Meshless Approximation Methods and Applications in Physics Based Modeling and Animation

Bart Adams[1,3] Martin Wicke[2,3]

[1]Katholieke Universiteit Leuven
[2]Max Planck Center for Visual Computing and Communication
[3]Stanford University

Abstract

With growing computing power, physical simulations have become increasingly important in computer graphics. Content creation for movies and interactive computer games relies heavily on physical models, and physically-inspired interactions have proven to be a great metaphor for shape modeling.
This tutorial will acquaint the reader with meshless methods for simulation and modeling. These methods differ from the more common grid or mesh-based methods in that they require less constraints on the spatial discretization.
Since the algorithmic structure of simulation algorithms so critically depends on the underlying discretization, we will first treat methods for function approximation from discrete, irregular samples: smoothed particle hydrodynamics and moving least squares. This discussion will include numerical properties as well as complexity considerations.
In the second part of this tutorial, we will then treat a number of applications for these approximation schemes. The smoothed particle hydrodynamics framework is used in fluid dynamics and has proven particularly popular in real-time applications. Moving least squares approximations provide higher order consistency, and are therefore suited for the simulation of elastic solids. We will cover both basic elasticity and applications in modeling.

1. Introduction

Computer graphics has evolved to a stage where content creation is highly automated. Physics-based animation can automatically compute realistic behavior for dynamic systems such as liquids, gases, or elastic solids. Researchers have created algorithms that simulate complex effects such as fracture, plasticity, or phase transitions. Physically plausible interactions have also proven exceptionally useful as a modeling metaphor in shape modeling.

Currently, the most common algorithms in all of these areas are mesh or grid-based. Traditional fluid simulation uses a *Eulerian* discretization of the velocity field, i. e., the space containing the fluid is discretized. For visualization, density values (in smoke simulation), or a representation of the surface (for liquids) are then advected using the discretized velocity field. Common methods for the simulation of elastic solids typically discretize the simulated material, not the embedding space. Because the discretization grid moves along with the material, this *Lagrangian* discretization is better suited for elasticity, where the relative displacements of neighboring sample points is essential for force computation.

All of these mesh-based methods require that their simulation mesh covers the discretized domain, and its elements are non-overlapping. We will call a mesh with this property *consistent*. The shape of its elements influence the implementation of the algorithm, and usually, they are chosen to be as simple as possible, i. e. hexahedral or tetrahedral.

In this tutorial, we discuss methods that do not require a consistent mesh. Instead, the necessary functions and derivatives are interpolated from irregular samples. The only connectivity information necessary is the neighborhood relationship between samples. The neighborhood graph is relatively cheap to compute, and therefore, *meshless methods*

are ideally suited for applications in which the topology of the discretized domain changes frequently. This also leads to one of the most important differences to mesh-based methods in fluid simulation: meshless methods for fluid simulation are Lagrangian methods, discretizing the fluid instead of the embedding space. Consequently, all sample neighborhoods are recomputed in each time step. Being able to change the topology of the simulation domain quickly is essential in simulations of fracture, or for adaptive computations that add samples where more detail is needed.

Physical systems are governed by partial differential equations (PDEs). For fluid dynamics, these are the Navier-Stokes equations, a system of PDEs in the velocity at each point in space. For elasticity, the governing equations can be written as a PDE in the displacement of each point in the material. In order to solve for the dynamic behavior of these systems, we have to discretize the PDE in space and time. In other words, we will replace the governing equations, which are statements about functions, by statements about samples of functions — discrete objects which we can represent in a computer. To properly translate the governing equations into equations pertaining to samples, we need to reconstruct a continuous function (and its derivatives) from the given samples, apply the governing equations to the approximated function, and transform the results back into changes that we can apply to the samples directly.

1.1. Outline

We have written this tutorial with two goals in mind. The readers should understand the foundations of meshless methods, enabling them to not only understand the methods presented here, but apply their knowledge to develop their own variations, or analyze meshless methods they encounter in the literature. On the other hand, we have included pseudocode for all core algorithmic parts. This should ensure that the reader is able to reimplement the algorithms discussed in this tutorial without being frustrated by common pitfalls. Throughout the text, there are plenty of references that point to further reading if any topic is of particular interest.

The tutorial is organized in two parts. Sections 2–5 treat fundamentals, while sections 6–8 discuss applications based on the frameworks introduced in the first part.

In the first part of this tutorial, we will discuss in detail how to spatially discretize functions using irregular, unstructured samples. We will treat two meshless discretization approaches. *Smoothed Particle Hydrodynamics* (SPH) is a function approximation framework which has been used primarily for fluid simulation. It is fast but provides only low-order accuracy in function values and derivatives. In Section 2, we will rigorously derive its properties, and provide the background necessary to apply the framework both as a function approximation technique and for the specific problem of fluid dynamics. SPH relies on so-called *kernel functions* to reconstruct a continuous function from scattered samples. We will discuss the options for choosing these kernel functions, as they significantly impact the performance quality of SPH simulations.

Moving Least Squares (MLS) approximations are more expensive to compute, but provide higher-order accuracy, making them suitable approximation methods for elasticity computations. Since their construction involves a non-trivial amount of precomputation whenever the neighborhood graph changes, these methods are not usually used for fluid simulation. They have shown great potential in the simulation of elastic solids, where fracture can be computed without costly remeshing. Adaptive simulations are also far easier to achieve. Section 3 will introduce MLS approximation.

