particular application: for example, a car may be treated as a rigid body when simulating physical interactions for a computer game, but as a deformable body for a numerical crash simulation. The complexity of underlying physical simulation methods may vary accordingly: for a game, simple – and not too physically accurate – bouncing calculated with rigid body dynamics would be enough, while very accurate deformations are calculated for a crash test using, for example, the finite element method (FEM). Between such extreme cases lies the area where only some aspect of the deformable behavior matter, or no high precision is required as

long as the visualization result looks convincing. This includes movies, virtual reality (VR) and augmented reality applications, video games and so on. Good examples are clothing and hair simulation: the numerical models vary greatly, from very simple ones with real–time performance, used in games and VR applications, to very complicated, which may require hours of computation to provide realistic results to be used in a movie scene. In games, for example, human characters design is still very much restricted in the ways that allow treating clothing and hair as rigid, inseparable parts of the body, because more realistic cloth simulation would require more computations as well as higher resolution models (see Fig. 1.1 for a simple example).

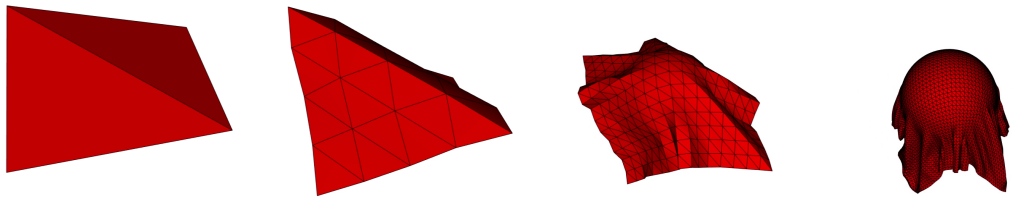

Figure 1.1: Level of details and corresponding deformability: only two triangles are enough to represent a flat sheet of rigid material (left), that cannot be deformed realistically without a sufficient increase in polygon count (right).

This chapter provides a survey of the methods that are used for deformable body simulation and, in more general terms, deformable body representation in computer graphics domain, as well as a survey and general descriptions of visualization methods used in computer graphics. Problem statements and the goals of the thesis are formulated in this chapter. Surveys of more specific techniques, tied more closely to the problems solved in the thesis, are provided in their respective chapters.

1.2 General classification of methods used for deformations simulation

To summarize the historical review given above, here is a broad categorization of methods used for simulating deformable bodies in computer graphics. More technical details, as well as further references regarding each category can be found, for example, in a technical report of Gibson and Mirtich [3].

There are more than one way to classify the methods, but, perhaps, the most important divisions are

implicit/parametric and *discrete/continuous*. In implicit methods the deformations are not applied directly to the body of interest, but rather the space around it, or a primitive the body is embedded into, changes the shape, implicitly causing the body to deform accordingly. In parametric methods, on the other hand, the deformations are introduced directly into the equations of the deformable body as parameters (hence the name "parametric"). In discrete methods, the body shape is approximated by a discrete set of elements, and all deformations are applied only to those elements. In continuous methods, the body is treated as a monolithic entity, with its properties continuously changing throughout the body. It is important to note, however, that this distinction between discrete and continuous methods applies only to the representation of the body itself — it has nothing to do with the methods used to solve underlying equations: numerical methods will be discrete for both cases.

Curves, Splines and patches

Catenary curves, introduced by Leibniz, Huygens and Bernoulli in 17th century to describe shapes of a hanging chain, are used in some cloth simulations. Obviously, this approach is quite limited, but computationally efficient. The equation of a catenary in Cartesian coordinates is

$$y = a\cosh(\frac{x}{a}) = \frac{a}{2}(e^{x/a} + e^{-x/a}) \tag{1.1}$$

Bézier curves and surfaces, B–splines, rational B–splines, non–uniform rational B–splines and many other methods of specifying curves and surfaces with a relatively small number of control points are widely used in computer–aided design (CAD) to model deformable objects. They allow a high degree of control and are computationally efficient to deform (but not to visualize, for that purpose they are usually split into sets of flat triangles). However, they are not very suitable for automatic deformations which occur, for example, in response to pressure or collision, and may require a lot of manual labour to adjust the control points to achieve desired shape changes.

Free–form deformation

Free–form deformation (FFD) deforms an object by deforming the space in which the object lies. It is a very general technique and can be applied to many graphical representations, such as polygons, splines, parametric patches and implicit surfaces. In general, it can be expressed as a space mapping $f : R^3 \to R^3$. Because the possible regions and types of deformations are limited, the object is usually embedded into a cubical or cylindrical lattice of grid points, forming a set of three–dimensional cells $\{U_i\}$. The free–form deformation thus becomes a collection of mappings in the form $f_i : U_i \to R^3$.

Mass–spring systems

Mass–spring systems, as the name suggests, is a physically–based technique of approximating a deformable object by a set of mass particle, interconnected by springs, usually in a lattice pattern. The spring forces may be linear, according to the Hooke's law, or non–linear, to model inelastic tissues. During the simulation process, the sum of forces is calculated for every mass point. It includes forces exerted by the springs connected to the point, as well as the external forces, if applicable. The point is, then, moved according to Newton's Second Law.

With mass–spring system, real–time performance can be achieved with today's commodity computers. They are easy to construct and animate, and are widely used in deformable body simulation, such as facial and cloth simulation and so on. It is, however, difficult to model some of the aspects of soft tissues, such as incompressibility. This problem can be solved by introducing additional forces to preserve the volume, or even simply additional springs — obviously, however, that increases the computational cost. Mass–spring system can also exhibit poor stability for nearly rigid bodies.

Finite element method

While the finite element method (FEM) is a numerical technique for finding approximate solutions to boundary value problems for partial differential equations in general, and as such can be used in modeling of wide

variety of physical phenomena, in the context of this work we are going to discuss it with regard to deformable body simulations only. Unlike mass–spring models, which are discrete by nature, more accurate physical techniques model deformable object as a continuum, with mass and energy existing not only in the nodes, but distributed throughout the object. The models are derived from equations of continuum mechanics, however, it is important to note that the numerical methods used to solve the equations are still discrete. Potential energy of a deformable body is given by:

$$\Pi = \Lambda - W, \tag{1.2}$$

where Λ is the total strain energy of the deformable objects, and W is the work done by external forces on the deformable object. The object reaches equilibrium when its potential energy is minimal. To determine the equilibrium shape of the body, both Λ and W are expressed in terms of the object deformation. The potential energy reaches its minimum when the derivative of Π with respect to the material displacement is zero. Because it is not always possible to find an analytic solution, FEM divide the object into a set of elements (such as triangles and quadrangles in 2D, tetrahedrons and hexahedrons in 3D) and approximate the equilibrium equation over each element. FEM is much more physically realistic than mass–spring systems, but it also has much higher computational requirements. Therefore, it is mostly used in applications where precision is valuable and real–time performance is not required, such as mechanical engineering, an iconic application being a crash test.

Approximate continuum models

These are physically–motivated models, where some laws of physics are used to achieve desired effects, but usually strict adherence to the laws of physics is not the goal. Active contour models, or "snakes" are used to find contours of static or moving objects. Snakes are deformable bodies that respond to external forces (for example, image intensity to find contours) and resist to stretching and bending, thus finding a compromise between, for example, points with high gradient on an image and a curve with minimal bending energy —

this effectively attracts the snake to image edges, while keeping its shape smooth. There are other continuum models for deformable curves, surfaces and solids, used in animation applications.

Low degree of freedom models

Discretization of a physically–based model into a large number of nodes characterized by their masses, positions and velocities leads to systems with many degrees of freedom. Such general systems support a wide variety of deformations, but they are slow to simulate. Low degree of freedom models limit the deformable object to fewer degrees of freedom, sacrificing generality for speed. This is achieved, for example, by separating geometry and dynamics, as in [4], or using constraints to connect globally deformed non–rigid pieces into complex models, as in [5]. There are also techniques to add physical behavior to parametric surface patches by minimizing an energy functional defined on the surface. The surface is restricted to a class by control point locations and weights, and the methods of constrained optimization are used to find a state vector that minimizes the energy functional while satisfying the constraints.

1.3 General overview of 3D rendering techniques

3D rendering is, essentially, the process of creating a 2D image from a 3D scene. The earliest examples of 3D rendering used simple wireframe representation of objects with different techniques for hidden lines removal and without any shading. As the computational power grew, *shading* became possible to generate more realistic images (although wireframe–based rendering is still in use today). In 1970 Bouknight developed LINESCAN algorithm [6] for producing computer generated half–tone presentations of three–dimensional polygonal surface structures. A technique called *flat shading* shades each polygon based on its surface normal and the direction of the light source. More advanced smooth shading techniques were introduced by Gouraud in 1971 [7] and Phong in 1975 [8] — in contrast to flat shading, with these techniques the color changed from pixel to pixel, not from polygon to polygon. *Texture mapping* for 3D graphics was first used by Catmull [9]. Blinn [10] introduced a technique now called *bump mapping*, which allowed to simulate small dents and

wrinkles on the surface.

To add more realistic lighting to 3D scenes, different numerical approximation to the *rendering equation* are used. The rendering equation, introduced independently in works of Kajiya [11] and Immel [12] is:

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_e(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) + \int_{\Omega} f_r(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t)\, L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)\, (\omega_{\mathrm{i}} \cdot \mathbf{n})\, \mathrm{d}\,\omega_{\mathrm{i}} \tag{1.3}$$

where

- λ is a particular wavelength of light
- t is time
- $\mathbf{x}$ is the location in space
- $\mathbf{n}$ is the surface normal at that location
- ω_{o} is the direction of the outgoing light
- ω_{i} is the negative direction of the incoming light
- $L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t)$ is the total spectral radiance of wavelength λ directed outward along direction ω_{o} at time t, from a particular position $\mathbf{x}$
- $L_e(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t)$ is emitted spectral radiance
- Ω is the unit hemisphere centered around $\mathbf{n}$ containing all possible values for ω_{i}
- $\int_{\Omega} \ldots \mathrm{d}\,\omega_{\mathrm{i}}$ is an integral over Ω
- $f_r(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t)$ is the bidirectional reflectance distribution function, the proportion of light reflected from ω_{i} to ω_{o} at position $\mathbf{x}$, time t, and at wavelength λ
- $L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)$ is spectral radiance of wavelength λ coming inward toward $\mathbf{x}$ from direction ω_{i} at time t

- $\omega_\mathrm{i} \cdot \mathbf{n}$ is the weakening factor of inward irradiance due to incident angle, as the light flux is smeared across a surface whose area is larger than the projected area perpendicular to the ray, often written as $\cos\theta_i$.

Solving this equation is the primary challenge in realistic rendering. One of the approaches is based on finite element method and called *radiosity algorithm*. Developed in 1950 for modeling of heat transfer, it was adapted to 3D rendering in the work of Goral et al. [13] Their method modeled the interaction of light between diffusely reflecting surfaces and allowed to model effects such as diffuse light sources, soft shadows and "color–bleeding" effect caused by diffuse reflections. Another approach, based on Monte Carlo methods, has led to methods such as photon mapping, path tracing and Metropolis light transfer.

Although very general, the equation does not model the whole range of lighting effects — for example, phenomena such as light transmission or subsurface scattering are not taken into account. The rendering equation was further generalized into a volume rendering equation by Kajiya [14].

Two major techniques used today to convert an illuminated, shaded and textured 3D scene into its final 2D representation on the screen surface are *ray tracing* and *polygon rasterizing*. Polygon rasterization is based on projecting vertices of the polygons the scene is made of onto the screen surface, then using interpolated values for lighting, shadowing, textures and so on inside the projected polygons. Rasterization sacrifices visual realism for speed, and is typically used for interactive applications, such as games. Usually there is specialized hardware for rasterization inside today's *graphic processor units* (GPU), available through standardized interfaces, such as OpenGL and DirectX.

Ray tracing, or calculation of paths and intersections of rays, waves or particles with different media is used in several fields of science and engineering, for example, to calculate sound, radio, seismic and light waves propagation and optical system design. It is also a computer visualization technique based on similar principles of light propagation, and further in this work we use the term *ray tracing* only in relation to visualization. Ray tracing is mainly used for more realistically looking, but not real–time visualization. It is based on an idea of rays cast from the eye, through the screen plane and onto the scene. Speaking in terms

of today's digital displays, basically, one ray is cast per screen pixel, from the eye and towards that pixel, and then further into the scene. The pixel is then colored according to the object the ray intersects, as well as any additional information that may be present (light sources, shadowing, textures and so on). Obviously, that is the opposite of what happens in nature, where light travels from the light sources, gets reflected from the objects and then part of it comes into the eye, but in many cases such "change of direction" is justified, because the light that does not enter the eye is of no interest.

Ray tracing for 3D visualization was introduced in 1968 by Appel [15]. His goal was to improve line–based printouts produced by digital plotters by adding shadowing. The next big advancement of that technique was done in 1979 by Whitted [16]. He introduced *recursive ray tracing*, where each ray hitting an object can cast several new rays of three types: reflection, refraction and shadow rays. A reflection ray travels in the mirror–reflection direction, a refraction ray's direction is determined by its parent's direction and the refractive properties of the transparent materials. Shadow rays are traced toward each light and determine the illumination intensity. The contribution of each secondary ray is then taken into account when calculating the resulting color of the screen pixel. Reflected and refracted rays, when hitting other objects, may produce their own reflection, refraction and shadow rays, and so on, in a recursive manner. This algorithm is widely used in ray tracing today, often being called "whitted–style ray tracing". In early works, term "ray casting" was also used interchangeably with "ray tracing", but recently "ray casting" is mostly used to call simple forms of ray tracing, that do not recursively trace secondary rays.

Although it is more economical to trace rays from the eye (and tremendously wasteful, if at all possible, to trace every ray from every light source), not every phenomenon can be represented with this approach. For example, *caustics*[1] cannot be rendered that way. Therefore, algorithms that shoot rays from light sources as well as from the eye were also developed. These algorithms are sometimes called *backwards ray tracing*, although ray directions are not "backwards". Terms *eye–based ray tracing* and *light–based ray tracing* are also used.

Photon mapping, a two–pass global illumination algorithm, was introduced by Jensen [17]. At the 1st

[1] Bright patterns of light caused by focusing from wide surfaces onto a narrow area

pass, light packets called *photons* are cast from the light sources, and their intersections with the scene objects are stored in *photon maps*. At the 2nd pass, the rendering equation is solved to calculate the surface radiance. With this approach, caustics, diffuse interreflection and subsurface scattering can be added to the rendered scene. Although originally designed to work with ray tracers, photon mapping can be extended to work with polygon rasterizing algorithms.

1.4 Problem statement and the goal of the thesis

There are two major application domains for deformable body simulations. One is where they mainly serve artistic purposes, such as painting, sculpting and various kinds of 3D modeling. There simulation techniques are creative instruments, and although both physically and non–physically–based techniques may be employed, in general obeying the laws of physics is not the goal. The other domain is where they are used to simulate deformation and elasticity phenomena seen in real life. The methods used to simulate such deformations are usually physically–based.

The goal of our research is to develop fast, real–time techniques for rendering practically important cases of deformable bodies used for real life phenomena simulations and today dominated by purely physically–based methods. Although physics simulation is good for generality, in many cases (for example, multiple garment simulation) physically–based methods are too slow to achieve both satisfactory visual appearance and real–time speed at the same time.

This thesis presents a new hybrid technique based on three key parts:

1. Simplified physical simulation. We neither develop new physically–based methods nor contribute to any existing physically–based method. The key difference between our approach and most others is that, instead of making the physical simulation more sophisticated, we use simpler methods that have already been proven to be fast. For a big part of this thesis we used NVidia PhysX library, but our approach is not tied to it in any particular way, so other libraries (for example, Bullet Physics) or physically–based methods can be used as well.

2. Specialized recursive ray tracing–based visualization. While simple physical simulation is fast, by itself it certainly cannot achieve good results in most cases (especially since many of those cases would be problematic even for more sophisticated physically–based methods). A lot of problems remain unresolved by the physical simulation, in form of relatively small numerical errors, resulting in incorrect visualization. In our work we extend capabilities of recursive ray tracing to be able to detect and correct such errors in real–time. We call our method *layered ray tracing*.

3. Radial basis function (RBF)–based geometrical deformations. With simplified physics providing results that are *mostly* physically correct, and layered ray tracing correcting small (but numerous) simulation errors, there are still some cases that cannot be handled by those two techniques only. Such, relatively rare, cases are too problematic for the physically–based methods, producing simulation errors too big to be corrected by the layered ray tracing in a realistic–looking way. RBF–based geometrical deformations are employed for such cases to improve the results of the physical simulation before layered ray tracing is applied.

This thesis is organized as follows. Chapter 2 explains the techniques our work is based upon, as well as describes our general approach to the problem. RBF space mapping and recursive ray tracing, used throughout this work, are presented there. A technique for non–physically–based deformation of solid bodies is presented in Chapter 3. A case study of evolutionary optimization of a jellyfish is used in that chapter to validate the proposed technique. Chapter 4 presents a technique for multi–layered garment rendering, a practical case of a more general problem of cloth–cloth interference. Additional introductory sections are present at the beginning of chapters 3 and 4 to explain problems more specific to the techniques presented there: fluid–solid interaction and marine animals study in Chapter 3; garment modeling and textile simulation in Chapter 4.

Chapter 2

General methodology and approach to rendering deformable bodies

2.1 Introduction

In this chapter we present a set of general techniques we developed through the course of this work and used in later chapters. Our general approach is based on modification, adaptation and combination of three major techniques: physical simulation, RBF–based space mapping and recursive ray tracing.

The rest of this chapter is organized as follows: in the sections 2.2 – 2.4 we introduce physical simulation, RBF space mapping and recursive ray tracing, respectively, both in general terms and in the context of our work, comparing to other methods, if possible, and stating our contribution, where applicable. Then, in the sections 2.5 and 2.6 we discuss spatial hierarchical structures and GPU acceleration, applicable to both physical simulation and ray tracing. We give a brief summary of this chapter in section 2.7.

2.2 Physical simulation

There are several primary physical concepts to consider related to deformable body simulation: *elasticity*, *stress* and *strain*. Elasticity is the property of a body which allows it to restore its original shape after the

forces which caused deformations are removed. The stress is the force applied to the body divided by the area of the surface to which it is applied. It is, essentially, same as *pressure*, and can be measured in the units of pressure, *pascals*, where one pascal is equal to one newton of the force applied over one square meter of surface. The strain is the relative deformation of a body, caused by stress, e. g. for an elastic rod of length L changing its length by ΔL due to the force pulling on the end, the strain is $\Delta L/L$. Material's elasticity is described by a *stress–strain* curve, which shows the relationship between stress and strain. Three primary characteristics related to stress and strains are:

- *Young's modulus* describes how an object deforms along an axis when opposing forces are applied along that axis.
- The *shear modulus* is an object's tendency to deform without changing its volume and surface area.
- The *bulk modulus* describes how an object deforms in all directions when uniformly loaded in all directions, changing its volume.

For most materials the stress–strain relationship is linear for small deformations and can be described by Hooke's law. If stress is higher than the *elastic limit* for the material, the stress–strain relationship is no longer linear, and for even higher stresses the material exhibits *plastic behavior*, not restoring its original shape after the stress is removed. Because continuum mechanics equations are too computationally expensive to solve in real–time, simpler models, such as mass–spring systems are typically used for an approximation.