After giving a concluding comparison between SPH and MLS in Section 4, we will turn our attention to search data structures in Section 5. These are essential for computing neighborhoods and are used heavily in any meshless simulation. The most common data structures used for neighborhood lookups are kd-trees and spatial hashing. We will treat their complexities and pitfalls, and discuss scenarios in which each is optimal.

Part two of this tutorial deals with applications of the methods introduced in part one. SPH is used for fluid simulation, and Section 6 discusses the basic technique, as well as several extensions. We will see how the Navier-Stokes equations can be discretized using irregular, Lagrangian samples, and discuss the inherent advantages and limitations of meshless Lagrangian fluid simulation compared to the traditional mesh-based Eulerian setting.

Simulations of elastic objects require higher-order accuracy. In a meshless setting, the MLS function approximation can be used to implement meshless simulations of continuum elasticity. Section 7 treats the basics of continuum elasticity, and we discuss how to apply MLS approximations to the problem. We will briefly touch upon simulation of more complex phenomena such as plasticity and fracture, with special emphasis on sampling issues.

Natural deformations as computed by continuum elasticity simulations can be used in shape modeling. Section 8 discusses how to use the basic simulation developed in Section 7 in a modeling application. We can use the physical framework not only for shape modeling, but also for motion modeling. Since physical simulations produce natural deformations, we can use non-linear optimization paired with objective functions derived from elasticity to compute optimal motion paths between keyframes.

Finally, we conclude with some high-level considerations in Section 9.

I. Function Approximation from Unstructured Samples

This part of the tutorial will deal with the problem of function approximation from unstructured samples. Given samples of a function at irregular locations, we want to reconstruct the original function (or an approximation). The key difference between mesh-based and meshless methods for function approximation is the additional structure that a consistent mesh offers. A consistent mesh partitions the sampled space into disjoint elements or cells. These additional guarantees can be used in various ways: Many finite element methods define the value of the reconstructed function at any point $\mathbf{x}$ as a linear combination of only those samples that are corners of the element that contains $\mathbf{x}$. We can integrate over the whole domain by simply adding the contributions of all elements.

For regularly sampled domains, regular or adaptive grids (octrees) provide a simple way of constructing such a consistent mesh. Computing meshes for irregular samples is not straightforward. Most commonly, Delaunay methods are used, but numerical problems due to slivers are common and hard to avoid, especially in dynamic settings where frequent re-meshing is necessary. Repairing or modifying a mesh after local resampling is often a global operation.

Meshless methods do not require a consistent mesh, and work on *unstructured* samples instead. For these methods, it is sufficient to know the distances between samples. Meshless methods are typically designed to be local, i. e. evaluating a function only involves values from nearby samples. These neighborhoods can be efficiently computed using search data structures like kd-trees or hash grids (see Section 5). Since inter-sample distances and neighborhoods can be easily recomputed, resampling is usually not a problem, and generally a local operation. On the downside, we lose the space partition property of a consistent mesh.

In the following, we introduce two meshless methods for function approximation: smoothed particles hydrodynamics and moving least squares. There are other methods that have not found widespread use in animation, for example approximation using radial basis functions. Although they are important in other areas of computer graphics, we will not treat these techniques in this tutorial.

2. Function Approximation using Smoothed Particle Hydrodynamics

In this section, we derive the basic formulation of *Smoothed Particle Hydrodynamics* (SPH). The term smoothed particle hydrodynamics was coined by Gingold and Monaghan [GM77], who introduced the method as a way to simulate interstellar gas. The function approximation method that is at the core of SPH is older, and originated from statistics as a method to approximate probability distributions from scattered samples [Ros56]. Lucy [Luc77] independently rediscovered the technique. In this tutorial, we will discuss the derivation as well as the most common formulation of SPH. The excellent review article [Mon05], or the book [LL03] give a good overview of different variations of the technique developed since its inception. While SPH is often discussed in the context of fluid animation, its function approximation framework is general. We will treat it independently of applications in this section, and return to the application of fluid dynamics in Section 6.

Consider the problem of reconstructing an (unknown) function f from a set of irregular samples $f_i = f(\mathbf{x}_i)$, where each sample has an associated importance weight m_i.

Using the Dirac-delta function δ, we can rewrite $f(\mathbf{x})$ as a convolution

$$f(\mathbf{x}) = \int_{\mathbf{x}'} f(\mathbf{x}')\delta(\|\mathbf{x}-\mathbf{x}'\|)\,\mathrm{d}V. \tag{1}$$

This doesn't yet help us, since we cannot evaluate f everywhere. We therefore replace δ with a *kernel* function ω_h. As long as $\int \omega_h = 1$, we recover a smoothed function $\tilde{f}$:

$$\tilde{f}(\mathbf{x}) = \int_{\mathbf{x}'} f(\mathbf{x}')\omega_h(\|\mathbf{x}-\mathbf{x}'\|)\,\mathrm{d}V. \tag{2}$$

We can discretize the integral in (2) into a sum over all sample points to obtain the SPH approximation

$$\langle f \rangle(\mathbf{x}) = \sum_i f_i \omega_h(\|\mathbf{x}_i-\mathbf{x}\|)V_i. \tag{3}$$

The volumes V_i associated with each sample point are still unknown. Drawing upon a physical metaphor, we associate the importance weights m_i with mass and observe that volume equals mass divided by density:

$$V_i = \frac{m_i}{\rho_i}. \tag{4}$$

We will see in Section 6 how this analogy is useful when we apply the approximation framework to physical problems. The density ρ_i is the sampling density around the sample i. We can measure ρ using (3):

$$\begin{aligned}\rho_i = \langle\rho\rangle(\mathbf{x}_i) &= \sum_j \omega_h(\|\mathbf{x}_i-\mathbf{x}_j\|)\,\rho_j V_j \\ &= \sum_j \omega_h(\|\mathbf{x}_i-\mathbf{x}_j\|)\,\rho_j \frac{m_j}{\rho_j} \\ &= \sum_j \omega_h(\|\mathbf{x}_i-\mathbf{x}_j\|)\,m_j.\end{aligned} \tag{5}$$

We can obtain the same result by defining a mass function in the continuous setting

$$m(\mathbf{x}) = \begin{cases} m_i & \mathbf{x} = \mathbf{x}_i, \\ 0 & \text{otherwise}, \end{cases} \tag{6}$$

which concentrates the weights of the samples at the sample points. We then smear out the weight over the area of support of the kernel by setting the density at each point to the

smoothed version of $m(\mathbf{x})$:

$$\begin{aligned}\rho_i = \tilde{m}(\mathbf{x}_i) &= \int_{\mathbf{x}'} m(\mathbf{x}')\omega_h(\|\mathbf{x}_i - \mathbf{x}'\|)\,\mathrm{d}V \\ &= \sum_j \omega_h(\|\mathbf{x}_i - \mathbf{x}_j\|)\, m(\mathbf{x}_j) \qquad (7) \\ &= \sum_j \omega_h(\|\mathbf{x}_i - \mathbf{x}_j\|)\, m_j\end{aligned}$$

The reader should employ the derivation for ρ_i they find most intuitive. Note that in the absence of weights ($m_i = 1$), the density serves as a simple normalization term that makes sure that the kernel weights add up to one. In that case, and for a specific choice of kernel function, the method is equivalent to Shepard's scattered data interpolation [She68].

Note that even if the kernel functions have sufficient support (i. e. the whole region of interest is covered by the union of the kernel support regions), SPH approximation in general do not recover constant functions exactly. This problem will lead us to introduce the more sophisticated moving least squares interpolation in Section 3.

We still have to choose the kernel function ω_h before the SPH approximation (3) is fully defined.

2.1. Kernel Functions

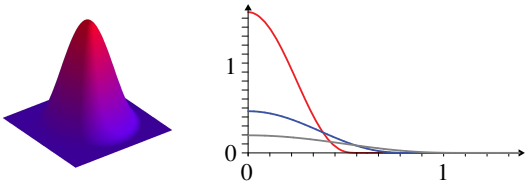
We have a wide range of options when choosing kernel functions. Figure 1 (a) shows a plot of a typical 2D kernel function. In order to produce valid results, kernel functions need to fulfill a number of requirements. We have already encountered one essential requirement for admissible kernel functions: they must be normalized.

$$\int_{\mathbf{x}} \omega_h(\|\mathbf{x}\|)\mathrm{d}V = 1 \qquad (8)$$

The kernel functions are one-dimensional ($\mathbb{R} \to \mathbb{R}$), however, the normalization has to consider the space in which the samples points live. This means that kernel functions need to be renormalized for each space in which the function interpolation is to be performed. The normalization criterion yields to $\int 2\pi r\omega_h(r)\mathrm{d}r = 1$ for two-dimensional kernels, and $\int 4\pi r^2\omega_h(r)\mathrm{d}r = 1$ in three dimensions.

In our derivation of the SPH approximation, we have silently introduced the *smoothing parameter* h. Kernels are parameterized with h, allowing control over how far the influence of each sample point reaches. Figure 1 (b) shows a kernel function for several values of h. The choice of h influences the quality of the resulting function reconstruction, and depends on the sampling density. Too large values of h produce unnecessarily smooth reconstructions, while small values of h might lead to region in which no sample has non-zero weight. To ensure that the approximation (3) converges to the continuous result (1), it is crucial that the kernel function converges to a Dirac-delta function as h goes to zero.

If the kernels have local support, only a small fraction



(a) (b)

Figure 1: *(a) A polynomial kernel function (see Eq. 86 for a definition). (b) $\omega_h(d)$ for several values of h, red: $h = 1$, blue: $h = \frac{3}{2}$, and gray: $h = 2$.*

of the samples have non-zero influence on the approximation (3) at any point. Acceleration data structures can then be used to quickly find neighboring samples. These will be discussed in Section 5.

As can be verified in (3), the smoothness of the kernel functions determines the smoothness of the reconstructed function. It is therefore desirable that the kernel function is as smooth as possible. Although it is tempting to use higher-order interpolation kernels that work well in other contexts, negative values of ω_h are problematic when the sample points are not equi-distant.

Note that even though in this tutorial, the kernels are defined as radially symmetric, this is not a strict requirement. All derivations herein can be adapted for kernels that are anisotropic, as long as they fulfill the requirements listed above.

Gingold and Monaghan initially proposed normalized Gaussian kernel [GM77], while Lucy proposed to use polynomial splines [Luc77]. Polynomial spline kernels, while of lower smoothness, have local support. Often, several kernel functions are used for different interpolation tasks within the same application [MCG03, CBP05]. If the kernel function has a singularity at zero, the SPH approximation is *interpolating*. This is often desirable, but comes at the expense of highly unstable gradient approximations as the kernel gradients tend to infinity for nearby sample points. Appendix A contains some good polynomial kernel functions for use in $\mathbb{R}^2$ and $\mathbb{R}^3$.