In addition to integrating equations of the chosen mechanical model of deformation, collision detection is crucial for realistic interaction between deformable (and rigid) bodies. While collision handling may be easy for small datasets of simple geometry, as the number of colliding objects and their complexity (for example, polygon count) grow, comprehensive methods for ensuring correct collision handling become impractical to implement. This is not just because the computation time becomes too big. Other problems include numerical errors, poor stability, slow convergence, geometrical singularities and oscillatory solutions. Furthermore, in many cases (notably, multiple cloth simulation, which we discuss in more details in chapter

4) the system tends to enter physically incorrect states, which may require special handling (this is well illustrated by Volino and Magnenat–Thalmann [18]).

The approach we take in this work is not to develop a new physically–based method that does not have some of the aforementioned problems and limitations of existing physically–based methods, but to use alternative techniques to circumvent those problems and limitations. For this reason we tried to use third party physical engines whenever possible, instead of implementing physical simulations by ourselves. We used NVidia PhysX engine for basic cloth simulation in chapters 3 and 4: it provided simple physical behavior, sufficient for simpler applications, such as games, but not good enough for more sophisticated problems we are trying to solve. Therefore, we did not have to implement physical aspects of cloth simulation, rigid body simulation for the rest of the scene, forces such as gravity and friction, collision detection/response and so on — all that we got for free from PhysX. We then used RBF space mapping and recursive ray tracing to geometrically and visually improve those results.

2.3 RBF space mapping

A radial basis function (RBF) is a real–valued function whose value depends solely on the distance from a specified origin point, so that

$$\phi(x, c) = \phi(\|x - c\|), \tag{2.1}$$

where c is the origin. Any function ϕ, satisfying the equation 2.1 is a radial basis function. Their sums are often used to approximate other functions. The approximation has the form

$$y(\mathbf{x}) = \sum_{i=1}^{N} w_i \, \phi(\|\mathbf{x} - \mathbf{x}_i\|), \tag{2.2}$$

where the function $y(x)$ is represented as a sum of N radial basis functions, associated with different origins x_i. The coefficients w_i, called *weights*, can be found through the linear least squares method. To build this kind of approximation, we don't have to know the target function, only a set of its values at some

points. In this sense such approximation can also be regarded as a simple single–layer neural network, and used in time series prediction and control of non–linear systems.

We use these properties of RBF approximation in our work to create mapping functions, used for implicit space deformation. We consider a mapping function as a thin-plate interpolation. For an arbitrary area Ω, the thin–plate interpolation is a variational solution that defines a linear operator T when the following minimum condition is used:

$$\int_{\Omega} \sum_{|\alpha|=m} m!/\alpha!(D^{\alpha}f)^2 d\Omega \rightarrow min, \tag{2.3}$$

where m is a parameter of the variational function and α is a multi-index. It is equivalent to using the RBFs $\phi(r) = rlog(r)$ or r^3 for $m = 2$ and 3 respectively, where r is the Euclidean distance between two points.

The volume spline $f(P)$ having values h_i at N points P_i is the function

$$f(P) = \sum_{j=1}^{N} \lambda_j \phi(|P - P_j|) + p(P), \tag{2.4}$$

where $p = \nu_0 + \nu_1 x + \nu_2 y + \nu_3 z$ is a degree-one polynomial. To solve for the weights λ_j we have to satisfy the constraints h_i by substituting the right part of Equation (2.4), which gives

$$h_i = \sum_{j=1}^{N} \lambda_j \phi(|P_i - P_j|) + p(P_i). \tag{2.5}$$

λ and ν are the coefficients that satisfy a linear system $Tx = b$, where

$$\begin{aligned} T &= \begin{bmatrix} A & B^T \\ B & D \end{bmatrix}, \\ x &= [\lambda_1, \lambda_2, ..., \lambda_N, \nu_0, ..., \nu_3]^T, \\ b &= [h_1, h_2, ..., h_N, 0, 0, ..., 0]^T \end{aligned} \tag{2.6}$$

Matrix T consists of three blocks: a square sub–matrix $A = [\phi(|P_i - P_j|)]$ of size $N \times N$, a zero sub–matrix $D = 0$ of size 4×4 in 3D case or 3×3 in 2D case, and a sub–matrix $B = p(P_i)$ of size $N \times 4$ ($N \times 3$ for 2D cases). For 2D and 3D cases we call $f(P)$ a volume spline. An important property of such interpolation

is its "smoothness", which means that the bending energy is minimal, as shown, for example, by Carr et al. [19]. We use this property for realistically–looking deformations, completely replacing physically–based simulation with RBF space mapping for jellyfish deformations in chapter 3, and combining it with simplified physical simulation of cloth in chapter 4.

2.4 Recursive ray tracing

General description and comparison with polygon rasterization

Rendering techniques widely used today fall in two categories: ray tracing and polygon rasterization. Ray tracing is based on simulating rays of light travelling from light sources and into the eye. It is a very general technique which can be used to accurately portray a wide range of rendering effects, such as shadows, reflection and refraction, dispersion and so on. Its usage is not limited to polygonal models, as it can be used to visualize any object for which a ray intersection point can be computed: analytical functions of all kinds in their explicit or implicit form, voxel data and so on.

Rasterization is based on projecting vertices of a 3D polygon (typically a triangle) onto the 2D screen surface, interpolating textures and rendering effects inside the projected triangle and performing a depth test to discard the invisible pixels (similar techniques exist for voxel–based rendering as well). It lacks the generality of ray tracing, so it cannot portray rendering effects accurately – instead, software developers have to invent a new "trick" every time they want to implement a new effect in their graphical application. For example, to create a reflective surface in a ray tracing application, we can simply and literally reflect a ray from the surface – in an application that uses rasterization, we have to render the entire scene once again, from a different camera position and angle, then clip the resulting image and project it onto our reflective surface, blending it with the surface textures. To create a shadow, in a ray tracing application we cast secondary rays towards the light sources – in a rasterizing application we have to employ a variety of special techniques, such as shadow volumes. As a result, ray tracing–based techniques can produce far more realistic results, but they are, generally, much slower and, therefore, used in non–interactive applications, such as

generation of photorealistic images or movie scenes. There are recent developments in GPU–accelerated ray tracing, which bring it to real–time speed by means of general purpose GPU computing (GPGPU), but rasterization is still much faster. Rasterization–based rendering is usually implemented in hardware of modern GPU, used through OpenGL/DirectX and dominates the area of interactive applications, such as games and virtual reality simulators. Even though the resulting rendering may be achieved quite fast and look quite convincing, it is a general wisdom that there's no such thing as too good or too fast rendering. No matter how performant a GPU is and how beautiful the special effects are – until it has "better resolution than the real world", there's always something that can be added, making the GPU struggle with the rendering. Techniques such as level of details (LOD) and spatial subdivision (bounding volume hierarchies (BVH), kd–trees) are used with both ray tracing and rasterization to decrease computational costs of the rendering. For example, lower LOD models are often used to represent objects that are far away from the projection plane, because when rasterized on the screen, their higher level details would become indistinguishable anyway.

Both ray tracing and rasterization are considered highly parallelizable, but in different ways. Rasterization can use data coherence to share computation between pixels (for example, neighboring pixels of a same flat triangle usually have the same textures, shaders and so on). In ray tracing, on the other hand, each ray is completely independent even from its closest neighbor, and they may end up hitting completely different objects in completely different areas of the scene, which requires very divergent control flow of the underlying shading programs.

It does not, however, mean that rasterization is always faster than ray tracing: there are cases where ray tracing outperforms rasterization. Consider, for example, a very big polygonal model: using a spatial subdivision to accelerate ray–polygon intersection, ray tracing computational complexity grows logarithmically with the increasing number of polygons. Computational complexity of rasterization, on the other hand, will grow linearly with the increasing number of polygons. Consequently, rasterization may outperform ray tracing on smaller scenes, because it typically needs less computations per polygon; however, as the number of polygons grows, so does the computational cost, because all polygons need to be rasterized before performing the depth test. It is demonstrated in works of Wald et. al [1] [20], where they used ray tracing to visualize

Figure 2.1: Sunflowers (1 billion triangles) and Boeing 777 (370 million triangles) from the works of Wald et al. [1], [2]. Scenes of such complexity are much easier to render with ray tracing than with rasterizing.

scenes like a very detailed model of a Boeing 777 aircraft (350 million triangles) or a sunflower field (about 1 billion triangles) (see Fig. 2.1). Another example where ray tracing would probably be faster is a scene with reflective object: with rasterization, it would be required to render the entire scene (or at least big portions of the entire scene) from a different perspective for every reflective surface to generate reflections (even in the very limited way that rasterization allows). For more details on computational complexity of ray tracing see, for example, the work of Reif et al. [21].

Theory and implementation details

Ray tracing in its simplest form (often called "ray casting" nowadays to distinguish it from recursive ray tracing) works as follows (Fig. 2.2):

- An imaginary ray is cast from the "camera" towards the 3D scene, passing through a particular pixel coordinates on the computer monitor.
- The ray is checked for intersection with every object of the 3D scene.

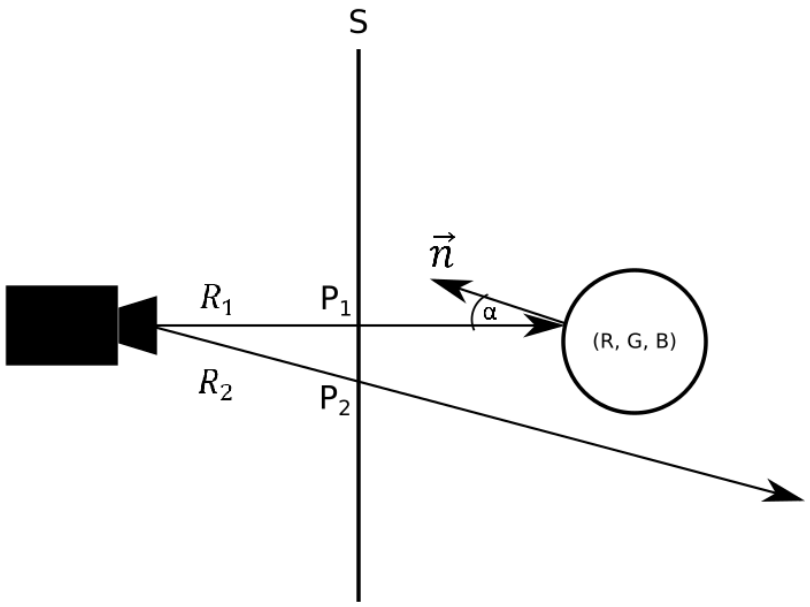

Figure 2.2: Scheme of simple, non–recursive ray tracing. Two rays are shown: R_1 hitting and R_2 missing the scene object (the circle). Solid line S represents the screen. Using cosine law, the screen point P_1 is going to have the same (R, G, B) color as the object material, multiplied by $\cos\alpha$, where α is the angle between $-R_1$ and the object's surface normal at the hit point.

- If no intersection is found, a background color value is assigned to the screen pixel (for example, black color).
- If one or more intersections are found, the closest intersection point is determined. With the ray represented as an origin point P_o and a direction vector $\overrightarrow{V}$, that is the intersection point $P = P_o + t\overrightarrow{V}$ for which t is minimal.
- Color for the screen pixel is calculated based on the material properties of the intersected object as well as any other data, such as the distance from the camera, the surface normal at the intersection point and so on (Fig. 2.3).

With this scheme, only ambient illumination can be used, with no additional light sources and no shadows. Simple (e. g. absolute) transparency can be used, but light attenuation, shadows, reflection and refraction cannot.

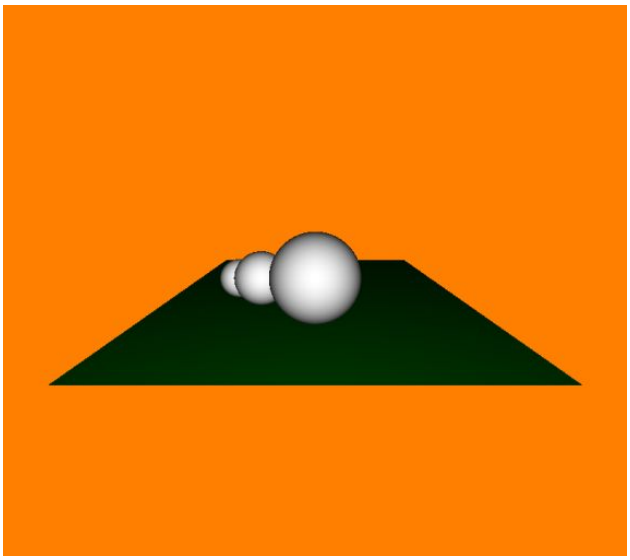

Figure 2.3: Simple non–recursive ray tracing of a green plane and three white spheres. Orange is used as the background color, and cosine law is used for shading.

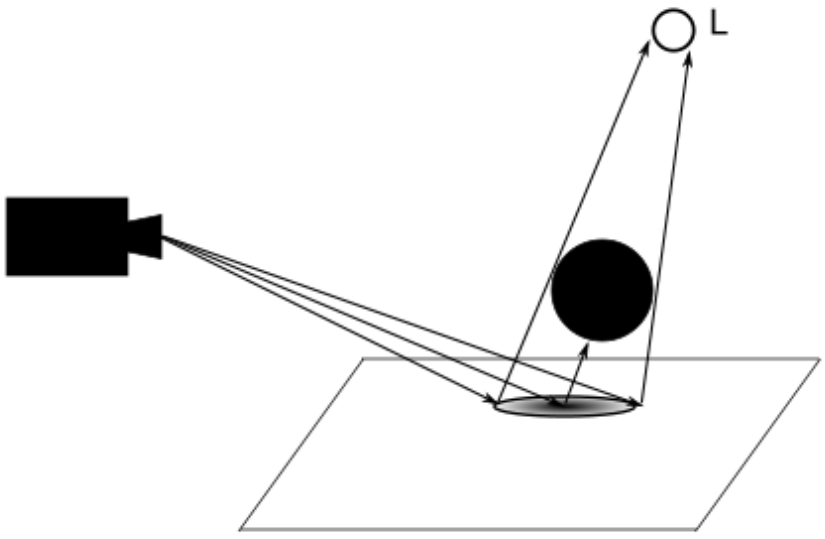

Figure 2.4: Shadow rays in recursive ray tracing. Primary rays are cast from the camera, spawning secondary shadow rays, originating at their points of intersection with the ground plane and directed towards the light source L. Where these shadow rays are blocked by scene objects and unable to reach the light source, a shadow is created.

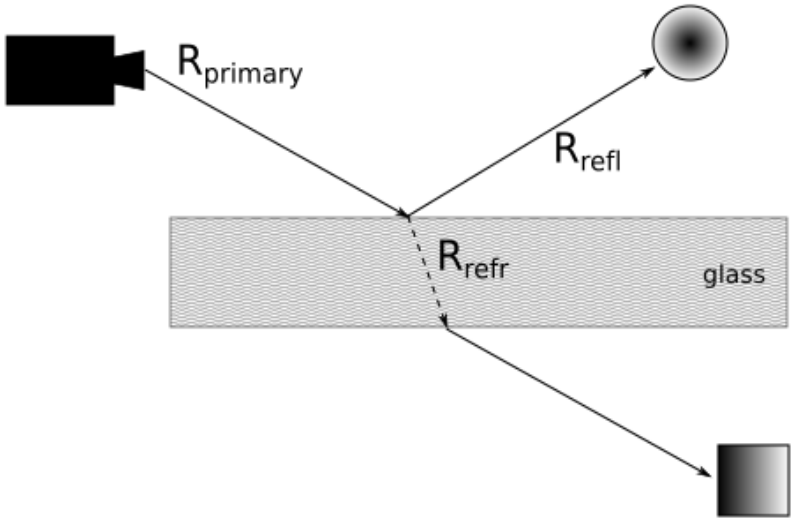

Figure 2.5: Reflection and refraction rays in recursive ray tracing. Primary ray $R_{primary}$ spawns two secondary rays: a reflection ray R_{refl} and a refraction ray R_{refr}. Possibly intersecting other scene objects, these ray bring back some additional color information, which is then blended according to the selected coloring scheme, creating reflection and refraction.

Recursive ray tracing, introduced by Whitted [16], extends the ray casting scheme described above with possibility of casting secondary rays from the intersection points. Those rays include:

- Shadow rays, cast towards every light source in the scene (see Fig. 2.4).
- Reflection rays, cast if the material at the current intersection point has reflective properties (see Eq. 2.9 below).
- Refraction rays, cast if the material at the current intersection point is not completely opaque. Such rays change direction when the material changes its refractive index (see Eq. 2.12) and can get attenuated, if the material is not completely transparent (Eq. 2.14).

Each secondary ray in its turn can produce more secondary rays when intersecting a scene object, although for practical reasons a recursion limit is usually imposed. With recursive ray tracing, different kinds of additional light sources can be used. Shadows, partial transparency, reflection and refraction can be modeled

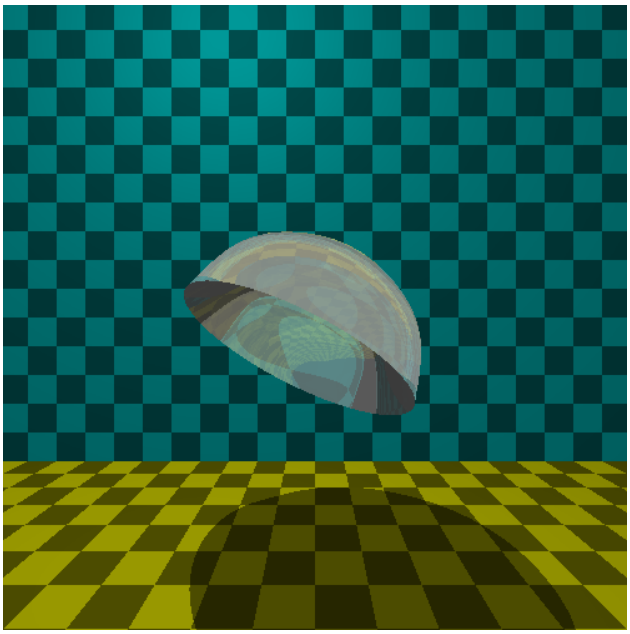

Figure 2.6: Recursive ray tracing of a semi–transparent colorless bell and two planes with procedural checkered textures. Reflection, refraction and light attenuation can be seen on the bell, and a shadow effect — on the floor underneath it.

in a very unified and general way (Fig. 2.6). Visual effects such as caustics still cannot be modeled, requiring more advanced techniques, such as photon mapping.

For shadowing we used Blinn–Phong shading model with the following parameters:

- k_a, which is an ambiend reflection constant
- k_d, which is a diffuse reflection constant
- k_s, which is a specular reflection constant
- *alpha*, which is a material shininess constant

For m light sources, the illumination of each surface point I_p is then defined as

$$I_{\mathrm{p}} = k_{\mathrm{a}} i_{\mathrm{a}} + \sum_{m \in \text{ lights}} (k_{\mathrm{d}} (\hat{L}_m \cdot \hat{N}) i_{m,\mathrm{d}} + k_{\mathrm{s}} (\hat{N} \cdot \hat{H})^{\alpha} i_{m,\mathrm{s}}) \tag{2.7}$$

where

- $\hat{L}_m$ is the direction vector from the surface point towards each of the m light sources
- $\hat{N}$ is the surface normal at the point
- $i_{m,\mathrm{d}}$ and $i_{m,\mathrm{s}}$ are diffuse and specular RGB color intensities of the light sources
- i_a is the intensity of the ambient light

$\hat{H}$ is defined as

$$\hat{H} = \frac{L + V}{\|L + V\|}, \tag{2.8}$$

where L is a direction vector from the surface point towards the light source and V — from the surface point towards the camera.

To generate objects reflections, used in jellyfish rendering, reflectivity constant k_r was added to the material parameters, and recursion limit r_c and importance cutoff y_c — to the ray parameters. Primary rays start with their recursion counter $r = 0$ and their importance $y = 1$. If the following three conditions are satisfied for any ray R_i

- $r < r_c$
- $y \geq y_c$
- $k_r > 0$ at the hit point

then a secondary ray R_{i+1} is cast with the following parameters

$$\begin{aligned} \hat{v}_{i+1} &= \hat{v}_i - 2\hat{N}_i(\hat{N}_i \cdot \hat{v}_i) \\ r_{i+1} &= r_i - 1 \\ y_{i+1} &= y_i Y' \end{aligned} \tag{2.9}$$

where $\hat{v}$ is the vector direction, $\hat{N}$ is the surface normal at the hit point and Y' is a luminance constant, defined as

$$Y' = Y_{ntsc} \cdot k_r, \tag{2.10}$$

where $Y_{ntsc} = \{0.299, 0.587, 0.114\}$ is a constant color vector defined by NTSC standard. Each subsequent ray makes its contribution to the resulting color, which is

$$I_r = k_r \times I_p, \tag{2.11}$$

where I_p is the illumination, calculated according to equation 2.7.

To simulate refraction of light occurring at the jellyfish boundaries, we used Schlick's approximation of Fresnel equations, which is

$$\begin{aligned} R(\theta) &= R_0 + (1 - R_0)(1 - \cos\theta)^5 \\ R_0 &= \left(\tfrac{n_1 - n_2}{n_1 + n_2}\right)^2 \end{aligned} \tag{2.12}$$

where θ is the angle between the direction of incoming light, n_1 and n_2 are refractive indices of the two materials at the boundary of which the refraction occurs. $R(\theta)$ determines what part of the incoming light intensity gets reflected off the boundary in the reflection direction, while the rest travels through the material with refractive index n_2, and its direction is determined by Snell's law:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1} \tag{2.13}$$

In our case, there were two translucent materials: the sea water and the mesoglea of jellyfish. The refraction index of sea water varies slightly with the water temperature, salinity and for different light wave lengths, but in our work we kept it constant and equal to 1.33. Jellyfish mesoglea may have varying refraction index, depending on the species, and in our work we used values in the 1.0–1.4 interval.

Similarly to the reflection, secondary refraction rays are cast recursively, their contribution to the final illumination value is accumulated. Travelling through materials, rays get attenuated according to Beer–Labmert law, which is

$$T = e^{-\mu l}, \tag{2.14}$$

where μ is the attenuation coefficient and l is the distance travelled by the ray through the material.

Using recursive ray tracing for purposes other than optical effects

The usage of ray tracing in computer graphics is not limited to visualization. It is used, for example, for collision detection: similarly to casting camera rays and checking their intersection with scene objects in order to visualize them, it is possible to cast rays off the surface of a moving scene object in order to check if the object is about to collide with another object. In this work, while using whitted–style ray tracing for visualization, we recursively cast additional "probe" rays to visually correct problems left unsolved by underlying physical simulation.

2.5 Spatial hierarchies

Space partitioning is widely used in this work. As rendering scenes tend to be quite large in terms of polygon count, naive approach to calculating ray–polygon intersection, which is testing every ray for intersection with every polygon, is too slow for real–time visualization. That is also true for collision detection and for particle interaction in fluid simulation in Chapter 3. We used HLBVH2 (see [22] for details) for recursive ray tracing, and *k-d tree* to quickly search for neighbouring particles in the MPS fluid simulation in 2D. In both cases, that changed the complexity from $O(n^2)$ to $O(n \log n)$.

2.6 GPU acceleration

Graphics processing units (GPU) typically have much higher number of processing cores than CPUs, although the cores themselves are less computationally powerful. This makes GPUs to be much faster than CPUs for some tasks, most noticeably, computer graphics–related, such as polygon rasterizing and image processing, where a single program (often called "shader" or "kernel") performing relatively simple calculation can

be executed simultaneously on a large array of data. While originally GPUs could only accept graphical data from CPUs, process it and pass forward to a display, in later GPUs two–way interaction became available, with data coming back from the GPU to the CPU. This made general purpose GPU programming (GPGPU) possible. While polygon rasterizing was supported even in earliest GPUs, GPU–accelerated ray tracing became possible only using later GPGPU features, and it does not benefit from GPU acceleration as much as rasterizing does. That is due to more divergent control flow in ray tracing: while with rasterizing a GPU operates on polygon vertices, performing exactly same operations on each one, with ray tracing it has to operate on rays, performing different sets of operations depending on whether the ray misses or hits an object, gets reflected/refracted, and so on. Still, 1–2 orders of magnitude speedup can be achieved, compared to the CPU ray tracing, depending on the complexity of the computational model.

2.7 Conclusion

In this chapter we outlined our technique for deformable body visualization, based on simplified physics, RBF space mapping and special case of recursive ray tracing. While we were using PhysX in our particular implementation, any physical simulation software can be used instead, its results taken, modified with RBFs and visualized with ray tracing. Perhaps the biggest drawback of our method is its lack of generality: while it has shown good result for practically important cases, such as marine animal simulation and multiple garment visualization, it will not work for every imaginable case of deformable body, or even for some more exotic sets of garment. We further discuss these limitations in chapters 3–5. Our heavy reliance on recursive ray tracing, which is not the fastest rendering method known to mankind, can be considered a drawback as well. However, complex physical interaction, typically used for the tasks we were trying to accomplish, are usually much slower, and as we replace a part of those physical interactions with our ray tracing, it results in significant saving in required computations. Finally, an advantage of our method is that, however it can be used together with an iterative physical simulation, it is itself not iterative, with all the computations done in a single pass, so there is no need to wait for an iteration to complete in order to start the next iteration.

Chapter 3

Artificial jellyfish: optimization of deformable shape

3.1 Introduction

Jellyfish are the earliest known animals to use muscle power for swimming [23]. They swim by contracting and expanding their mesogleal bells. The swimming muscles contract to expel a portion of water rearward out of the subumbrellar cavity, thus generating a thrust force to move the animal forward. The bell is refilled when it restores its shape after deformation it received during the thrust phase. The bell consists of a fiber-reinforced composite material called "mesoglea". The elastic characteristics of the mesogleal tissue were studied, for example, by Megill et al. [24]

The contractile muscle fibers of the medusae are only one cell layer thick, so the forces that they can produce do not scale favorably with the increasing medusa size. For a medusa with the bell of diameter D, the mass of water that needs to be expelled from within the bell scales as D^3, while the muscle force only scales as D^1. Therefore the force required for jet propulsion increases with the animal size more rapidly than the available physiological force [23]. Thus, the swimming performance may change dramatically with the increase of the medusan body size, and it is impossible to predict the optimal swimming parameters based

on the geometric and kinematic similarity.

The physics of jellyfish swimming is not well understood. Existing animation techniques use combinations of sinusoidal curves to specify the deformations. However, it is important for animation to achieve a realistic movement depending on the size and shape of a bell. We assume that "realistic" also means "optimal", as the movements of the real jellyfish were "optimized" by the process of natural evolution, and we, therefore, would be able to find realistic movements for an artificial 3D model of jellyfish by means of artificial evolution. Other applications, such as computational biology, soft robotics and development of new propulsion techniques can benefit from development of a generalized model of jellyfish swimming.

In this thesis we present a system for finding optimal swimming parameters for jellyfish models, based on our previous work where we studied vortex simulation for jellyfish [25]. The system consists of two main parts: simulated swimming and motion optimization. We introduce a simple technique based on radial basis functions (RBF) to model deformations of the jellyfish bell and a particle-gridless hybrid method for the analysis of incompressible flows. We modeled the interaction between the fluid particles and the surface of the bell in a form of elastic collision and reflection of the fluid particles off the boundary surface. The swimming efficiency was estimated for the bell and its particular movement specified by a set of control points. Genetic algorithms were used to find the optimal swimming pattern. To the best of our knowledge, this is the first work where the optimal swimming parameters for the jellyfish movement were found by solving the optimization problem. Throughout the paper we refer to two other paper concerning computational simulation of jellyfish ([26] and [27]), but neither of those employs any numerical optimization.

The remainder of this chapter is divided into 7 sections. In section 3.2 we discuss related work. We describe our approach in sections 3.3 to 3.5 and outline the algorithm in section 3.6. In section 3.7 we outline the specifics of our prototype implementation and report of the experimental results, and in section 3.8 we conclude the chapter and explain some limitations and possible directions of future work.

3.2 Related work

Studies of real life jellyfish

Experimental studies, including dye injection, filming and analyzing the resulting flow, indicate that smaller prolate medusae create strong jets during their bell contraction stage. Bigger oblate medusae, however, produce substantially less distinct jets and broad vortices at the bell margins. A hypothesis proposed by Colin and Costello [28] [23] [29] [30] is that oblate species are using their bell's margins as "paddles", thus utilizing a paddling, or rowing, mode of swimming. According to the model presented by Dabiri et al. [23], big oblate medusae are not capable of swimming via jet propulsion. There is, however, a study of McHenry and Jed [31] which suggests that the jetting model still provides more accurate approximation of swimming in oblate jellyfish.

The flow generated by oblate medusa's pulsatile jets consists mostly of radially symmetric rotating currents called vortex rings. To better understand the vortex formation and their effect on swimming performance, numerous experimental studies of real live jellyfish were performed [28] [23] [31] [32] [29] [30]. Researches using mechanical jet generators demonstrate that there is a physical limit – called the "vortex formation number" – for the maximum size of the vortex rings. Once this number is reached, no bigger vortex formation is possible, and the extra water creates a trailing current behind the vortex. The energy cost for generating this current is higher than that of creating the vortex ring, so it is optimal to generate the largest possible vortex without any trailing current [29]. Both thrust and efficiency increase in direct proportion with vortex ring volume [30]. Lipinski and Mohseni [26] used digitized motions of two real hydromedusae to computationally simulate the flows. Their results confirm the hypothesis proposed by Colin and Costello and demonstrate that distinct type of jellyfish ("jetting" and "paddling") produce substantially different kinds of vortices.

Fluid-solid interaction

Müller et al. proposed a particle-based method for interaction of fluids with deformable solids [33]. In their method they model the exchange of momentum between Lagrangian particle-based fluid model and solids represented by polygonal meshes with virtual boundary particles to model the solid-fluid interaction.

Lipinski and Mohseni [26] used digitized motions of two real hydromedusae to computationally simulate the flows. They used a new arbitrary Lagrangian-Eulerian method with mesh following the boundary between the fluid and the jellyfish body.

Yoon et al. presented a particle-gridless hybrid method for the analysis of incompressible flows [34]. Their numerical scheme included Lagrangian and Eulerian phases. The moving-particle semi-implicit method (MPS) was used for the Lagrangian phase, and a convection scheme based on a flow directional local grid was developed for the Eulerian phase.

Chentanez et al. presented a method for simulating the two-way interaction between fluids and deformable solids [35]. The fluids were simulated using an incompressible Eulerian formulation where a linear pressure projection on the fluid velocities enforces mass conservation, whereas elastic solids were simulated using a semi-implicit integrator implemented as a linear operator applied to the forces acting on the nodes in Lagrangian formulation.

Hirato et al. proposed a method for generating animations of jellyfish with tentacles [36]. They used a simplified computational model based on the MPS method to simulate the fluid. Their work is mainly focused on visually plausible modeling of tentacles.

Rudolf and Mould created a system for physically-based animation of jellyfish [27]. Their approach may look very similar to ours, as they also exploited the radial symmetry, simulating only a 2D cross-section, and then creating a 3D bell for the visualization. The main difference between the approach proposed in [27] and the one discussed in this paper is that Rudolf and Mould did not employ any optimization, instead assigning a visually plausible set of parameters manually, by trial and error. They used a spring-mass system to represent the body of a jellyfish and a grid-based immersed boundary method for fluid-solid coupling. As

they note in their work, there is still very little knowledge about physical properties of real jellyfish. Thus, we didn't feel necessary to employ something as complex as a spring-mass system, since the actual physical accuracy of the model would still be uncertain. Moreover, modeling a multi-layered structure of the jellyfish bell with only one layer of springs attached directly to the opposite sides of the bell does not look realistic. Some fugures from [27] demonstrate drastic change of both area and linear size of the umbrella cross-section during the contraction, something we failed to observe in real species, such as presented in experiments of Colin and Costello [30]. Instead of a spring-mass system, we use a simpler approach, with the umbrella of the jellyfish represented in 2D as two spline curves, deformed by RBFs. Instead of a grid-based method, for fluid simulation we used a particle-based method [34] with elastic collision and reflection of the fluid particles off the boundary surface to prevent fluid leaking across the boundary. Finally, [27] employs a very primitive visualization technique, an issue we were trying to address with a GPU-based parallel ray tracer, capable of representing transparency, reflectivity and venous structure.

Optimization

The problem was studied by many researchers from the computer graphics and animation community, but we have no room for the comprehensive referencing, so we will mention only a few we found most relevant to out work.

Sims was one of the pioneers of artificial evolution. In his work [37] he used genetic algorithms to create evolving images, textures, animations and plants, represented by procedural geometry, with human aesthetical selection instead of a fitness function. In [38] he used similar approach to artificially evolve both morphology and behavior of articulated (e. g. composed of rigid parts and connecting joints) creatures, which were evolved and trained to perform specific tasks, like walking, jumping, following a light source, competing for a ball with other creatures etc.

Terzopoulos et al. [39] modeled artificial fish as NURBS and spring-mass systems, using simulated annealing to find efficient moving patterns. Based on simulated sensory input, their fish could learn complex group behaviour, such as schooling, mating etc.

Tan et al. [40] used covariance matrix adaptation to find optimal swimming motion for fish, frog, turtle and even some fictional creatures, represented as articulated bodies; however, they stated that their simulation method is unsuitable for soft body creatures, such as jellyfish.

The works, discussed in this section, inspired our attempt to create a combined approach suitable for modeling and optimizing jellyfish. We emphasize that our work unites two themes of different research history: generation of time-dependent shapes and estimation of dynamical characteristics of the generated models.

3.3 Bell simulation

To simulate the bell contraction-expansion cycle we used a simplified approach based on non-linear deformations of a geometric object. Because the model of jellyfish has radial symmetry, we used a 2D model (cross-section) with the surface of the bell represented by two hemi-ellipsoidal curves – the upper and the lower. For our model we used a piece-wise linear approximation with the initial number of nodes equal 40. A space mapping technique based on RBFs (see [41], and references therein) was used for non-linear approximation of shape deformations in numerous applications. Space mapping in R^n defines a relationship between each pair of points in the original model and the model after geometric modification. Let an n-dimensional region $\Omega \subset R^n$ of an arbitrary configuration be given, and let Ω contain a set of arbitrary control points $\{q_i = (q_1^i, q_2^i, ..., q_n^i) : i = 1, 2, ..., N\}$for the non-deformed object, and $\{d_i = (d_1^i, d_2^i, ..., d_n^i) : i = 1, 2, ..., N\}$for the deformed object. By assumption, the points q_i and d_i are distinct and given on or near the surface of each of two objects. The goal of the construction of the deformed object is to find a smooth mapping function that approximately describes the spatial transformation. The inverse mapping function can be given in the form

$$q_i = f(d_i) + d_i, \tag{3.1}$$

where the components of the vector$f(d_i)$ are volume splines interpolating displacements of initial points q_i (see Section 2.3 for the details).

3.4 Fluid-solid coupling

Using a grid-based approach for jellyfish is possible, but poses a number of problems. Using a regular grid, as in [40] for an elastic body with varying thickness will result either in a huge computational overkill (if the grid is dense enough to accomodate the thin edges), or in a poor accuracy of the computation (if the grid is more sparse). Using an irregular grid, as in [26] requires solving a mesh warping/re-meshing problem. Solving Navier-Stokes equations with moving boundary is a hard problem. For simplicity, we chose a particle-based method. Particle-based methods became a de-facto standard for a class of problems where high precision is not required. For modeling we used almost the same scheme as proposed by Yoon, Koshizuka and Oka [34]. They proposed a particle-gridless hybrid method for the analysis of incompressible flows, where tracing of virtual moving particles is used instead of solving nonlinear equations of velocity field. A particle interacts with other particles according to a weight function $w(r)$, where r is the distance between two particles. The weight function used by Koshizuka et al. is

$$w(r) = \begin{cases} -(2r/r_e)^2 + 2 & (0 \le r < 0.5r_e) \\ (2r/r_e - 2)^2 & (0.5r_e \le r < r_e) \\ 0 & (r_e \le r) \end{cases} \tag{3.2}$$

Density for a particle is calculated as the sum of weights of its interactions with the other particles (all interaction happens only within the radius r_e):

$$\langle n \rangle_i = \sum_i w(|r_j - r_i|). \tag{3.3}$$

Note, that, unlike in the MPS method, the particle number density here is not required to be constant. A gradient vector between two particles i and j possessing scalar quantities ϕ_i and ϕ_j at coordinates r_i and r_j is equal to $(\phi_j - \phi_i)(r_j - r_i)/|r_j - r_i|^2$. The gradient vector at the particle i is given as the weighted average of these gradient vectors:

$$\langle \nabla \phi \rangle_i = \frac{d}{n^0} \sum_{j \neq i} \left[\frac{\phi_j - \phi_i}{|r_j - r_i|^2} (r_j - r_i) w(|r_j - r_i|) \right], \tag{3.4}$$

where d is the number of space dimensions and n^0 is the particle number density.

Diffusion is modeled by distribution of a quantity from a particle to its neighbors using the weight function:

$$\langle \nabla^2 \phi \rangle_i = \frac{2d}{\lambda n^0} \sum_{j \neq i} [(\phi_j - \phi_i) w(r_j - r_i)], \tag{3.5}$$

where λ for a two-dimensional case with Equation (3.2) as the weight function is equal to $\frac{31}{140} r_e^2$. This model is conservative, because the quantity lost by the particle i is obtained by the particle j.