2.2. Approximations of Differential Operators

In order to apply SPH approximations to the solution of partial differential equations, we need not only a reconstruction of the continuous function f, but also the derivatives of the function.

Differential operators can be directly applied to the SPH approximation $\langle f \rangle$. In this section, we will introduce approximations for the gradient and Laplacian of a real-valued function, as well as for the divergence of a vector-valued function. All expressions are summarized in Table 1.

Since the sample values f_i are constants, we can write approximation of ∇f as

$$\langle \nabla f \rangle (\mathbf{x}) = \sum_i f_i \nabla \omega_h(\|\mathbf{x} - \mathbf{x}_i\|) V_i, \tag{9}$$

where the gradient $\nabla \omega_h(\|\mathbf{x} - \mathbf{x}_j\|)$ can be rewritten in terms of the kernel derivative

$$\nabla \omega_h(\|\mathbf{x} - \mathbf{x}_i\|) = \frac{\mathbf{x} - \mathbf{x}_i}{\|\mathbf{x} - \mathbf{x}_i\|} \omega_h'(\|\mathbf{x} - \mathbf{x}_i\|). \tag{10}$$

Other linear operators can be treated similarly: the Laplacian Δf can be approximated as

$$\langle \Delta f \rangle (\mathbf{x}) = \sum_i f_i \Delta \omega_h(\|\mathbf{x} - \mathbf{x}_i\|) V_i, \tag{11}$$

where $\Delta \omega_h(\|\mathbf{x} - \mathbf{x}_i\|) = \omega_h''(\|\mathbf{x} - \mathbf{x}_i\|)$, and the divergence of a vector-valued function $\mathbf{f}$ becomes

$$\langle \nabla \cdot \mathbf{f} \rangle (\mathbf{x}) = \sum_i \mathbf{f}_i \cdot \nabla \omega_h(\|\mathbf{x} - \mathbf{x}_i\|) V_i. \tag{12}$$

The accuracy of the approximations of derivative quantities strongly depends on the distribution of sample points within the support region of the kernel function. For highly irregular sample distributions, the differential properties can be very noisy. Larger values of h provide more sample points and add stability to the derivative estimates. However, larger values of h also imply more smoothing, which might not be desirable.

Fortunately, we have some freedom in defining "correct" approximations to these differential quantities. By choosing the approximation carefully, we can enforce specific properties that are important in the context of the application. In many applications, we are interested in the gradient of a function at the sample points (but not necessarily in between points). From (9), we can see that the gradient approximation $\langle \nabla f \rangle (\mathbf{x}_i)$ can yield non-zero values even if the function $f(\mathbf{x}) = c$ is constant. Below, we will discuss several ways of defining $\langle \nabla f \rangle$ at the sample points to rectify the situation.

Note that if we substract any constant function g from f, the gradient of f will remain unchanged. We can then enforce a zero gradient for constant functions by substracting the constant f_i when computing $\langle \nabla f \rangle (\mathbf{x}_i)$:

$$\begin{aligned} \nabla f(\mathbf{x}_i) &\approx \langle \nabla [f - f_i] \rangle (\mathbf{x}_i) \\ &= \sum_j (f_j - f_i) \nabla \omega_h(\|\mathbf{x}_i - \mathbf{x}_j\|) V_j. \end{aligned} \tag{13}$$

The above gradient expression yields zero gradients for constant functions. There are different methods to derive the above result, for a more general derivation, see [Mon05]. The same reasoning can be applied to the divergence and Laplace operators, leading to the corrected expressions

$$\langle \nabla \cdot \mathbf{f} \rangle (\mathbf{x}_i) = \sum_j (\mathbf{f}_j - \mathbf{f}_i) \cdot \nabla \omega_h(\|\mathbf{x}_i - \mathbf{x}_j\|) V_j, \tag{14}$$

$$\langle \Delta f \rangle (\mathbf{x}_i) = \sum_j (f_j - f_i) \Delta \omega_h(\|\mathbf{x}_i - \mathbf{x}_j\|) V_j. \tag{15}$$

Techniques for finding alternative valid approximations are usually based on finding expressions that are equivalent in the continuous setting but lead to different discretizations. For example, we can use the product rule for differentiation

$$\nabla(\rho f) = f \nabla \rho + \rho \nabla f \quad \Rightarrow \quad \nabla f = \frac{\nabla(\rho f)}{\rho} - \frac{f \nabla \rho}{\rho} \tag{16}$$

to approximate $\nabla f(\mathbf{x}_i)$ as

$$\begin{aligned} \nabla f(\mathbf{x}_i) &\approx \frac{\langle \nabla(\rho f) \rangle (\mathbf{x}_i)}{\rho_i} - \frac{f_i \langle \nabla \rho \rangle (\mathbf{x}_i)}{\rho_i} \\ &= \frac{1}{\rho_i} \sum_j m_j (f_j - f_i) \nabla \omega_h(\|x_i - x_j\|). \end{aligned} \tag{17}$$

Note the subtle difference between the approximations (13) and (17). Eq. 17 also gives zero gradients for constant functions. Another variation of the gradient approximation that is important in particular for fluid simulation (see also Section 6.1) can be derived similarly. By observing that

$$\begin{aligned} \nabla \left[\frac{f}{\rho} \right] &= \frac{\rho \nabla f - f \nabla \rho}{\rho^2} \\ \Rightarrow \quad \nabla f &= \rho \left(\nabla \left[\frac{f}{\rho} \right] + \frac{f \nabla \rho}{\rho^2} \right), \end{aligned} \tag{18}$$

we find

$$\begin{aligned} \nabla f(\mathbf{x}_i) &\approx \rho_i \left(\left\langle \nabla \left[\frac{f}{\rho} \right] \right\rangle (\mathbf{x}_i) + \frac{f_i \langle \nabla \rho \rangle (\mathbf{x}_i)}{\rho_i^2} \right) \\ &= \rho_i \sum_j m_j \left(\frac{f_j}{\rho_j^2} - \frac{f_i}{\rho_i^2} \right) \nabla \omega_h(\|x_i - x_j\|). \end{aligned} \tag{19}$$

Eq. 19 has important symmetry properties that we will revisit in Section 6.1.

One final word of caution: one should not take results derived in the continuous setting for granted, even if they hold for mesh-based approximations. Having defined gradient, divergence, and Laplace operators, it is instructive to examine the identity $\Delta f = \nabla \cdot \nabla f$ which holds in the continuous setting, and carries over to most mesh-based function approximation methods. As is easily verified in Eqns. 9–12, this identity does not generally hold in the SPH setting:

$$\langle \Delta f \rangle \neq \langle \nabla \cdot \langle \nabla f \rangle \rangle. \tag{20}$$

While it is certainly possible to define an approximation to the Laplacian which preserves this identity, this comes at a computational cost, and might carry additional disadvantages.

3. Function Approximation using Moving Least Squares

The SPH method discussed in the previous section has in general poor accuracy and lacks already zero order consistency, i.e., it does not always recover constant functions. In this section, we will discuss an alternative particle approximation method that can be constructed to have any order

of consistency, i.e., it recovers polynomial functions up to the desired order exactly, which is for example important for elastic solid simulations. This of course comes at the cost of more involved computations. However, when the particle sampling and neighborhood relations remain constant, much can be precomputed and evaluation boils down to computing simple linear combinations, making the method as efficient as SPH. This method was proposed in the graphics community [MKN*04, PKA*05] for use in elastic solid simulation, where indeed the material does not deform significantly and particle samplings remain constant.

The approach discussed in this section achieves its higher order consistency through moving least squares approximations (MLS), and will hence be referred to as the MLS method.

3.1. Shape Function Approximation

We again consider the problem of reconstructing an unknown function f from a set of irregular samples $f_i = f(\mathbf{x}_i)$. This function will locally be defined as a polynomial $\langle f \rangle(\mathbf{x}) = \mathbf{p}^T(\mathbf{x})\mathbf{a}$, where $\mathbf{p}(\mathbf{x})$ is a complete polynomial basis of order n. For example for dimension $d = 3$ and order $n = 1$, we have $\mathbf{p}(\mathbf{x}) = [1\ x\ y\ z]^T$, with $\mathbf{x} = [x\ y\ z]^T$. The vector $\mathbf{a}$ is the coefficient vector and for our example will have the form $\mathbf{a} = [a\ b\ c\ d]$.

Given the polynomial basis, the approximation $\langle f \rangle$ is now found as the polynomial that locally best fits the sample data in a least squares sense. More formally, given an evaluation point $\mathbf{x}$ and a chosen basis $\mathbf{p}(\mathbf{x})$, we wish to obtain the coefficient vector $\mathbf{a}$ that minimizes the error

$$E = \sum_i \omega_{h_i}(\|\mathbf{x} - \mathbf{x}_i\|)\left(\mathbf{p}^T(\mathbf{x}_i)\mathbf{a} - f_i\right)^2. \tag{21}$$

Here, the summation is over all particles and nearby particle contributions are weighted according to a weight function ω_{h_i} similar to the one used for SPH approximations (cf. Section 2). The support radius h_i can be uniform or particle dependent and defines each particle's influence region.

The coefficients are found as $\mathbf{a} = \arg\min E$ and it can be easily seen by differentiation that they are the solution of the linear system of equations

$$\sum_i \omega_{h_i}(\|\mathbf{x} - \mathbf{x}_i\|)\mathbf{p}(\mathbf{x}_i)\left(\mathbf{p}^T(\mathbf{x}_i)\mathbf{a} - f_i\right) = \mathbf{0}. \tag{22}$$

Solving this system yields

$$\mathbf{a} = \mathbf{M}(\mathbf{x})^{-1}\sum_i \omega_{h_i}(\|\mathbf{x} - \mathbf{x}_i\|)\mathbf{p}(\mathbf{x}_i)f_i, \tag{23}$$

where

$$\mathbf{M}(\mathbf{x}) = \sum_i \omega_{h_i}(\|\mathbf{x} - \mathbf{x}_i\|)\mathbf{p}(\mathbf{x}_i)\mathbf{p}^T(\mathbf{x}_i) \tag{24}$$

is called the moment matrix. Note that $\mathbf{M}$ does not depend on the particle function values f_i and hence, given that the particle distribution and coupling remains constant, can be precomputed for every $\mathbf{x}$. The final approximation is now given as

$$\begin{aligned}\langle f \rangle(\mathbf{x}) &= \mathbf{p}^T(\mathbf{x})\mathbf{a} \\ &= \mathbf{p}^T(\mathbf{x})\mathbf{M}(\mathbf{x})^{-1}\sum_i \omega_{h_i}(\|\mathbf{x} - \mathbf{x}_i\|)\mathbf{p}(\mathbf{x}_i)f_i.\end{aligned} \tag{25}$$