The continuity equation for incompressible fluid can be written as follows:

$$\frac{D\rho}{Dt} = -\rho(\nabla \cdot u) = 0. \tag{3.6}$$

The velocity divergence at the particle i is given by:

$$\langle \nabla \cdot u \rangle = \frac{d}{n^0} \sum_{j \neq i} \frac{(u_j - u_i) \cdot (r_j - r_i)}{|r_j - r_i|^2} w(|r_j - r_i|). \tag{3.7}$$

Then the pressure is calculated as:

$$\frac{u_i^{**} - u_i^*}{\Delta t} = -\frac{1}{\rho} \langle \nabla P^{n+1} \rangle_i, \tag{3.8}$$

$$\langle \nabla^2 P^{n+1} \rangle_i = \frac{\rho}{\Delta t} \langle \nabla \cdot u^* \rangle_i, \tag{3.9}$$

where u^* is the temporal velocity obtained from the explicit calculation and u_i^{**} is the new-time velocity. The left side of (3.9) is calculated using the Laplacian model (3.5). The right side is the velocity divergence, calculated by (3.7). We use variable r_e to avoid cases where some particles near the boundary will have very few neighbours to interact with. It gives a system of linear equations represented by an unsymmetric matrix, which is solved by an unsymmetric-pattern multifrontal method [42]. Solving (3.9) may seem computationally expensive, but with jellyfish, the most important fluid-solid interaction often happens near the very thin edges of the bell, so calculating accurate pressure field is necessary.

Instead of using a higher-order gridless convection scheme as it was proposed by Yoon et al. [34] to approximate flow directions, we applied averaging of the velocities field by a simple scheme, based on Shepard's method (partition of unity) [43].

Boundary conditions are perhaps the most important factor influencing the accuracy of the flow computation. The manner in which the boundary conditions are imposed influences the convergent properties of the solution. Usually in particle-based methods boundary particles are used to approximate the no-penetration condition [33] [44]. Repulsion and adhesion forces between the particles are used to simulate the no-penetration, no-slip and actio = reactio conditions on the boundary of the solid.

In our work contour points represent the geometry of the model and also define fluid boundaries. That is, the solution points are defined by the fluid particles and the particles located on the boundary of the bell. For each boundary particle we can calculate the boundary normal vector, pointing outwards, into the flow domain. For the no-slip condition, only the normal speed component of any boundary particle is used, while the tangential speed component is discarded. The no-penetration condition is modeled in a form of elastic collision and reflection of the fluid particles off the boundary surface. The motion of the bell was computed using only translational parts in y direction. One component of the force F on a rigid body is a derivative of linear momentum mv of the gravitational center. It is assumed that jellyfish body density is equal to the density of the water. Thus m is a volume occupied by the jellyfish.

The force F also invokes fluid and rigid body interaction. Points on the curve used to represent the bell can be considered as rigid particles. When the bell is deformed, distances between boundary particles may change, so we put a new set of boundary particles after each deformation, by evenly subdividing the curves. The strategy of using rigid particles we followed was first proposed in [45]. The forces on rigid particles are computed by assuming the rigid body as a fluid. Therefore, for a particle i with the pressure p_i, mass density ρ_i and speed v_i, the force from the fluid acting on the node particle $f_i^{fluid} = f_i^{press} + f_i^{vis}$ is calculated by using the physical values of the neighbor particles as follows [46]:

$$f_i^{press} = -\sum_{i \neq j} (p_i + p_j)/2\bar{\rho_j} \nabla_i w_h^{ij} \tag{3.10}$$

$$f_i^{vis} = -\sum_{i \neq j} \Pi_{ij} \nabla_i w_h^{ij} \tag{3.11}$$

where

$$\Pi_{ij} = \begin{cases} -(c\mu_{ij} + 2\mu_{ij}^2)/\bar{\rho_j} & \mu_{ij} < 0 \\ 0 & \mu_{ij} \geq 0 \end{cases} \tag{3.12}$$

$\mu_{ij} = e_r v_{ij} r_{ij}/(r_{ij}^2 + 0.01 e_r^2)$, $\bar{\rho_j} = 0.5(\rho_i + \rho_j)$, $r_{ij} = r_i - r_j$, $v_{ij} = v_i - v_j$

3.5 Optimization

In techniques based on the error functional minimization it may become necessary to solve highly non-linear problems. Minimization by standard techniques requires high computational effort. Minimization of a simplified functional, for example a quadratic one, is reduced to solving a simple system of linear equations. However, it leads to iterative minimization that depends on a sufficiently good initial guess. It seems to us that an attractive way of attacking this problem is to use optimization techniques based on genetic algorithms, proposed by Mahfoud and Goldberg in [47]. In this work we used an algorithm with simulated annealing type selection.

The application of the genetic algorithm starts with initially selecting a set of N variable control points $\{d_i = (d_{1i}, d_{2i}, ...) : i = 1, 2, ..., N\}$ for the definition of the space transformation generating the deformed object. Actually, the user defines points q_i on the initial image of the bell in its rest state with corresponding points d_i on the model of fully contracted bell (Fig. 3.1). The collection of coordinates d_i and the contraction time t_{cont} define a creature. The algorithm begins by randomly distorting the initial creature and generates s creatures, which form the initial population. Now, the genetic algorithm with sequential simulated annealing is applied to this initial population to minimize the fitness function, same as used by Savchenko and Schmitt [41].

The spline $f(P)$, determined by the set of N variable control points d_i which constitute a creature, used for global space mapping, provides a minimization of quantity $h^t A^{-1} h$, that is called "bending energy" E^b. 4

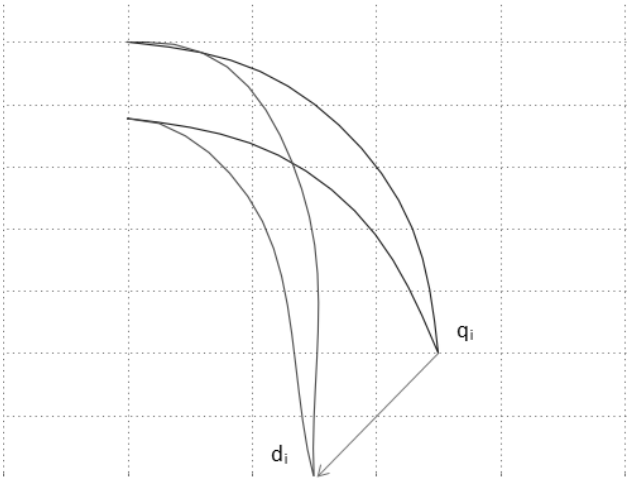

Figure 3.1: An example of a creature, with a deformation vector assigned at the bell margin. q_i designate the initial position of the bell margin point (rest shape), d_i – the fully deformed position of the same bell margin point.

points belonging to the border of the bounding box and two additional points in the center of the bell ($x = 0$) on the upper and lower curve are used as anchor points. k destination points define general deformation of the jellyfish bell. A^{-1} is the bending energy matrix, which is the inverse $N \times N$ upper left submatrix of T, and h_i are so-called heights and $N = 10 + k$. Space transformation h_i is the difference between the coordinates of the initial and destination point placements as shown in Fig. 3.1. The bending energy of a general transformation is the sum $x^t A^{-1} x + y^t A^{-1} y$ of the bending energy of its horizontal x-components, modeled as a "horizontal" plate, and the bending energy of its vertical y-component, modeled similarly as a "vertical" plate.

In our simulation we used the "economy" principle: the jellyfish is striving to reach maximum speed with minimum deformation of the bell. Thus, one of the fitness function component is the bending energy E^b. In the numerical analysis we also measured two quantities characterizing jellyfish locomotion, i. e. distance D passed by a body which is defined by swimming speed ν and energy loss E^l. Energy loss is assumed to be equal to surrounding water energy, and is calculated as

$$E^l = \sum_k \sum_i v_{ik}^2, \tag{3.13}$$

where v_{ik} is the speed of i-th water particle at the k-th substep of the simulation. To be used in the fitness function, E^l is normalized by dividing it by the global maximum value of E^l for all the creatures simulated. Following the "economy" principle, we define the fitness function as follows:

$$Fitness = w_d \cdot D - w_b \cdot E^b - w_e \cdot E^l \tag{3.14}$$

where w_b is the weight for RBF energy, w_e is the weight for kinetic energy of a particle, w_d is the weight for the object velocity. These parameters are set by the user to choose a mode of movement.

Although movement optimization was the goal, comparing and evaluating different optimization methods was not the focus of this particular work. As "optimization" of real creatures involves evolutionary genetic changes, evolutionary methods such as genetic algorithms look like a natural choice, as reflected in works of Sims ([37], [38]), Terzopoulos ([39]) and others. Following their example, we used genetic algorithms with simulated annealing for our optimization. As we did not develop any new genetic algorithm (in fact, we used a 3rd party software library for it — see below in the "Implementation" section), we are going to only briefly outline it here (for more technical details and code examples, see [48]).

In this work, creatures (or "chromosomes") are represented as sets of x and y coordinates of the deformation vectors and a contraction time t_{cont}. They are subject to mutation, crossover and selection. We used binary chromosome representation, where the three variables are packed into a bit string. Representing a chromosome not as a bit string, but as an array of real numbers is also possible. Mutation, with a certain mutation probability, changes a random bit in the chromosome in case of a bit string representation, and replaces a parameter with a new randomly generated one in case of a real number representation. Crossover creates a new chromosome from the bits of two "parent" chromosomes in case of a bit representation, or by picking a random value from between the parent's values for the same parameter, in case of a real number representation — in either case, a completely new value for the parameter can be created, which is not present in any of the parents. The creatures in the current population compete for survival, and with the simulated annealing a creature survives the competition with another creature based not solely on their fitness function values, but on a probability function P based on both their fitness functions and a global

"temperature" parameter T, which decreases over time. With such selection scheme, even a worse creature can be selected, but the probability of such selection decreases along with T, e. g. $\lim_{T\to 0} P = 0$. Such selection type prevents the system from being trapped at a local optimum and, thus, failing to find the global optimum. For the temperature, we used the following:

$$T = \frac{1}{G_{offset} + \log(1 + G_{counter})}, \tag{3.15}$$

where $G_{counter}$ is the generation number and G_{offset} is a constant used to avoid large acceptance probabilities for bad elements.

3.6 Algorithm

Initially, the 2D contour of the bell cross-section is specified as an array of points. Deformations are assigned to the bell margin points (see Fig. 3.1). Particles are placed at a regular interval (on a regular grid) inside the bounding box, except the inner area of the bell. Then, the following steps are performed iteratively for each step of the contraction/expansion cycle:

1. The averaged density is calculated for every particle. A ball is generated for every particle, and the density is defined as the volume of the ball divided by the number of particles inside the ball. The ball diameter is not constant, and is adjusted so that all balls contain roughly similar number of particles.

2. The bell margin points are moved by a step along the deformation vectors. For the rest of the points their displacement vectors are calculated using RBFs.

3. A cardinal spline is fit through the displaced boundary points. Because some segments may become too long, we discard the boundary points and insert them again by subdividing the spline curve evenly, so that all the distances between neighboring points are mostly equal.

4. A Poisson equation in a matrix form (unsymmetric, about 10000 linear equations) is solved, giving new values of pressure for each particle.

5. Gradient vectors are calculated applying equation (3.4). For every particle, the speed vector is calculated, and the particle is then moved along the vector by the time step Δt.

6. New pressure values for the displaced particles are interpolated back to the nodes of the regular grid.

7. Distance passed and energy lost at this step are calculated for the creature.

At the end of a swimming cycle, we have a fitness for the creature according to (3.14). A population of such creatures is evolved until convergence within 10%. The best creature is then selected as "optimal". Because the simulation is quite computationally expensive, we tried to keep the population size at the minimum, as long as it did not increase the convergence speed. Studies of the optimal population size are contradictory: for example, in [49] population sizes from 5 to 200 chromosomes were tested, and the the general conclusion was that increasing the population size causes the convergence time to decrease (e. g. faster convergence), whereas in [50] the increase in population size caused the convergence time to increase (e. g. slower convergence). As it seems to depend not only on the number of optimizing parameters, but on the nature of the problem as well, we found the optimal population size experimentally. In case of a jellyfish we can also know approximate range of the motion parameters (e. g. how much of contraction/expansion is definitely "too much"), and we can use that knowledge to further limit the search space of the genetic algorithms, improving the convergence time. Experimentally we found the number of 10 creatures to be optimal for our problem.

For the animation, we created a 3D model out of the 2D contour, and then visualized by ray tracing. At first, the mesh was created by rotational extrusion and tessellation of the original 2D contour. Finally, we used Blender 3D modeling suite to create a roughly similar 3D model with some embellishments, such as inner "veins", inward-oriented "velum" and several tentacles. Rotation of the original 2D contour was still used to put anchor points at some interval around the bell. The anchor points were used as a "skeleton" for RBF-based deformations of the entire bell model in 3D, including the veins, at each animation step. Some random perturbations were added to the anchor points to make our jellyfish look less artificial (Fig. 3.2).

Number of sub-steps	Calculation time (sec)	Path (m)
6	7	0.02
10	13.14	0.032
14	18	0.031
18	23.99	0.033
20	26.6	0.038

Table 3.1: Performance results of Aurelia aurita jellyfish simulation, showing how the number of simulation substeps (temporal resolution) affects the calculation time and the distance, travelled by the jellyfish.

The tentacles were modeled as soft cloth with one side attached to the bell and deformed separately. The cloth structure was represented by a triangular mesh, with nodes affected by the simulated water flow.

3.7 Implementation details

We implemented this method and used it to find the optimal swimming parameters for a simple oblate jellyfish similar to Aurelia aurita [30]. The program was written in C++ and CUDA C. UMFPACK library [42] was used for solving large sparse systems of linear equations, and GAlib library — for genetic optimization. The algorithm terminates when a satisfactory fitness level has been reached for the population. In practice it happens when the number of generations is approximately 30. Table 3.1 shows the dependence between the number of simulation substeps and the resulting path travelled by the jellyfish. As it is common in iterative methods, more substeps mean more precision, and we stop increasing the number of substeps when the corresponding increase of precision becomes small (the path differs only at the 3rd decimal digit, as shown in Table 3.1). In our implementation we typically used 14 substeps with the calculation time of about 18 seconds to simulate the full contraction–expansion cycle of a single creature. That amounted to 10 minutes on a single core of an Intel Core 2 Quad computer for the entire calculation, e. g. genetic optimization of the entire population of 10 creatures. The size of the simulation area is 20x20 cm, with the resoultion of 100x100 particles. The bell diameter is 10 cm.

In the particular case (see Fig. 3.2) weights (lazy mode) $w_b = 0.3$, $w_e = 0.3$ and $w_d = 0.4$ were used. The vortices produced by the simulation (Fig. 3.3) were similar to those observed for real jellyfish, showing that

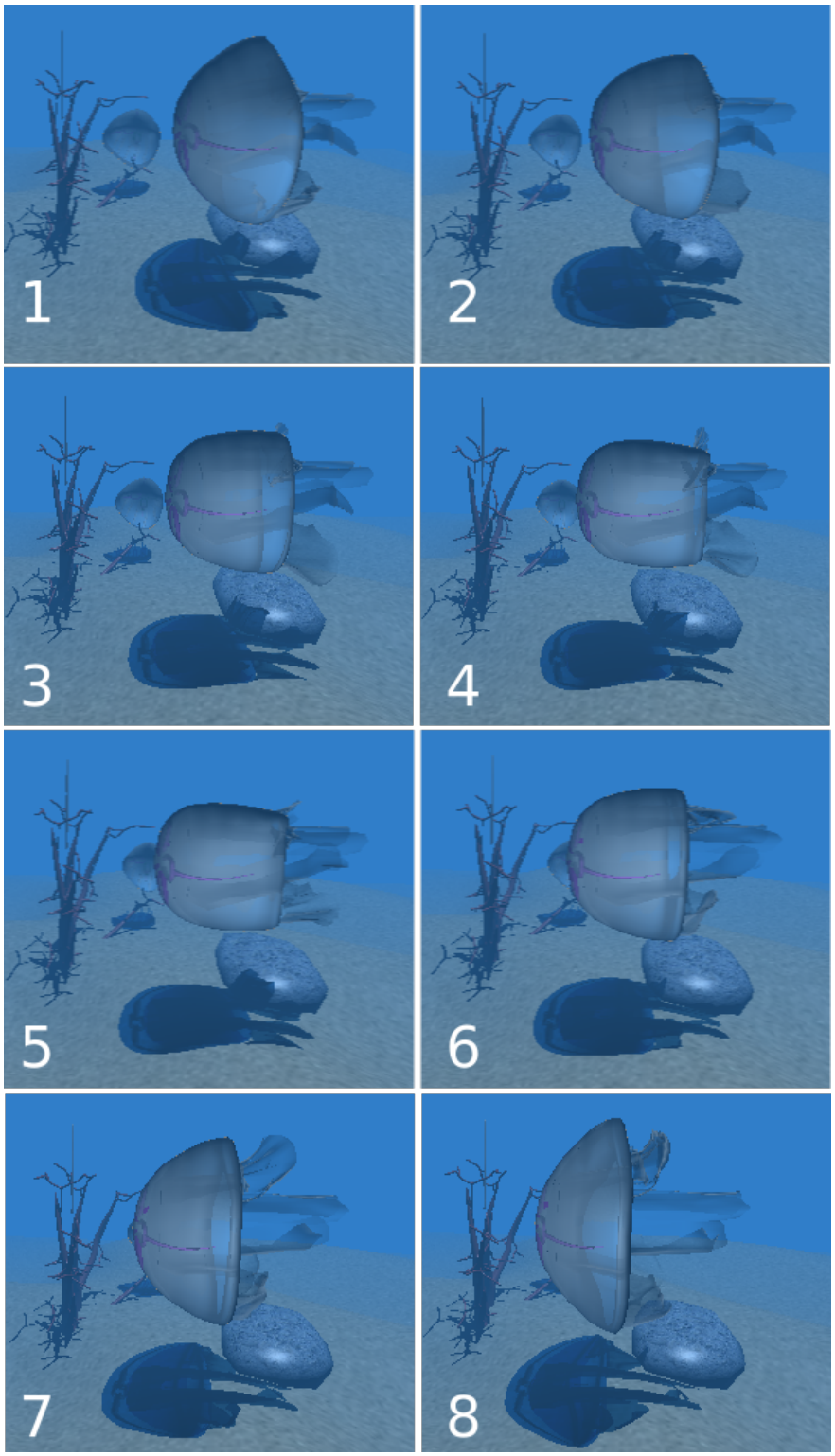

Figure 3.2: Animation sequence of a single contraction–expansion cycle of a jellyfish. The contraction phase is shown with frames 1–4, the expansion phase — with frames 5–8. Floating tentacles are animated according to the water velocity field. Small additional deformations are applied to the bell for a more realistic look.

application of the no-slip condition looks reasonable.

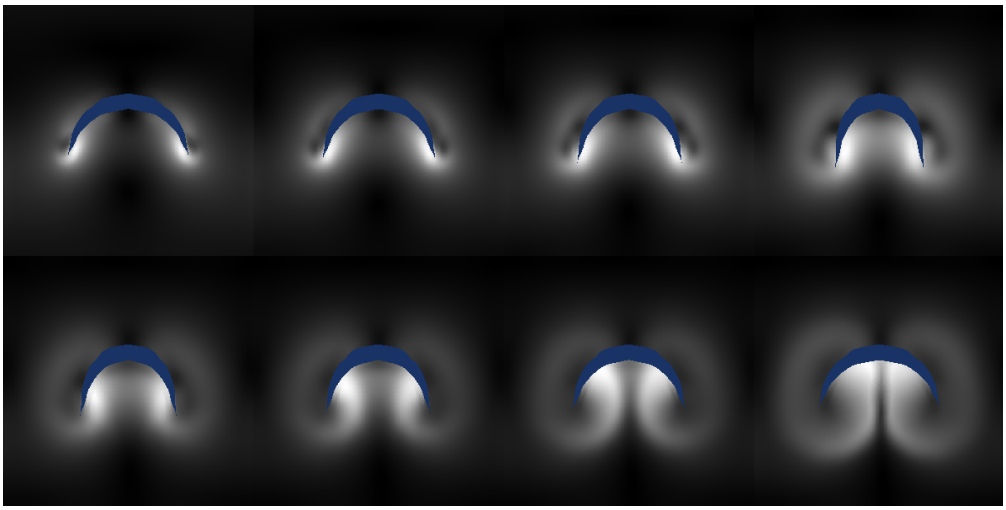

Figure 3.3: Formation of a vortex in 2D during one full expansion–contraction step (the expansion is shown in the top row, the contraction – in the bottom row, left to right).