This is often written as

$$\langle f \rangle(\mathbf{x}) = \sum_i \Phi_i(\mathbf{x})f_i, \tag{26}$$

where

$$\Phi_i(\mathbf{x}) = \omega_{h_i}(\|\mathbf{x} - \mathbf{x}_i\|)\mathbf{p}(\mathbf{x})^T\mathbf{M}(\mathbf{x})^{-1}\mathbf{p}(\mathbf{x}_i) \tag{27}$$

are called *shape functions*. Similar to the moment matrices, the shape functions do not depend on f_i and can hence be precomputed and reused for varying f_i if the particle sampling is constant. Hence, in this setting, evaluating the MLS approximation using Eq. 26 becomes as efficient as evaluating an SPH approximation. Nevertheless, (pre)computing the shape functions requires inversion of a moment matrix which can potentially fail if this matrix becomes singular.

Note that there are other ways of deriving Eq. 26, for example using Taylor series expansion [FM03].

The weight functions ω_{h_i} can have any form (interpolating or not, isotropic or anisotropic, local or global) and do not have to be normalized as opposed to the weight functions used for SPH approximations. Of interest are the shape functions Φ_i, which have, by construction very desirable properties such as consistency up to the order of the used basis.

3.2. Consistency

Given a basis $\mathbf{p}(\mathbf{x})$ of order n, consistency up to this order can be easily proved by showing that $\langle \mathbf{p} \rangle(\mathbf{x}) = \mathbf{p}(\mathbf{x})$, i.e., the basis is approximated exactly for every $\mathbf{x}$. This is easily proved as

$$\begin{aligned}\langle \mathbf{p} \rangle^T(\mathbf{x}) &= \sum_i \Phi_i(\mathbf{x})\mathbf{p}^T(\mathbf{x}_i) \\ &= \sum_i \omega_{h_i}(\|\mathbf{x} - \mathbf{x}_i\|)\mathbf{p}^T(\mathbf{x})\mathbf{M}(\mathbf{x})^{-1}\mathbf{p}(\mathbf{x}_i)\mathbf{p}^T(\mathbf{x}_i) \\ &= \mathbf{p}^T(\mathbf{x})\mathbf{M}(\mathbf{x})^{-1}\sum_i \omega_{h_i}(\|\mathbf{x} - \mathbf{x}_i\|)\mathbf{p}(\mathbf{x}_i)\mathbf{p}^T(\mathbf{x}_i) \\ &= \mathbf{p}^T(\mathbf{x})\mathbf{M}(\mathbf{x})^{-1}\mathbf{M}(\mathbf{x}) \\ &= \mathbf{p}^T(\mathbf{x}).\end{aligned}$$

Here we used the definition of the shape functions in the first step and the definition of the moment matrix in the third step. Since the basis is reconstructed exactly, every linear combination of the basis is also recovered exactly and the MLS approximation method is consistent up to the used basis order.

Higher order basis functions should be used when higher accuracy is required. However, this not only comes at the larger computational cost of having to invert larger moment

matrices, stability problems might arise such as having to deal with (near-)singular moment matrices.

3.3. Stability

A simple example illustrates the problem of singular moment matrices. Assume that all particles lie in the plane $z=0$ in $\mathbb{R}^3$ and that a linear basis $\mathbf{p}(\mathbf{x}) = [1\ \mathbf{x}]^T = [1\ x\ y\ z]^T$ is used. It is easy to see that the moment matrix will always be singular:

$$\begin{aligned}\mathbf{M}(\mathbf{x}) &= \sum_i \omega_{h_i}(\|\mathbf{x}-\mathbf{x}_i\|)\mathbf{p}(\mathbf{x}_i)\mathbf{p}^T(\mathbf{x}_i)\\ &= \sum_i \omega_{h_i}(\|\mathbf{x}-\mathbf{x}_i\|)\begin{pmatrix} 1 & x_i & y_i & z_i \\ x_i & x_i^2 & x_iy_i & x_iz_i \\ y_i & x_iy_i & y_i^2 & y_iz_i \\ z_i & x_iz_i & y_iz_i & z_i^2 \end{pmatrix}\\ &= \sum_i \omega_{h_i}(\|\mathbf{x}-\mathbf{x}_i\|)\begin{pmatrix} 1 & x_i & y_i & 0 \\ x_i & x_i^2 & x_iy_i & 0 \\ y_i & x_iy_i & y_i^2 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.\end{aligned}$$

It is hence clear that care has to be taken when evaluating the moment matrices. Even in near-singular cases it is good practice to use safe inversion methods such as using the Singular Value Decomposition (SVD). Next to the fact that the MLS method is computationally more involved than the SPH method, the singularity problem renders it impractical for particle fluid simulation. Indeed, thin sheets and isolated particles often appear for splashing fluids which would yield singular moment matrices.