A GPU-based parallel ray tracing system was developed for the visualization. We used recursive ray tracing to visualize the bell as a transparent object with refraction and reflection. Each primary ray hitting the bell was spawning several secondary rays, so the animation performance varied depending on the number of primary rays hitting the bell, and, thus, on the distance from the camera to the jellyfish and on the viewing angle. Our ray tracer produced 20-30 frames per second on a GeForce GT 540M GPU. We must add, that experience in areas such as 3D art and texture painting would add significantly to the observed realism of the animation, but that is beyond the scope of this work.

To validate our results, we compared them with experimental data received by Colin and Costello [28] [30]. We could not expect our simulation data to match their experimental data exactly because of the simplification we had to make in our jellyfish model (see Section 3.8 for the limitations), as well as fundamental limitations of computational fluid simulation. Still, we found our results to be in agreement with the results of real jellyfish studies in several key points. Vortices, generated by the jellyfish model in our simulation (see Fig. 3.3) were similar to those produced by real jellyfish of similar shape and filmed as the jellyfish emerged from a ball of dyed sea water. Experimental studies in [28] and [30] show big differences in kinematic profile

of prolate and oblate jellyfish of different size: smaller prolate jellyfish contract more rapidly and can travel a distance greater than its body size during a single contraction–expansion cycle, while bigger oblate jellyfish contract slower and travel less distance (relative to their body size). A similar slower contraction pattern was found by genetic algorithm in our optimization. Speed, acceleration and path seem to be affected by factors other than the size and shape of the bell and the contraction speed (no interpretation is given in [28] and [30]). In our simulation they can be affected by assigning different weight parameters in the fitness function equation 3.14. While those parameters of our jellyfish model did not match any of the real species exactly (due to the simplifications of the computational models), they fit well in between those parameters for real jellyfish of similar geometry.

3.8 Conclusion

We employed our simulation software to find the optimal movement for a very simple 2D model, swimming straight ahead. A more thorough validation of our technique, using a variety of sizes and shapes, as well as robustness and sensitivity studies are required and are subjects of future work. A few control points were used to specify bell contraction, and the movements of the bell margins were set by only two axially-symmetric vectors. Real jellyfish have no brain or eyes (although some of them have photosensitive spots on their bells) and do not deliberately choose any complex swimming trajectory, so we think our simplification has no big impact on the results veracity for jellyfish. However, following complex paths would be crucial for jellyfish-like robots. To simulate such behavior, it may be necessary to perform the simulation in 3D and with larger number of (possibly asymmetric) control points.

We did not, either, take into account jellyfish feeding behavior and tentacles. Real jellyfish have an oral opening inside the bell. Some of them also have numerous tentacles spread in the water. The tentacles add drag force and decrease swimming performance, but are used to catch prey. Jellyfish create water flows to carry their prey through the tentacles or into the oral cavity itself. Modeling such behavior is important for computational biology. Additional parameters for it can be incorporated into the fitness function.

Another possible future work would be optimization of the shape itself, e. g. finding both optimal movement and optimal shape, satisfying constraints imposed by a 3D designer.

Chapter 4

Virtual mannequin: real–time rendering of multi–layered clothing

4.1 Introduction

Virtual garment simulation has been developed considerably along with ever increasing computational power of modern day computer hardware — yet, despite the ever increasing computational power of modern day computer hardware, it remains a very challenging subject. The reasons for that lie within the very nature of cloth: thin and elastic, it requires both a lot of polygons to convincingly portray the elastic behavior, and a high precision simulation to convincingly model any interaction, such as collision. Time and space constraints (polygon budget), as well as quality requirements vary greatly, from hours per frame on a render farm to produce images for a movie, to fractions of second per frame on a commodity laptop (or even a smartphone) for interactive applications like garment design tools or virtual fitting rooms.

In this chapter we explore a problem of rendering multiple garments put together on a human figure (Fig. 4.1). Creation of a 3D garment model typically involves manual design (in a form of sketching or cutting and attaching 2D patterns together) to define the overall shape, and physical simulation to create realistic looking drapes and wrinkles. As we can easily imagine, while designers are able to create garments that

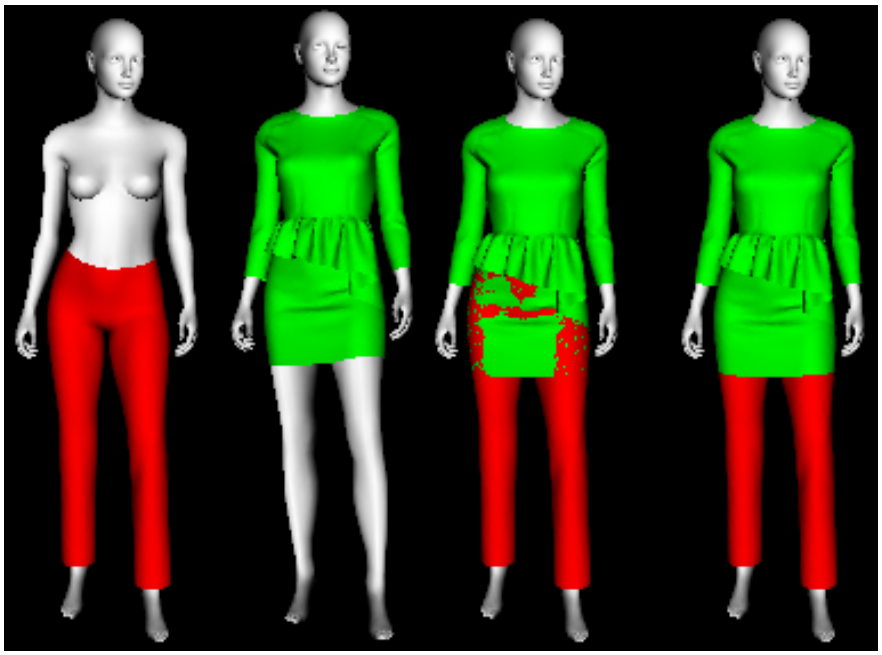

Figure 4.1: While garment models can be pre–made to fit a given human figure one by one, they cannot be pre–made to also fit each other in all possible combinations. Left to right: pants fitting correctly; dress fitting correctly; interference between dress and pants; desired result.

fit a given human body model individually, it would be totally impractical (if at all possible) to design the entire set of garments in such a way that they fit nicely on top of each other in all possible combinations. Cloth interpenetrations may occur, and they have to be resolved. Collision detection comes to help, but with clothing it poses additional problems. High precision is required on big data sets, and while modern GPUs can render tens of thousands of polygons at real–time rates, collision detection remains relatively slow. It is easier to prevent cloth surfaces to penetrate body volume, where a clear distinction between the inside and the outside of the volume can be made — that is not the case with several cloth surfaces. Simplified meshes are often used for physical simulations, along with finer meshes used for visualization — an approach often used in games, it lacks precision required for multi–layer cloth simulation. The approach presented in this chapter let us avoid the high precision requirement altogether, allowing the usage of a very basic physical simulation, or, in some cases, allowing not to use any physical simulation at all.

Main contribution

Presented in this chapter is a GPU–accelerated ray tracing–based approach to fast and visually correct rendering of human figures dressed in multiple garments, the main problem solved being cloth–cloth interference. Most other works related to that problem use iterative collision detection/response schemes to achieve high physical accuracy. Studying them, we often saw one problem: while the physical configuration is already mostly correct, with only very little left to fix, those little physical inaccuracies create rather big visualization problems. A typical solution proposed there is to achieve even higher physical accuracy. That is done by either improving computational performance to be able to use more iterations, or by improving computational accuracy to achieve a good result with less iterations. In this work we assume that physical correctness only matters to a degree: instead of improving it further we can, at some stage, use faster and simpler geometrical retouching technique to correct the remaining visualization problems directly. Our approach is tailored specifically to dressed human figures, more general cases of cloth simulation are not considered. Clothes are represented with polygonal surface meshes (unstructured grids). Mesh quality and topology is not necessarily preserved — completely invisible parts of the meshes may even be removed from the data sets to reduce further calculation costs, as long as the resulting visualization looks convincing. Physical properties of the clothes are not taken into account — we believe that our approach can complement a physically–based method, significantly reducing the required simulation precision and, thus, the computational cost. Operating mostly in image space, our method can work with garments created completely manually or with physical simulations of arbitrary quality, and it is not so sensitive to the mesh resolution.

Organization

The rest of the chapter is organized as follows. In section 4.2 we briefly survey other works related to cloth interference (for an extensive survey of such works we refer the reader to the work of Magnenat–Thalmann and Volino [51]). Section 4.3 gives a general description of our approach, and section 4.4 describes in details a particular implementation we use to receive the results discussed in section 4.5. We conclude with section

4.6, where we discuss some shortcomings and limitation of this approach, as well as possible directions of further research.

4.2 Related work

There are a number of methods for creation of 3D garment models, and for a more extensive survey we refer to the work of Liu et al.[52]. There are several works presenting more or less complete frameworks for creation and simulation of dressed human bodies, which also provide good introductions into many associated problems, such as [53], [54] and [55]. Some works are only concerned with overall appearance and may use simplified physics, others may attempt to precisely estimate how well it fits by measuring tensile strain–stress, such as the work of Volino et al.[56]. If particularly realistic cloth parameters are required, there are standard procedures and equipment to measure them, for example, Kawabata evaluation system[57].

Garment creation typically involves a combination of 3D scanning, manual design and physical simulation. The dominant method is 2D pattern construction, attaching them together and draping the result — that is where physical simulations are used. Umetani et al. [58] presented a system for interactive 2D/3D garment design. They used StVK to simulate 3D cloth drapes in response to the user modifying a 2D cloth pattern. Sketch–based cloth modeling has also received some attention. Yasseen et al. [59] presented a garment design system, where they constructed quad meshes from sketched contours of the desired garments. The quad meshes are then converted into triangular meshes for mechanical simulation of elasticity, gravity and collision forces.

Cordier et al.[54] developed a methodology for virtual garment fitting on different body sizes, where 3D models of humans were produced from photographs or body measurements, with a pre–calculated generic database used to produce personally sized bodies for a web application.

While many works only consider cloth–body collisions, cloth–cloth interaction is more challenging (and probably more practical: we do not often see people dressed in just a single piece of garment). Volino and Magnenat–Thalmann [18] presented a collision response scheme based on finding intersection contours of

cloth surfaces (a physically incorrect state, resulting from approximate collision detection) and displacing the meshes in a way that decreases the contours length, ultimately leading to their disappearance. Volino et al. [60] presented techniques for simulating motion of deformable surfaces, where a statistical evaluation of all the collisions in a region is used to enforce collision consistency.

As z–buffer is commonly used in GPU rendering hardware to render overlapping polygons correctly, an idea of using it to fix cloth interference seems natural. To our knowledge, the first work related to GPU–accelerated interference detection was that of Shinya and Forgue [61]. Lacking modern days GPGPU features in the year 1991, they proposed an algorithm where all polygons were first rasterized onto a screen surface, and for each screen coordinates all z–buffer values were collected and then used to determine if interference occurred along the z–direction. Based on z–values counting, their approach only worked for closed surfaces. However computationally fast, it had very limited precision and was intended mostly as an efficient culling method for more precise interference detection algorithms. Knott and Pai [62] presented an algorithm to detect collisions between solid object, based on ray casting, which is used implicitly through OpenGL frame buffer operations.

Vassilev et al. [63] used hardware–accelerated OpenGL off–screen rendering to generate two depth maps of a human body (front and back), which were then used to detect collisions with clothes, modeled as mass–spring systems. Their approach only worked for cloth–body interaction, but not for cloth–cloth interaction. In addition, it did not work for more complex poses, where visual occlusions do not allow to create unambiguous depth maps. Vassilev and Spanlang extended that work, developing a system [64] that works with multiple layers of clothing but not with more complex poses. Rodriguez-Navarro et al. [65] extended [63] to alleviate the occlusion problem by generating separate depth maps for each body segment, but not the cloth–cloth interaction problem. Still, it is unclear to us how it will work for big folds and creases.

Ray tracing is a more general and less restrictive method of rendering than polygon rasterization used in OpenGL, and it was used for decades as a collision detection method as well, albeit for simple cases involving small number of rays. Until recent development of general purpose GPU programming (GPGPU) it was too slow for real–time visualization, but with technologies like CUDA and OpenCL it became possible to

use it for real–time applications. Hermann et al. [66] presented a raytracing–based approach to collision detection between deformable bodies. Rays are traced not from the camera along the viewing direction, but from the vertices along their normal vectors. Vassilev [67] used GPU–accelerated ray tracing to detect collisions of cloth, represented as a mass–spring system, with other clothes as well as rigid objects. The conclusion drawn there was that their approach is fast enough to simulate one layer of cloth in a scene with static rigid objects, but still too slow for something as complex as animated dressed characters. Their work is, in some parts, very similar to our own: ray tracing is used for interference detection; rays are traced from the vertices in the directions opposite to surface normals as well as some additional vectors ("velocity vectors" in [67], "global normals" in our work); even the choice of ray tracing framework (NVidia OptiX). However, in [67] an iterative collision detection/response simulation is used for physically–based models of cloth, which are deformed in object space; our approach is not iterative, does not use collision detection and physically–based models, and most of the data augmentation is performed in image space. The problem we solve is also different — we do not dynamically simulate (e. g. fit and drape) pieces of cloth, but rather augment pre–made (e. g. already simulated) garments.

There are also data–driven techniques that do not internally use any explicit physical simulation — instead, they first train on data received from such a simulation, treated as a "black box". After learning a simpler model of garment behavior from pre–recorded human motion sequences, they can reproduce it much faster and without any physical simulation involved. De Aguiar et al. [68] presented a data–driven technique which uses no a priori physical model and can still approximately resolve cloth–body collisions. However, it cannot handle motions significantly different from those used for the training, and sometimes the precision is not enough even for cloth–body interaction. In what looks like a further development of this approach, Guan et al. [69] presented a system primarily aimed at real–time hair simulation, which can also handle cloth-body interaction. They, however, had to incorporate a more conventional collision detection/response technique into their work to achieve satisfactory results. Kavan et al. [70] proposed a method for learning linear upsampling operator allowing to enrich coarse meshes used for physical simulation to achieve better context–specific detalization than simple subdivision or interpolation schemes allow.

While they are not so much related to this work, for the sake of completeness we also want to mention 2D–based techniques, such as the work of Yamada et al.[71], where 2D photographic images are morphed to fit underlying 2D human images. With all the restrictions imposed by the 2D approach, such as pose and lighting limitations, under those restrictions they can easily provide photorealistic results, quite costly to achieve with today's 3D rendering capabilities. From a consumer perspective, they do not require expensive and specialized hardware, such as 3D scanners to create body models, because a simple photo camera is used instead (although a 3D approach may benefit from it as well, like [54]). They also do not require nearly as much data to be stored or transferred over computer networks, making them a good fit for internet–based services.

4.3 Description of the method

Problem formulation

Even though physical simulation of deformable bodies in general is already a challenging field of computer science, we believe that several inherent properties of cloth simulation make it even more difficult. Compared to most other deformable bodies, cloth is even more deformable, therefore simulating it creates additional problems specific to cloth. If modeled with conventional polygonal meshes, it requires a very large amount of very small polygons (in other words, very high polygon resolution) to portray the elasticity, because each separate polygon by itself is not deformable. Consequently, it leads to high computational costs during the simulation, because the amount of data is big. The high deformability and low bending stiffness also leads to largely cloth–specific problems such as self–intersection, which does not commonly occur, for example, in deformable solids. The thin nature of cloth also requires high computational precision during the simulation to avoid incorrect collision detection: there is, generally, no inside–outside space relationship, and a very small computational error may lead to very big simulation and visualization problems. It also tends to create numerical instabilities and oscillatory solutions. While some techniques to alleviate these problems of physically–based simulation methods do exist, we attempted to develop a non physically–based method.

Our goal is a method, fast enough to be practical, of visually correct rendering of human figures dressed in multiple clothing. We assume that each clothing is correctly modeled as a polygonal surface mesh (unstructured grid), fitting the human figure, also represented with a polygonal surface mesh. Each garment model represents a separate layer of cloth (even though a single garment model may consist of multiple surface layers, we treat it as one layer), and the layers order can be unambiguously defined, starting from the human model. In real life that is not always the case, different parts of a garment set often interlock, like a shoelace in a shoe or a handkerchief sticking out of a pocket — still, in computer graphics they are usually modeled together as a single surface mesh.

Each garment model fits the underlying human figure, but if several of them are put on together, they do not necessarily fit each other. Typically, mesh intersections occur, causing incorrect rendering (see, for example, Fig. 4.1). Even if there are no intersections, when all the meshes are visualized together a condition called "z–fighting" may still occur. It gets worse in areas where drapes were designed or simulated: as those were probably simulated separately, only with respect to the human body model, the drapes and wrinkles of a clothing tend to intersect those of other clothing. Such condition does not even represent a physically correct state, therefore, physically based cloth simulation methods cannot be directly applied to this problem. Under the assumption that all the garments already fit the same body model correctly, we can also assume that in the interfering areas the distances between the layers are relatively small (for example, if there are two polygons incorrectly positioned one over the other on a sleeve, the distance between them is sufficiently smaller than the diameter of the sleeve itself). Let us also note that for the clothes to be incorrectly positioned one over another, they do not necessarily have to intersect (although they usually do). For example, in a context of an application such as virtual fitting room, two T–shirts may be originally designed so that shirt 1 is completely hidden by shirt 2, when both are put on together, and no interference occurs. Still, a user may wish to put shirt 1 over, not under shirt 2.

Our approach is not based on finding intersecting regions or collision detection. Instead we use a ray tracing based approach to estimate relative positions of the layers and local distances between the layers at the same time. The entire mesh is processed all at once in a manner that is highly parallelizable. We use

this approach in three different ways. One is purely visual, where "incorrect" texels are simply not drawn on the screen. This works reasonably well for small interpenetrations, although it requires ray tracing to produce every frame of animation. Another way is to use depth fields generated by ray tracing to actually apply deformations to the meshes, which then can be efficiently rendered by any modern GPU. Finally, we can combine both of these techniques to only apply mesh deformations around boundary edges of higher layer models.

We are not concerned with preserving mesh quality or topology, as long as it does not affect the visual results. With several garments rendered together, big parts of some of them often get completely hidden under higher layer garments. So, for example, simply removing completely invisible parts of the data sets is no problem, it may even be desirable for reducing further calculation costs.

Layered ray tracing

The basic idea is very simple: for every ray intersecting several garment models, consider only the intersection with the top model, ignoring the rest, so at each screen pixel the top model appears on top, even if covered by lower layers due to incorrect positioning. By "higher" and "lower" we mean the intuitive order the clothes are put on, so an overcoat model is "higher" than an underpants model and vice versa. A naïve implementation of this idea can as well be formulated in terms of z–buffering, e. g. re–ordering texels in the z–buffer, prioritizing those belonging to a higher layer over those simply positioned closer to the camera. This will work for simple and relatively flat surfaces (Fig. 4.3). However, garment models are usually not flat — they are curved and looped, which would cause problems, most notably at the edges of the silhouette (Fig. 4.4). Let us re–formulate the problem in stricter terms and extend the basic idea to work for garments.

There are two layers of clothing: M^1 (bottom layer) and M^2 (top layer), so that M^2 should always appear on top of M^1 (Fig. 4.2). Let's cast a ray R from the camera. If R hits M^1 at a point t_1 and M^2 at a point t_2, so that $t_2 > t_1$, then the layers' relative position is incorrect in some vicinity of t_1 and t_2 (Fig. 4.2, a). It can be corrected visually by disregarding t_1 (Fig. 4.2, b) or geometrically by moving the point t_1 of surface M^1 along the ray R by a scalar value $|t_2 - t_1| + d$, where d is a small constant, big enough to prevent

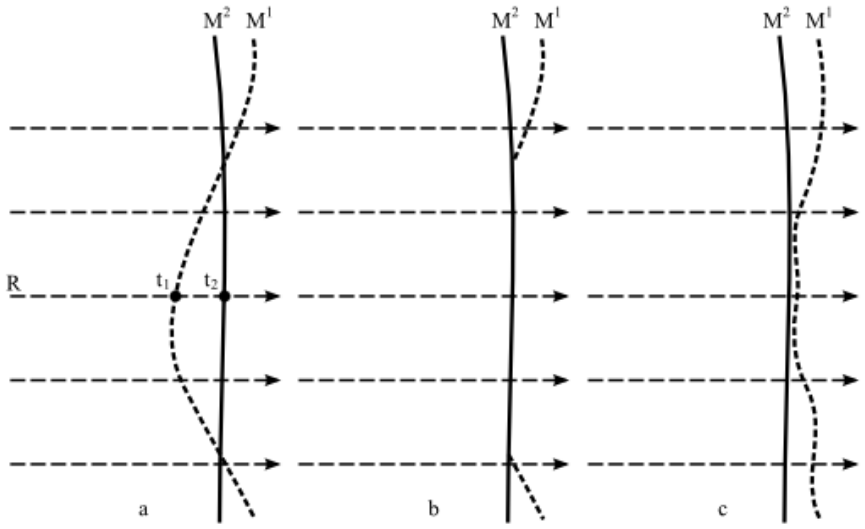

Figure 4.2: Basic idea: rays (dashed arrows), the higher layer M^2 (solid line) and the lower layer M^1 (dashed line). Problematic area (a) with M^1 showing through M^2; same area after fixing visually by "ignoring" M^1 (b) and geometrically by deforming M^1 to be positioned correctly (c)

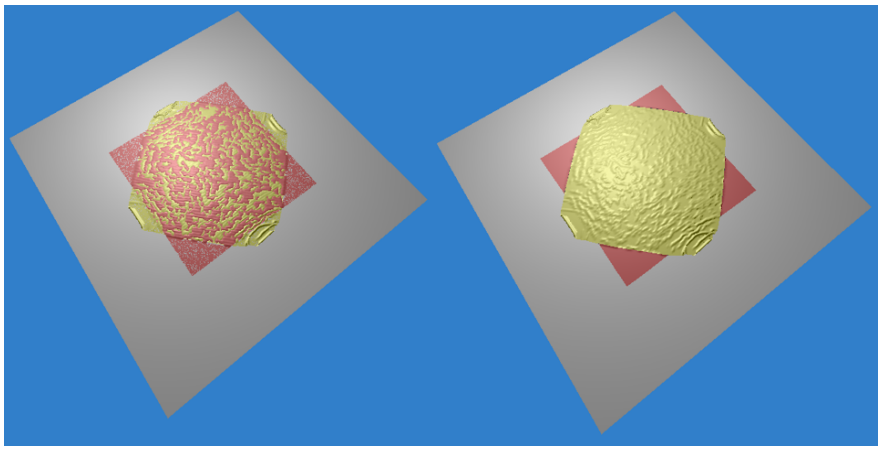

Figure 4.3: A simple case: interference and z-fighting of two sheets of deformable cloth and rigid floor, collided together (left) resolved visually (right)

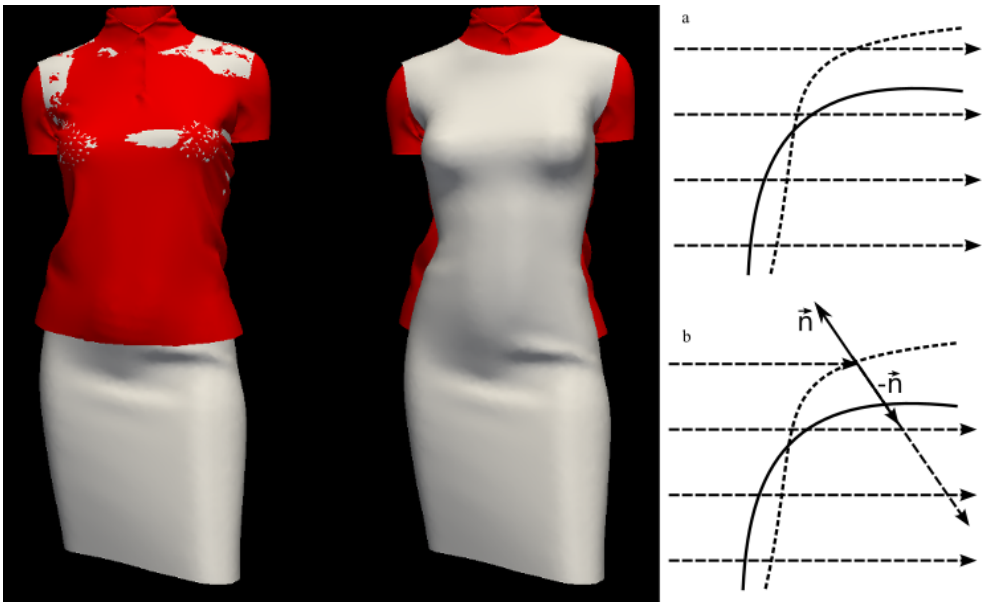

Figure 4.4: A naïve implementation used on two garments: a white dress (solid line in 2D view) worn over a red shirt (dashed line in 2D). All cloth interferences and incorrect positioning are resolved, except for the left and right sides, where the red shirt is still incorrectly shown. This happens because rays are missing one of the garments entirely at the silhouette edges (a). The proposed solution is to trace a secondary "probe ray" in the direction opposite to the surface normal (b).

possible z–fighting (Fig. 4.2, c).

The rays must be coming from the "outside" of the clothing, otherwise the layer order has to be reversed. For that, of course, we must be able to clearly distinguish between the "outside" and the "inside". In general, cloth surfaces have no orientation (both sides are "outside") — one of the reasons why collision detection is more difficult to implement for cloth surfaces than for volume hulls. However, if we only consider garment simulation, the idea of distinguishing between the "inside" and the "outside" starts making sense again. Also, the ray must be hitting the surfaces at a nearly straight angle, otherwise there's no guarantee that the ray is going to hit another layer anywhere close (or at all). To fix that, recursive ray tracing can be used, with a secondary ray cast from the hit point along the vector opposite to the surface normal (Fig. 4.4). That helps solving the problems at the edges of the silhouette as well, where a single ray would just skim one (possibly wrong) layer at a tangent. For another problematic example let us consider rendering a human dressed in a shirt from a side (Fig. 4.5): a naked arm will "disappear", because a higher layer (the shirt

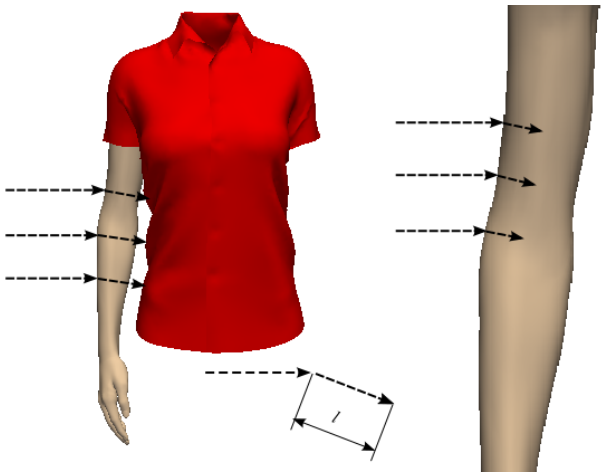

Figure 4.5: The arm "disappearing" if seen from the left, because probe rays find a higher layer object (red shirt). The proposed solution is to limit the probe ray search distance to a certain value l, roughly equal to the body part "radius".

surface) will be rendered over a lower layer (the skin surface). To prevent this, we can look for a layer with bigger index only at a fixed distance l from the previous hit point. In our study, setting l to be approximately the "radius" of the body part (e. g. an arm, a leg or the torso) has shown good results. We can also check the angle between the ray and the surface normal at the hit point to determine whether the ray is entering or exiting — an approach commonly used to simulate refraction inside transparent solid objects, it will not always work for garment models, because their surfaces are not close. Using geometric surface normals to direct secondary rays works for relatively simple surfaces, where surface normals of neighboring points have similar "outwards" directions. However, in presence of folds, creases and drapes, local surface normals may be changing significantly even inside a small local area, becoming tangential to the surface in general or even pointing in the direction opposite to each other. Therefore, we assign special vectors ("global normals") along the surface, pointing "outwards" (Fig. 4.6). They can be assigned manually at the design stage, or transferred from the underlying LOD model that has no folds, or geometric normals of the same model before draping was simulated can be preserved and used for this purpose. Because body models are smooth and their surfaces usually fit garment surfaces closely, we found that transferring geometric surface normals from the body model also works well. Note that, no matter what technique is used, this is a one–time

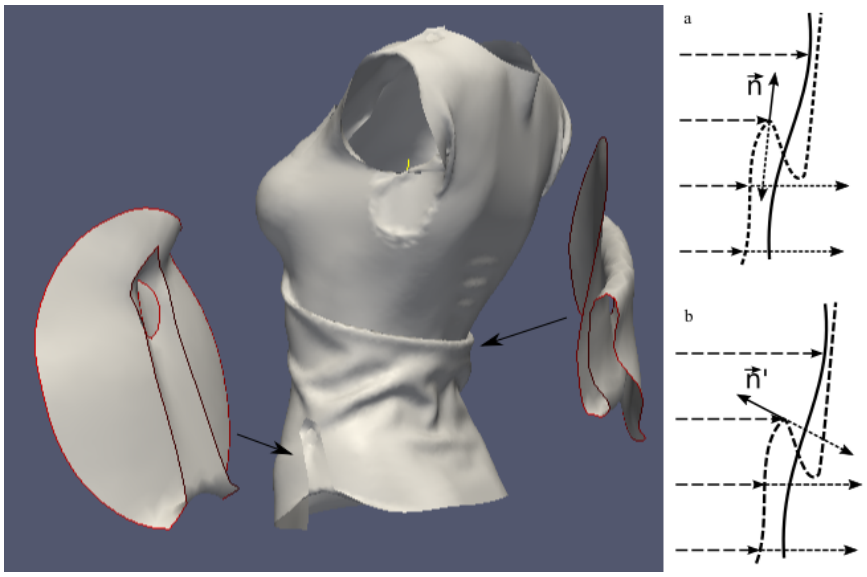

Figure 4.6: Folds and drapes of a 3D garment, making geometrical surface normal change its direction to be parallel or opposite of the surface normal "in general". In such cases, using geometric surface normals to direct the probe rays still causes them to miss garments (a). The proposed solution is to assign additional vectors ("global normals") to guide the probe rays (b).

preprocessing step, so it does not affect the overall calculation time. To assign global normal vectors $\overrightarrow{n}'$ (Fig. 4.5) and search radius l (Fig. 4.6, b) appropriately, we segment the models into their corresponding body parts (Fig. 4.7). So, for example, processing a right sleeve, we only look for rays intersecting right sleeves of other garment models, processing the left leg — for other left leg parts, and so on. That way we can also set l to be different for different body parts.

If we simply move mesh vertices according to ray traced distances, as shown in Fig. 4.2, disregarding cloth physical properties, polygon edges may get overstretched or entangled. To what degree do we need to correct that? With several garments, big parts of clothes get completely hidden under other clothes. Since they do not affect visual results, nothing needs to be done there. On the other hand, parts that are visible do affect visual results, but as they do not interfere with lower layers (after we correct the lower layers by moving their vertices), nothing is to be done there as well. In fact, we only can observe some problems around the boundaries between visible and occluded parts of the models: if we visually disregard "wrong" texels, visual gaps may appear (Fig. 4.10, b); if we deform the meshes, over–stretched polygons may become visible (Fig.

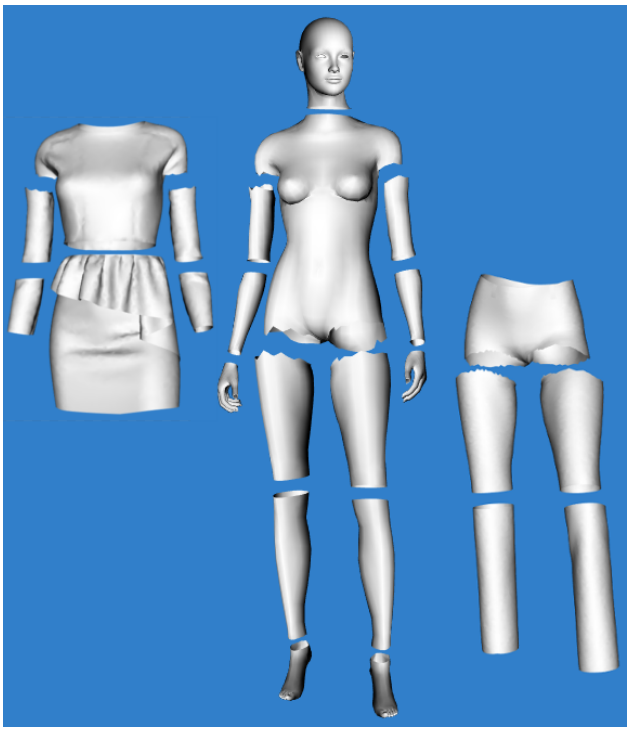

Figure 4.7: Body parts segmentation, representing the ray tracing hierarchy: corresponding parts are traced only against each other. The skirt is traced against both thighs and the torso.

4.10, c). Both of these problems can be avoided by applying smooth deformations near the boundary edges in order to reduce the inter–layer distance. *Boundary edges* are defined as the edges with only one adjacent polygon (a triangle, in case of triangular meshes), and can be found through simple topological queries, as shown in Fig. 4.8. There, for example, an edge AB is a boundary edge, because it has one and only one adjacent triangle ABC. An edge AC, on the other hand, is an inner, or non–boundary, edge, because it has two adjacent triangles ABC and ACD. Points, A and B, adjacent to the edge AB, are *boundary points*. Unless mesh topology is changed (which can happen if cloth tearing is simulated, or mesh subdivision is performed), such edges and points can be calculated once and stored for future use, and therefore don't have to be found at runtime.

To apply smooth deformations along the boundaries, probe rays are cast from the boundary points and in the direction of the global normals, looking for layers of cloth with *lower* layer number. Note that this is

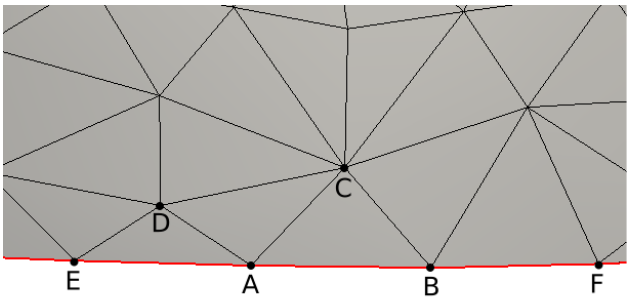

Figure 4.8: Boundary and non–boundary edges of a triangular mesh surface (shown in grey). Edges with only one adjacent triangle (EA, AB and BF on the picture) are considered boundary edges, and points adjacent to them (E, A, B and F on the picture) — boundary points.

the opposite of what we described earlier for general, non–boundary cases: there the probe rays were traced in the direction *opposite* to the global normals, and looking for layers of cloth with *higher* layer number. However, the cloth that is modified is the lower layer cloth in both cases. The reason we use different tracing order and direction is both implementation simplicity and computational efficiency: if we trace probe rays from the boundary points, we do not need to perform additional tests to determine whether the hit points are close to the boundary edges, because they are always so. When a hit point is found, the triangle it belongs to is also known, and the topological neighbors of such triangle can be found (e. g. triangles sharing at least one common point). Such topological search is performed recursively (see Fig. 4.9), with the recursion depth based on the mesh resolution.

For each set of connected boundary points (disconnected boundaries, e. g. boundaries not having common boundary points, are processed separately), such (possibly empty) area of connected triangles is determined on every lower layer cloth. RBF space mapping is used on every vertex belonging to the area, as described in Section 2.3, and the implementation details can be found below in Section 4.4.

If our approach is used together with a physically based approach, for example, one involving collision detection, we can also require better precision at the boundary edges — note that boundary edges usually constitute a tiny fraction of the entire data set, so in general this requirement will not noticeably increase computational costs.

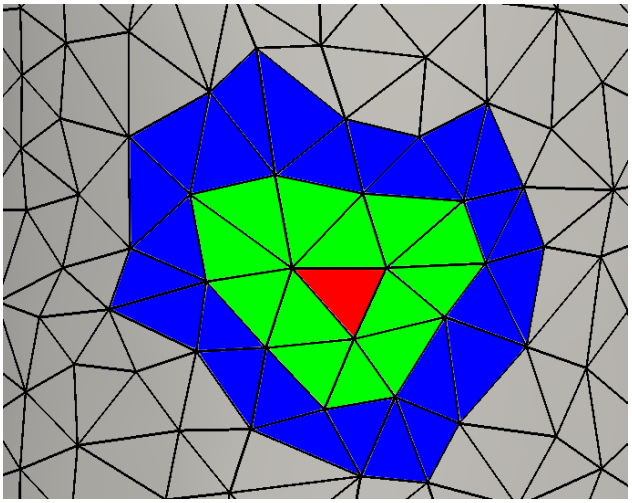

Figure 4.9: Recursively gathering topological neighbors of a triangle: the original triangle (red), the first level neighbors (green) and the second level neighbors (blue).