In addition to using safe inversion using SVD, it is advised to center the polynomial basis at the evaluation point [Saa86] to improve the conditioning number of the moment matrix. Instead of using the basis functions $\mathbf{p}(\mathbf{x}_i)$ in Eq. 27, one can equally well use the translated and scaled basis functions $\mathbf{p}((\mathbf{x}_i-\mathbf{x})/h)$ (for now assume $h_i = h$ is constant). The resulting modified moment matrix

$$\tilde{\mathbf{M}}(\mathbf{x}) = \sum_i \omega_h(\|\mathbf{x}-\mathbf{x}_i\|)\mathbf{p}(\frac{\mathbf{x}_i-\mathbf{x}}{h})\mathbf{p}^T(\frac{\mathbf{x}_i-\mathbf{x}}{h}) \tag{28}$$

has a better condition number and can be more stably inverted. Note that the resulting shape functions will now have the form

$$\tilde{\Phi}_i(\mathbf{x}) = \omega_h(\|\mathbf{x}-\mathbf{x}_i\|)\mathbf{p}(\mathbf{0})^T\tilde{\mathbf{M}}(\mathbf{x})^{-1}\mathbf{p}(\frac{\mathbf{x}_i-\mathbf{x}}{h}). \tag{29}$$

In the adaptive setting, one should use for h the average particle support radius in the above formulation.

3.4. Approximation of Differential Operators

First-order derivatives of $\langle f\rangle$ are obtained as

$$\frac{\partial\langle f\rangle(\mathbf{x})}{\partial\mathbf{x}_{(k)}} = \sum_i \frac{\partial\Phi_i(\mathbf{x})}{\partial\mathbf{x}_{(k)}} f_i, \tag{30}$$

where derivatives of the shape functions using the chain rule are

$$\begin{aligned}\frac{\partial\Phi_i(\mathbf{x})}{\partial\mathbf{x}_{(k)}} &= \frac{\partial\omega_{h_i}(\|\mathbf{x}-\mathbf{x}_i\|)}{\partial\mathbf{x}_{(k)}}\mathbf{p}^T(\mathbf{x})\mathbf{M}(\mathbf{x})^{-1}\mathbf{p}(\mathbf{x}_i)\\ &+ \omega_{h_i}(\|\mathbf{x}-\mathbf{x}_i\|)\mathbf{p}^T(\mathbf{x})\frac{\partial\mathbf{M}(\mathbf{x})^{-1}}{\partial\mathbf{x}_{(k)}}\mathbf{p}(\mathbf{x}_i)\\ &+ \omega_{h_i}(\|\mathbf{x}-\mathbf{x}_i\|)\frac{\partial\mathbf{p}^T(\mathbf{x})}{\partial\mathbf{x}_{(k)}}\mathbf{M}(\mathbf{x})^{-1}\mathbf{p}(\mathbf{x}_i),\end{aligned}$$

and the derivative of the inverted moment matrix is given as

$$\frac{\partial(\mathbf{M}^{-1})}{\partial\mathbf{x}_{(k)}} = -\mathbf{M}^{-1}(\frac{\partial\mathbf{M}}{\partial\mathbf{x}_{(k)}})\mathbf{M}^{-1}. \tag{31}$$

Using Eq. 30, the spatial gradient can be approximated as

$$\langle\nabla f\rangle(\mathbf{x}) = \sum_i \nabla\Phi_i(\mathbf{x})f_i \tag{32}$$

and the divergence of a vector-valued function as

$$\langle\nabla\cdot\mathbf{f}\rangle(\mathbf{x}) = \sum_i \mathbf{f}_i\cdot\nabla\Phi_i(\mathbf{x}). \tag{33}$$

Higher-order derivatives are obtained by repeated differentiation using the chain rule [FM03]. The Laplace operator is for example approximated as

$$\langle\Delta f\rangle(\mathbf{x}) = \sum_i \Delta\Phi_i(\mathbf{x})f_i. \tag{34}$$

Müller et al. [MKN*04] use a different approach to compute spatial gradients in the context of stress computations for meshless elastic solid simulation. Instead of taking the derivative of the approximation, they directly approximate the first-order derivative using moving least squares and local first-order Taylor expansions. In the neighborhood of a particle $\mathbf{x}_i$, the function f can be approximated as

$$\tilde{f}(\mathbf{x}_i+\Delta\mathbf{x}) = f_i + \langle\nabla f\rangle(\mathbf{x}_i)\cdot\Delta\mathbf{x} + O(\|\Delta\mathbf{x}\|^2). \tag{35}$$

Dropping the second order term, the approximation at a neighboring particle $\mathbf{x}_j$ is becomes

$$\tilde{f}_j = \tilde{f}(\mathbf{x}_j) \approx f_i + \langle\nabla f\rangle(\mathbf{x}_i)\cdot\mathbf{x}_{ij}, \tag{36}$$

with $\mathbf{x}_{ij} = \mathbf{x}_j - \mathbf{x}_i$. The weighted least squares error of making this approximation is

$$\begin{aligned}E &= \sum_j \omega_{h_j}(\|\mathbf{x}_{ij}\|)(f_j-\tilde{f}_j)^2\\ &= \sum_j \omega_{h_j}(\|\mathbf{x}_{ij}\|)\left(f_{ij}-\langle\nabla f\rangle(\mathbf{x}_i)\cdot\mathbf{x}_{ij}\right)^2,\end{aligned}$$

with $f_{ij} = f_j - f_i$. The unknown in this expression is $\langle\nabla f\rangle(\mathbf{x}_i)$ and can be found by minimizing E. Similarly to Eq. 23, this yields the least squares approximation for ∇f at particle $\mathbf{x}_i$

$$\langle\nabla f\rangle(\mathbf{x}_i) = \mathbf{A}_i^{-1}\sum_j \omega_{h_j}(\|\mathbf{x}_{ij}\|)f_{ij}\mathbf{x}_{ij}, \tag{37}$$

with

$$\mathbf{A}_i = \sum_j \omega_{h_j}(\|\mathbf{x}_{ij}\|)\mathbf{x}_{ij}\mathbf{x}_{ij}^T \tag{38}$$

a 3×3 moment matrix. Similar considerations hold for inverting $\mathbf{A}_i$ as before for computing $\mathbf{M}^{-1}$.