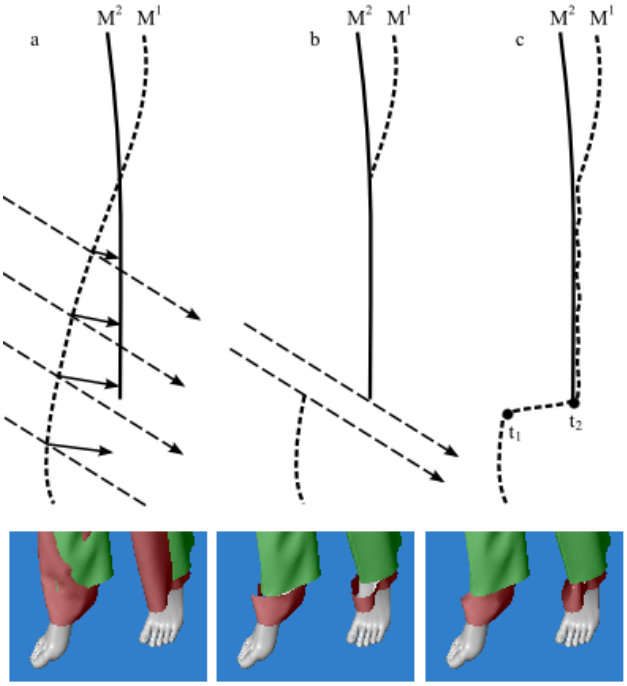

Figure 4.10: Camera rays (dashed arrows) and probe rays (solid arrows) and a problem that appears near a boundary edge of the green pants due to the green and red surfaces being too far apart (a); visual gap (b) and over–stretched edge (c) near the boundary. Solid line represents the outer layer of clothing (M^2), dashed line — the inner layer of clothing (M^1).

4.4 Algorithm

This is our particular implementation of the ideas explained above. The pre–processing step must only be done once for every model, possibly at the design stage. The "Boundary deformations" step is performed after the human figure is dressed, but before the interferences are fixed by ray tracing. From this point there are two options: to render the image with layered ray tracing, correcting interferences visually as we render, or to deform the models with layered ray tracing and then use OpenGL to render the image.

Pre–processing

1. Triangular surface meshes (unstructured grids) M^n, where $n = 0..N$ are used as input data sets. n is the layer number, with every model representing an entire layer: M^0 represents the human model, the rest are garment models. Except for $n = 0$, layer ordering plays no role at this step, because each pair M^0, $M^n|n > 0$ is pre–processed separately.

2. Models are segmented according to the body parts they cover, each segment given an integral number $s = [0..S]$. $S = 5$ is enough for static poses, with the parts numbered in the following order: torso, right arm, left arm, right leg, left leg, the skirt. For dynamic poses we use more detailed segmentation with $S = 13$ (see Fig. 4.7). The segmentation is performed manually once for every model. With ready–made models, high precision of the segmentation is not possible, but it is not required. We further denote points and triangles belonging to a particular segment s and layer n as P_s^n, T_s^n, respectively. For each body segment its l_s distance (explained above) is estimated, which is shared by all layers.

3. For each point $p \in P_s^n, n > 0$ on a garment model the closest point $p' \in P_s^0$ is found. Its normal $\overrightarrow{n}'$ is translated to the point p as an additional attribute vector. This works reasonably well except for skirts, which require some manual intervention.

4. For each $M^n, n > 0$, points belonging to boundary edges (an edge with only one adjacent triangle) of their respective polygons are marked as "boundary points" to speed up further lookup.

Changing pose (only applies if dynamic poses are used)

1. In case the pose was changed, rotation matrices are calculated for each body segment l_s.
2. For each body segment l_s its global normals $\overrightarrow{n_i}'$ are transformed by multiplying to the rotation matrix for the segment.

Boundary deformations

1. For every boundary point $p_i \in P_s^n$ a ray is cast in $\overrightarrow{n_i}'$ direction. If it intersects a lower–indexed triangle $t \in T_s^k, k < n$ and if $|h - p_i| \leq l_s$, where h is the hit point, then the hit point h shall be moved by a displacement vector $\overrightarrow{v_i} = -\overrightarrow{n_i}'(|h - p_i| + d)$, where d is a small constant (big enough to prevent z–fighting).
2. Because a hit point cannot be moved (it does not represent a vertex in the data set), vertices p_i of the triangle T_s^k are moved instead.
3. The movement vectors of p_i are smoothly propagated to their neighbors by an RBFs–weighted umbrella operator. For the base we used a Gaussian function $exp(-x^2/2c^2)$, where c is the neighborhood radius.

Layered ray tracing for visualization

1. For every ray r hitting a triangle $t \in T_s^n$ a secondary "probe" ray is cast from the hit point h in the $-\overrightarrow{n}'$ direction. If it hits a triangle belonging to T_s^k, so that $k > n$, then h is discarded and r continues as if no intersection has occurred; if not, r terminates and h is the final hit point.
2. Step 1 is performed recursively. When it terminates, the final hit point, its triangle and layer number are known.
3. The screen pixel color is calculated and drawn on the screen according to the selected coloring scheme, the texture for it is taken from the model corresponding to its layer number.

Layered ray tracing for mesh deformation

1. For every point $p_i \in P_s^n$ a ray is cast in the $-\overrightarrow{n_i}'$ direction. If it intersects a higher indexed triangle $t \in T_s^k : k > n$ and if $|h - p_i| \leq l_s$, where h is the hit point, then the point p_i shall be moved by a displacement vector $\overrightarrow{v_i} = -\overrightarrow{n_i}'(|h - p_i| + d)$, where d is a small constant (big enough to prevent z–fighting).

2. Now that all points were corrected, there might still be triangles in T^n intersecting triangles in T^k and thus showing through them incorrectly. Such triangles are removed.

4.5 Results and discussion

We implemented both variations of our approach described in Section 4.4 in C++ and CUDA C, using VTK library for general mesh handling, NVidia OptiX framework for GPU–accelerated ray tracing and NVidia PhysX for physical simulations. We used bounding volume hierarchy structures provided with OptiX to find intersections more efficiently. For the tests of our visual technique we were generating 800x800 pixel images on two different GPUs: GeForce GT 540M and GeForce GTX 660, both of them not among the newest at the time. The biggest data we used is shown in Fig. 4.12. There are two things we want to mention prior to further discussion: first, it makes little sense dressing like that in real life — we regard it more as a "torture test": the models are quite big, and so are the interfering areas. Second, we used somewhat simplistic visualization (no textures, shadows, anti–aliasing etc.) not for the lack of performance, but to make the problematic areas more clearly distinguishable.

Static poses

Testing with different garments in different combinations, we observed that for static poses performance does not significantly and consistently changes with the number of triangles itself (Fig. 4.11). The most contributing factor seems to be the area of layers interference, which is hard to express in numbers. Let us note, however, that we did not test for extreme cases, as it would be impossible, for example, to create a

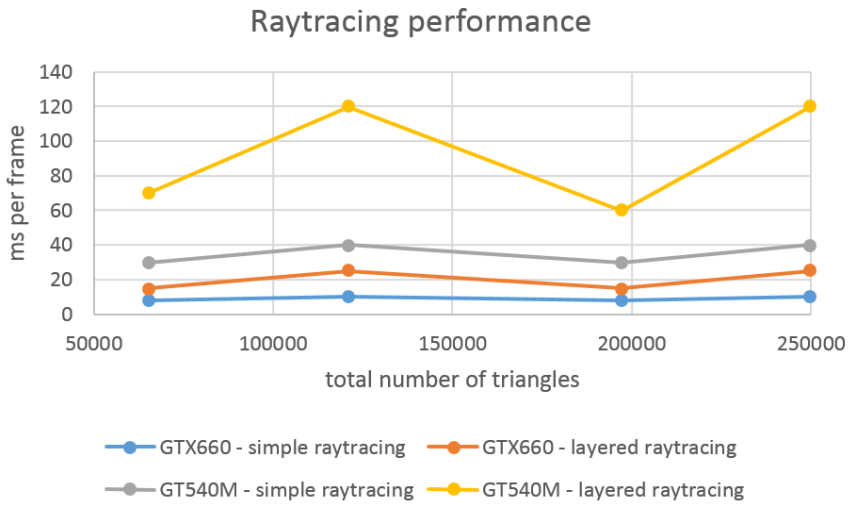

Figure 4.11: Performance of layered ray tracing on two different GPUs: computation time does not increase proportionally to the number of triangles.

realistic garment model consisting of only a few triangles instead of a few thousands triangles. As can be seen in Fig. 4.11, our test data varied in size from 65,000 to 250,000 triangles, and while the older GT 540M GPU was struggling to produce a real–time visualization, the newer GTX 660 GPU was able to produce around 40 FPS while visually correcting all the interferences on the fly.

There are several reasons why computational cost doesn't grow with the number of polygons. For example, for the models shown in Fig. 4.13 the computation time was almost exactly same as for Fig. 4.12, despite the fact that there are only 121,064 polygons, less than half of the number for Fig. 4.12. However, of the total 640,000 initial camera rays, only about one–quarter (160,000) hit any of the models at all. Bounding volume hierarchies (BVHs) are used to compute intersections efficiently, so the basic computational costs grow logarithmically, and at about 200K polygons 100K more or less don't make much difference. For every ray hitting anything but the highest or the lowest layer, we must trace at least one additional probe ray, then possibly another ray from the hit point and in the original rays' direction, and possibly repeat this process recursively, so it becomes hard predicting worst–case performance degradation.

For the mesh deformation part of our algorithm we created a "camera" that shoots rays from the mesh nodes — unlike OpenGL, OptiX allows that. For the case shown in Fig. 4.12 we have to trace 54731 rays

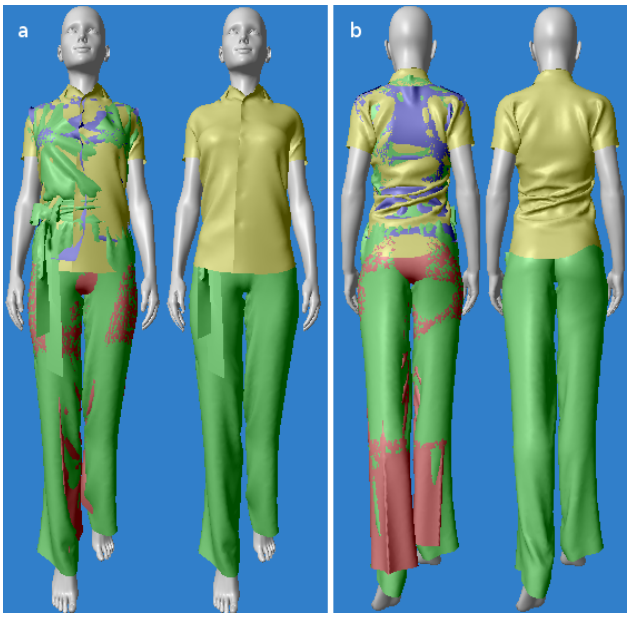

Figure 4.12: A test run: a human dressed in a jumpsuit, pants and two different shirts, 249644 triangles total. Front (a) and back (b) views of the same configuration are shown side by side in their original (left) and resolved (right) states.

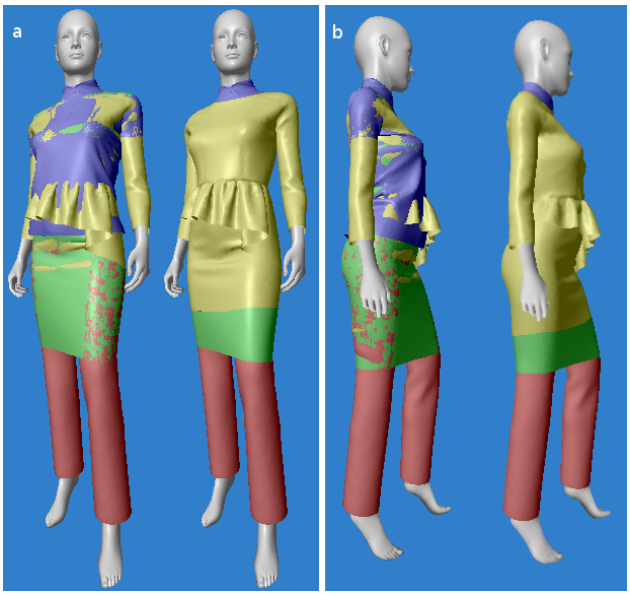

Figure 4.13: A test run: a human dressed in pants, shirt and two dresses, 121064 triangles total. Front (a) and side (b) views of the same configuration are shown side by side in their original (left) and resolved (right) states.

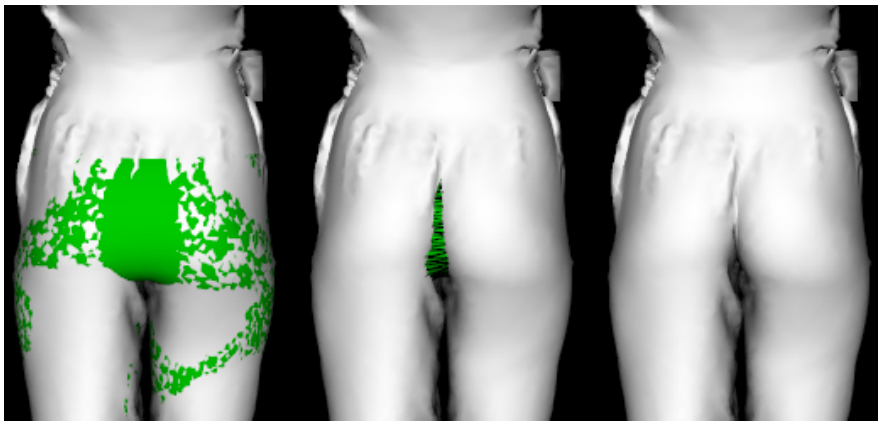

Figure 4.14: Cloth interference (left), polygons showing through in a concave area after all vertices were geometrically corrected (middle) and the same problem corrected visually (right).

(from every node except those belonging to the human model and to the topmost yellow shirt) against decreasing (as the layer number goes up) number of polygons: for the red pants (layer 1) we have to trace 14944 rays against 95806 triangles belonging to the layers 2–4; for the green jumpsuit (layer 2) — 15906 rays against 64883 triangles; for the violet shirt (layer 3) — 23881 ray against 21472 triangles. With BVHs and no secondary rays needed to be traced, it takes about 8 ms to compute all the intersections, and it doesn't have to be done at every frame, because we deform the entire meshes and then render them with OpenGL. However, with this approach it is more difficult to guarantee correct rendering: in concave areas incorrect polygons may still show through, even if all their vertices were moved under the higher layer (Fig. 4.14). Such polygons are easy to detect, and in our current implementation we simply remove them — however, that will not work with semi–transparent materials like gauze, or with sub–polygonal transparency represented with alpha–textures. If ray tracing is used for visualization as well, it is easy to blend the colors appropriately.

Dynamic cloth simulation

The combination of simple, purely geometrical deformations and visual correction at the rendering stage works well for static poses, where underlying mesh geometry does not change from frame to frame. Obviously,

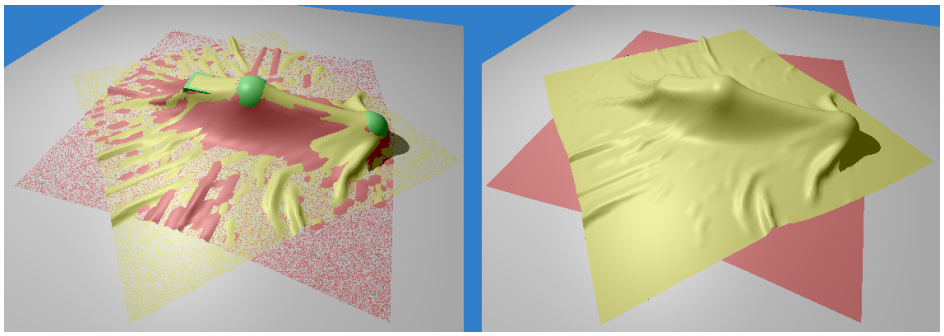

Figure 4.15: A test run: two polygonal sheets of cloth falling onto several collision primitives. Z–fighting, cloth–cloth and cloth–body interference due to imprecise physical simulation (left); same scenes after visual correction (right). It can be seen on the right, that some folds of the underlying red sheet were "flattened" by this correction, as well as sharp edges of the green cube were "smoothed".

it is not enough to handle dynamic scenes, where such big geometry changes are required. To make it look realistic, some physics has to be taken into account. To check how well our approach performs when combined with a physical simulation, we used NVidia PhysX to simulate the physics, because it is a well–known physical engine, simple enough and fast enough to be widely used in real–time video games nowadays.

For test purposes we created a scene with several collision primitives (two boxes and two spheres) and several (1 to 4) sheets of cloth of varying resolution. An example with two sheets of cloth is shown on Fig. 4.15, and the exact data is provided in Table 4.1. Two types of measurements are shown: average time per frame of animation and worst case time per frame of animation. Both are important: the former shows how fast our system performs in general, while the latter indicates whether the user is likely to experience occasional lags and freezes, when the system has to handle a particularly problematic situation. We considered time of ≤ 0.03 seconds per frame acceptable, but that, of course, depends on the application: if it is not a dedicated garment simulation and visualization, less computational time may be available for garment visualization.

Starting from version 3.0, PhysX supports cloth–to–cloth collisions, including inter–collisions. However, it requires very thick skinning parameter in order to work, and as a results sets colliding clothes very far apart — a reasonable compromise for dynamic scenes with only a few clothes colliding, but not for the problem of multiple garment rendering that we are trying to solve. In addition, as we mentioned earlier,

Number of layers	Number of triangles	Seconds per frame			
		simple, avg.	simple, worst	layered, avg.	layered, worst
1	512	0.012	0.017	0.013	0.021
2	1024	0.012	0.020	0.016	0.023
3	1536	0.014	0.030	0.016	0.032
4	2048	0.015	0.028	0.022	0.028
1	2048	0.011	0.019	0.013	0.020
2	4096	0.012	0.020	0.016	0.022
3	6144	0.013	0.027	0.017	0.030
4	8192	0.013	0.029	0.021	0.031
1	8192	0.011	0.018	0.013	0.019
2	16384	0.012	0.019	0.017	0.025
3	24576	0.014	0.025	0.019	0.027
4	32768	0.015	0.029	0.020	0.030
1	32768	0.013	0.018	0.016	0.026
2	65536	0.017	0.025	0.022	0.031
3	98304	0.021	0.031	0.027	0.032
4	131072	0.025	0.032	0.034	0.043

Table 4.1: Performance test results for simple ray tracing and our layered ray tracing of the same scenes consisting of several rigid primitives and from 1 up to 4 sheets of cloth of different resolution. 500 frames were generated for each simulation, average and worst–case performance data is provided.

if pre–simulated and pre–draped clothes are used, they tend to produce initial cloth to cloth interference, creating physically incorrect configurations and making it difficult to apply collision detection. Therefore we only use PhysX to simulate cloth interaction with rigid collision primitives (turning cloth–cloth collision off also increases performance). In other words, clothes are completely oblivious to each other during physical simulation steps, while cloth interference is resolved afterwards using our method.

While for static poses increasing triangle count did not drastically affect ray tracing performance (see Fig. 4.11), for dynamic clothes it did (see Table 4.1 and Fig. 4.16). The main reason is BVH updates: for a static pose we keep the same BVH tree for every frame, therefore we can use a high quality BVH, even if it takes time to build it (especially since it can be pre–built and cached for future use). For dynamic scenes coupled with physical simulations, on the other hand, we have to rebuild the hierarchy before we can produce the next frame. It is, therefore, important to find a compromise between ray tracing performance and BVH (or another spatial hierarchical structure) rebuilding/updating performance. OptiX programming guide recommends HLBVH2 ([22]) or Trbvh ([72]), generally favoring Trbvh, but for our test cases HLBVH2

Figure 4.16: Performance chart for the test data acquired during the tests simulations of 1 to 4 sheets of cloth interacting with several colliding primitives. Dashed line represents simple ray tracing, solid line — layered ray tracing. Black colored lines represent average, red colored lines — worst case performance (both values measured over 500 frames of simulation). Note that for the same triangle count less time is required if all the cloth triangles belong to a single layer, than when they are divided between 2, 3 or 4 layers.

showed better performance. It is also noticeable on Fig. 4.16 that the number of layers influences simulation speed much more significantly than simple polygon count: note that the spikes and drops on the chart correspond to the same triangles count (horizontal axis), with the only difference being the number of layers (4 and 1).

4.6 Limitations and future work

Relying on strictly maintained layers order, our system in its current state cannot resolve self–intersections. Also it won't work in situations where several layers interlock: for example, a shoelace in a shoe (if someone ever wants a configuration detailed enough to model the two of them separately). Other problematic examples

include a necktie with its neck strap hidden under a collar, or something like a shirt sleeve rolled up and partially hiding a sweater sleeve. It may require even more detailed segmentation than shown on Fig. 4.7, or even an automatic segmentation technique to handle every possible case.

Although our tests on static human figures show good results, and our tests with dynamic sheets of cloth and colliding primitives demonstrate that achieving real–time simulation speed is possible even with quite high resolution data (up to 130K of triangles in 4 layers of cloth, the performance of our system didn't go below 30FPS on our current hardware), combining our system with a fully operational motion–captured ragdoll is still an open task, which may uncover additional problems we have not yet considered.

In this work we always assume that lower layers of cloth do not affect higher layers of cloth, as if they are very thin and very soft. So all deformations (real or "fictional", when done in image space) go in one direction: from the outside and towards the body. We believe this is consistent with real human dressing behavior: we typically put stiffer, thicker garments on top of softer, thinner garments, but not the other way around. There are, however, exceptions, like a scarf hanging over a vest. In such cases, limited usage of cloth–cloth collisions (even with thick skinning) may be justified.

Finally, our method in its current state cannot handle cloth that, during the course of simulation, moves too far away from the body area it was close to at the beginning. This is directly consequential to the major trade–off we make with our approach, sacrificing generality for speed, and may cause problems with freely dangling parts of garments, such as belts, scarves or capes — a problem, similar in appearance to that faced by Kavan et al. [70], although for different reasons.

Chapter 5

General conclusion

In this thesis we developed a set of techniques for solving problems of deformable body simulation and visualization in real–time computer graphics applications. A general approach based on simplified physics, RBF space mapping and recursive ray tracing is developed. While traditionally deformation and visualization are two separate steps, we utilize a more hybrid approach, where some data modification happens during rendering time and in image space. We do not regard this coupling itself as an advantage — on the contrary, in general decoupling is preferred whenever possible, because it allows higher level of abstraction and system modularity. However, in our case advantage of such coupling is clear, as it let us to unify the system, reducing required number of iterations and level of data modifications. Let us also note that some other techniques for modifying models at the rendering time emerged recently, such as *vertex shaders* and *hardware tessellation*. Extensive usage of recursive ray tracing may be considered a drawback, especially taking into account the associated vendor lock of our current CUDA–based implementation, but as the GPGPU capabilities grow and technologies such as OpenCL mature, we expect that to be less and less of a problem.

We demonstrate the applicability and scope of our method in widely varying scenarios, such as evolutionary swimming optimization of elastic underwater creatures (jellyfish) and rendering simulated multi–layered cloth for dressed human figures.

In Chapter 3 we develop a system for finding optimal contraction–expansion cycle parameters of jellyfish.

Exploiting radial symmetry of jellyfish, we represent its deformations as a set of control vectors in 2D, which is treated as a space mapping function, constructed through RBF–based approximation. Using the economy principle, where the jellyfish is striving to reach maximum speed with minimum bell deformation, a genetic algorithm with sequential simulated annealing was used to find optimal values of the control vectors, minimizing the fitness function. The optimal 2D deformations are then mapped into 3D space, with recursive ray tracing used to visualize the swimming jellyfish in its natural underwater environment. Such approach is useful for automatically finding realistic–looking swimming animation parameters for jellyfish of varying size and shape, since, as we know, simple scaling does not work due to the square–cube law, a mathematical principle describing the relationship between the volume and the area as a shape's size changes: its biological implication is that with the increase in body size, body mass, proportional to the cube of the linear size (body volume), increases much faster than body strength, proportional to the square of the linear size (cross–section area), therefore the movement parameters can not be simply scaled up as well, and have to change significantly. Despite the simplicity of the mathematical model we used to represent the deformations, our results seem to be in agreement with experimental studies, cited in Chapter 3. Although we did not test it for creatures other than jellyfish, we expect it to be useful for other highly deformable aquatic animals as well.

In Chapter 4 we develop a system for plausible visualization of multi–layered clothes. As 3D rendering is produced by humans and for humans, obviously, rendering of human figures plays central role in many applications. However, human simulation and rendering is also one of the most difficult task, and that is to a large degree due to their deformable nature. It is especially true for garments, because of their high elasticity, low bending stiffness and thin layered structure. As of today, in the majority of practical real–time applications, such as 3D games or virtual reality simulators, a very simplified approach is used, where the garment is made as an integral part of the underlying human model, having the same skin–bone relationship as the body itself, and without the multi–layered structure. Needless to say that this approach is very limiting in many ways: changing of clothes is, basically, impossible — instead, a new body model wearing those clothes has to be created, and the absence of the multi–layered structure (note that in real life we

rarely wear just a single layer of clothing) often creates a very unrealistic impression of the garment being "painted" on top of the body. These simplifications are made to fit the tight requirements imposed by hardware limitations, but as the computational power grows, so does demand for higher degree of visual realism. Most approaches are based on physically–based simulations of bending, stretching and collision detection. That usually involves iterative numerical integration of motion equations. Our approach is different in a way that simple geometrical and visual enhancements are used instead of some final stages of the physical simulation. We use recursive ray tracing, originally invented to simulate shadowing, reflection and refraction by recursively casting secondary rays, for determining clothing layer order during the visualization, casting specialized "probe rays" along special pre–calculated control vectors we call "global normals". Our approach cannot replace, but can complement any other physically–based method, greatly decreasing associated computational costs, as demonstrated in the final sections of Chapter 4. Although not general enough to be used for arbitrary cloth simulation, our approach is useful in a wide variety of 3D applications involving dressed human figures.

Publications

Refereed journal

- V. Lazunin, V. Savchenko, Interactive visualization of multi–layered clothing, The Visual Computer, 2015, DOI 10.1007/s00371–015–1153–4 (in press)

International conference proceedings (refereed)

- V. Lazunin, V. Savchenko, Artificial jellyfish: evolutionary optimization of swimming, WSCG'20 conference proceedings, ISBN 978–80–86943–79–4, Part 1, pp. 131–138, Plzen, Czech Republic, June 26–28, 2012

- V. Lazunin, V. Savchenko, VR mannequin: multi–layered ray tracing for multiple garment rendering, proceedings of CGI'15, Strasbourg, France, June 24–26, 2015

- V. Lazunin. V. Savchenko, Artificial jellyfish: evolutionary simulation and foveated rendering, proceedings of The 34th JSST Annual International Conference on Simulation Technology (JSST2015), pp. 166–169, Toyama, Japan, October 12–14, 2015

Other

- V. Lazunin, V. Savchenko, Vortices formation for medusa–like objects (ECCOMAS CFD 2010), proceedings of Fifth European Conference on Fluid Dynamics, Lisbon, Portugal, June 14–17, 2010

Bibliography

[1] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An interactive out-of-core rendering framework for visualizing massively complex models. In *Rendering Techniques 2004, Proceedings of the Eurographics Symposium on Rendering*, pages 81–92, June 2004.

[2] Andreas Dietrich, Ingo Wald, and Philipp Slusallek. Interaktive darstellung hoch detaillierter landschaftsszenen (cover image). *Informatik Spektrum*, 27(5):480–480, October 2004.

[3] Sarah F. F. Gibson and Brian Mirtich. A survey of deformable modeling in computer graphics. Technical report, MERL – a Mitsubishi Electric Research Laboratory, 1997.

[4] A. Pentland and J. Williams. Good vibrations: Modal dynamics for graphics and animation. *SIGGRAPH Comput. Graph.*, 23(3):207–214, July 1989.

[5] Andrew Witkin and William Welch. Fast animation and control of nonrigid structures. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 243–252, New York, NY, USA, 1990. ACM.

[6] W. Jack Bouknight. A procedure for generation of three-dimensional half-toned computer graphics presentations. *Commun. ACM*, 13(9):527–536, September 1970.

[7] H. Gouraud. Continuous shading of curved surfaces. *Computers, IEEE Transactions on*, C-20(6):623–629, June 1971.

[8] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[9] Edwin Earl Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, 1974. AAI7504786.

[10] James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292, August 1978.

[11] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[12] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environments. *SIGGRAPH Comput. Graph.*, 20(4):133–142, August 1986.

[13] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18(3):213–222, January 1984.

[14] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. *SIGGRAPH Comput. Graph.*, 18(3):165–174, January 1984.

[15] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[16] Turner Whitted. An improved illumination model for shaded display. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '79, pages 14–, New York, NY, USA, 1979. ACM.

[17] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag.

[18] Pascal Volino and Nadia Magnenat-Thalmann. Resolving surface collisions through intersection contour minimization. *ACM Trans. Graph.*, 25(3):1154–1159, July 2006.

[19] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3d objects with radial basis functions. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 67–76, New York, NY, USA, 2001. ACM.

[20] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004.

[21] J.H. Reif, J.D. Tygar, and A. Yoshida. The computability and complexity of optical beam tracing. In *Proceedings of 31st Annual Symposium on Foundations of Computer Science*, volume 1, 1990.

[22] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. Simpler and faster hlbvh with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 59–64. ACM, 2011.

[23] J. O. Dabiri, S. P. Colin, and J. H. Costello. Morphological diversity of medusan lineages constrained by animal-fluid interactions. *The Journal of Experimental Biology*, 210:1868–1873, 2007.

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

[25] V. Lazunin and V. Savchenko. Vortices formation for medusa-like objects. In *Proceedings of Fifth European Conference on Fluid Dynamics (ECCOMAS CFD 2010)*, June 2010.

[26] D. Lipinski and K. Mohseni. Flow structures and fluid transport for the hydromedusae Sarsia tubulosa and Aequorea victoria. *The Journal of Experimental Biology*, 212:2436–2447, 2009.

[27] D. Rudolf and D. Mould. Interactive jellyfish animation using simulation. In *International Conference on Computer Graphics Theory and Applications (GRAPP)*, pages 241–248, 2009.

[28] S. P. Colin and J. H. Costello. Morphology, swimming performance and propulsive mode of six co-occuring hydromedusae. *The Journal of Experimental Biology*, 206:427–437, 2002.

[29] J. O. Dabiri, S. P. Colin, and J. H. Costello. Fast-swimming hydromedusae exploit velar kinematics to form an optimal vortex wake. *The Journal of Experimental Biology*, 206:3675–3680, 2003.

[30] J. O. Dabiri, S. P. Colin, J. H. Costello, and M. Gharib. Flow patterns generated by oblate medusan jellyfish: field measurements and laboratory analyses. *The Journal of Experimental Biology*, 208:1257–1265, 2005.

[31] M. J. McHenry and J. Jed. The ontogenetic scaling of hydrodynamics and swimming performance in jellyfish (aurelia aurita). *The Journal of Experimental Biology*, 206:4125–4137, 2003.

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

[33] M. Müller, S. Schirm, M. Teschner, B. Heidelberger, and M. Gross. Interaction of fluids with deformable solids. *Computer Animation and Virtual Worlds*, 15:159–171, 2004.

[34] H. Y. Yoon, S. Koshizuka, and Y. Oka. A particle-gridless hybrid method for incompressible flows. *International Journal for Numerical Methods in Fluids*, 30:407–424, 1999.

[35] N. Chentanez, T. G. Goktekin, B. E. Feldman, and J. F. O'Brien. Simultaneous coupling of fluids and deformable bodies. In *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*, pages 83–89, 2006.

[36] J. Hirato and Y. Kawaguchi. Calculation model of jellyfish for simulating the propulsive motion and the pulsation of the tentacles. In *18th International Conference on Artificial Reality and Telexistence*, 2003.

[37] K. Sims. Artificial evolution for computer graphics. In *Computer graphics*, pages 319–328. ACM SIGGRAPH, July 1991.

[38] K. Sims. Evolving virtual creatures. In *Computer graphics*, pages 15–22. ACM SIGGRAPH, July 1994.

[39] D. Terzopoulos, X. Tu, and R. Grzeszczuk. Artificial fishes: autonomous locomotion, perception, behavior, and learning in a simulated physical world. *Artificial Life*, 1(4):327–351, 1994.

[40] J. Tan, Y. Gu, G. Turk, and C. K. Liu. Articulated swimming creatures. In *Computer graphics*, volume 30. ACM SIGGRAPH, July 2011.

[41] V. Savchenko and L. Schmitt. Reconstructing occlusal surfaces of teeth using genetic algorithm with simulated annealing type selection. In *6th ACM Symposium on Solid Modeling and Applications*, pages 39–46, June 2001.

[42] T. A. Davis. Umfpack, an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, June 2004.

[43] D. Shepard. A two-dimensional interpolation function for irregularly spaced data. In *Proceedings of the 23th Nat. Conf. of the ACM*, pages 517–523, 1968.

[44] S. Premože, T. Tasdizen, J. Bigler, A. Lefohn, and R.T. Whitaker. Particle-based simulation of fluids. *Computer Graphics Forum*, 22(3):401–410, 2003.

[45] M. Carlson, P. J. Mucha, and G. Turk. Rigid fluid: Animating the interplay between rigid bodies and fluid. In *ACM Transactions on Graphics*, volume 23, pages 377–384, 2004.

[46] M. Desburn and M. P. Cani. Smoothed particles: A new paradigm for animating highly deformable bodies. *Computer animation and simulation*, pages 61–67, 1996.

[47] S. W. Mahfoud and D. E. Goldberg. A genetic algorithm for parallel simulated annealing. *Parallel problem solving from nature*, 2:301–310, 1992.

[48] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1986.

[49] O. Roeva, S. Fidanova, and M. Paprzycki. Influence of the population size on the genetic algorithm performance in case of cultivation process modelling. In *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, pages 371–376, Sept 2013.

[50] S. Gotshall and B. Rylander. Optimal population size and the genetic algorithm. In *Proceedings of the 2nd WSEAS International Conference on Soft Computing, Optimization, Simulation and Manufacturing Systems*, 2002.

[51] Nadia Magnenat-Thalmann and Pascal Volino. From early draping to haute couture models: 20 years of research. *The Visual Computer*, 21(8–10):506–519, 2005.

[52] Yong-Jin Liu, Dong-Liang Zhang, and Matthew Ming-Fai Yuen. A survey on cad methods in 3d garment design. *Comput. Ind.*, 61(6):576–593, August 2010.

[53] N. Magnenat-Thalmann, F. Cordier, Hyewon Seo, and G. Papagianakis. Modeling of bodies and clothes for virtual environments. In *Cyberworlds, 2004 International Conference on*, pages 201–208, Nov 2004.

[54] Frédéric Cordier, Won Sook Lee, Hyewon Seo, and Nadia Magnenat-Thalmann. From 2d photos of yourself to virtual try–on dress on the web. In *People and Computers XV – Interaction without Frontiers*, pages 31–46. 2001.

[55] Yuwei Meng, P. Y. Mok, and Xiaogang Jin. Interactive virtual try-on clothing design systems. *Comput. Aided Des.*, 42(4):310–321, April 2010.

[56] Pascal Volino, Nadia Magnenat-Thalmann, and Francois Faure. A simple approach to nonlinear tensile stiffness for accurate cloth simulation. *ACM Trans. Graph.*, 28(4):105:1–105:16, September 2009.

[57] S. Kawabata. The standardization and analysis of hand evaluation. *Textile Machinery Society of Japan*, 1980.

[58] Nobuyuki Umetani, Danny M. Kaufman, Takeo Igarashi, and Eitan Grinspun. Sensitive couture for interactive garment modeling and editing. *ACM Trans. Graph.*, 30(4):90:1–90:12, July 2011.

[59] Z. Yasseen, A. Nasri, W. Boukaram, P. Volino, and N. Magnenat-Thalmann. Sketch–based garment design with quad meshes. *Computer–Aided Design*, 45(2):562–567, 2013.

[60] Pascal Volino, Martin Courchesne, and Nadia Magnenat-Thalmann. Versatile and efficient techniques for simulating cloth and other deformable objects. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 137–144. ACM, 1995.

[61] Mikio Shinya and Marie-Claire Forgue. Interference detection through rasterization. *The Journal of Visualization and Computer Animation*, 2(4):132–134, 1991.

[62] Dave Knott and Dinesh K. Pai. CInDeR: Collision and interference detection in real–time using graphics hardware. In *Proceedings of Graphics Interface 2003*, pages 73–80, 2003.

[63] Tzvetomir Vassilev, Bernhard Spanlang, and Yiorgos Chrysanthou. Fast cloth animation on walking avatars. *Computer Graphics Forum*, 20(3):260–267, 2001.

[64] Tzvetomir Vassilev and Bernhard Spanlang. Fast GPU garment simulation and collision detection. In *WSCG 2012 Communications Proceedings I*, pages 19–26, 2012.

[65] J. Rodriguez-Navarro, M. Sainz, and A. Susin. Fast body–cloth simulation with moving humanoids. In *The 26th annual conference of the European association for computer graphics '05*, pages 85–88, 2005.

[66] Everton Hermann, François Faure, and Bruno Raffin. Ray–traced collision detection for deformable bodies. In *3rd International Conference on Computer Graphics Theory and Applications*, pages 293–299, 2008.

[67] Tzvetomir Vassilev. Collision detection for cloth simulation using ray–tracing on the GPU. *International Journal on Information Technologies & Security*, 4(4):3–12, 2012.

[68] Edilson de Aguiar, Leonid Sigal, Adrien Treuille, and Jessica K. Hodgins. Stable spaces for real-time clothing. *ACM Trans. Graph.*, 29(4):106:1–106:9, July 2010.

[69] Peng Guan, Leonid Sigal, Valeria Reznitskaya, and Jessica K. Hodgins. Multi–linear data–driven dynamic hair model with efficient hair–body collision handling. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '12, pages 295–304. Eurographics Association, 2012.

[70] Ladislav Kavan, Dan Gerszewski, Adam W. Bargteil, and Peter-Pike Sloan. Physics–inspired upsampling for cloth simulation in games. *ACM Trans. Graph.*, 30(4):93:1–93:10, July 2011.

[71] H. Yamada, M. Hirose, Y. Kanamori, J. Mitani, and Y. Fukui. Image–based virtual fitting system with garment image reshaping. In *Cyberworlds (CW), 2014 International Conference on*, pages 47–54, Oct 2014.

[72] Tero Karras and Timo Aila. Fast parallel construction of high–quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 89–99. ACM, 2013.