Note that above equation only allows approximating the first-order spatial derivative at a particle $\mathbf{x}_i$ and not regular derivatives at general positions $\mathbf{x}$.

4. SPH–MLS Comparison

The SPH and MLS method are just two of many particle approximation methods [FM03]. They are however the most popular in computer graphics for physically based simulation algorithms.

The SPH method has the advantage that evaluating a function or its derivatives simply amounts to computing a weighted average of particle quantities. The MLS method is computationally more involved and requires construction and inversion of a (small) moment matrix per evaluation. The latter method however comes with consistency guarantees, while the SPH method in practice does not even have zero order consistency.

The approximated function can for both methods be written as

$$\langle f \rangle (\mathbf{x}) = \sum_i \Phi_i(\mathbf{x}) f_i, \tag{39}$$

where the shape functions for SPH are given by

$$\Phi_i^{\mathrm{SPH}}(\mathbf{x}) = V_i \omega_h(\|\mathbf{x} - \mathbf{x}_i\|) \tag{40}$$

and the shape functions for MLS are given by

$$\Phi_i^{\mathrm{MLS}}(\mathbf{x}) = \omega_{h_i}(\|\mathbf{x} - \mathbf{x}_i\|)\mathbf{p}(\mathbf{x})^T \mathbf{M}(\mathbf{x})^{-1}\mathbf{p}(\mathbf{x}_i). \tag{41}$$

Table 1 gives an overview of the different SPH and MLS approximation equations.

It is instructive to make the distinction between Eulerian and Lagrangian kernels. Eulerian kernels are kernels that remain fixed in shape but move with the particles. Hence when Eulerian kernels are used, particle neighbors change during the course of a simulation. This means that the shape functions in Eq. 39 need to be evaluated in every simulation time step. Eulerian kernels are always used for particle fluid simulation using SPH. Lagrangian kernels on the other hand change shape during the course of simulation so that particle neighbors remain constant. Such kernels are typically used for elastic solid simulation using MLS, where particle displacement differences are rather small and it is feasible to use fixed particle neighbors. In this setting, shape functions have to be evaluated only once in so-called *material* coordinates and can be re-used in subsequent time steps for function approximation. As a consequence, the computational complexity disadvantage of the MLS method is dropped and it becomes as efficient as the SPH method.

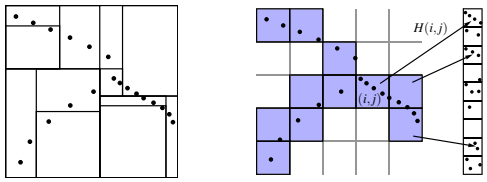


Figure 2: *(a) A balanced kd-tree. The tree adapts to the spatial distribution of the input samples. (b) A hash grid. The grid structure is virtual, only the hash table (right) is actually stored. The hash function H maps cell indices to hash table indices.*

An interesting combination of the ideas behind SPH and MLS is used in the Moving Least Squares Particle Hydrodynamics (MLSPH) method [Dil99, Dil00]. The idea is to use the MLS shape functions of Eq. 41, starting with the highest rank basis functions. If the inversion of the moment matrix however fails, then the rank of the basis is lowered until the inversion succeeds. One could imagine using MLSPH for particle fluid simulation and start with a linear basis $\mathbf{p}^1(\mathbf{x}) = [1\ x\ y\ z]$ and if needed, for example for a single isolated particle, lower the rank by using the constant basis $\mathbf{p}^0(\mathbf{x}) = [1]$ (and effectively reverting to Shepard's scattered data interpolation method). A consequence of this variable rank MLS method is that the resulting shape functions are not smooth. To our knowledge, this method has not been used in a computer graphics context.

5. Search Data Structures

Many operations in meshless function approximation rely on local neighborhood relationships between samples. If the kernels used in function approximation have local support, function values at a point $\mathbf{x}$ are only influenced by a small set of *neighbors* $\mathcal{N}_r(\mathbf{x}) = \{\mathbf{x}_i : \|\mathbf{x}_i - \mathbf{x}\| \leq r\}$, where r is the maximum distance at which the kernel is non-zero.

Finding the set $\mathcal{N}_r$ requires $\mathcal{O}(n)$ time if a brute force algorithm is used. Acceleration data structures can significantly lower the complexity of these queries. This section discusses kd-trees and spatial hashing, which are particularly well-suited for the queries most relevant to our application. Figure 2 shows an illustration. A comparison to other data structures, such as uniform grids, octrees, or bounding volume hierarchies, can be found in [BF79, Sam05].

5.1. kd-Trees

A kd-tree, or k-dimensional binary search tree, is a special case of a binary space partition tree with axis-aligned separating planes [Ben75]. Its root cell contains all points in $\mathbb{R}^k$.

	Eq.	SPH	Eq.	MLS
$\langle f \rangle(\mathbf{x})$	(5)	$\sum_i f_i \omega_h(\|\mathbf{x}_i - \mathbf{x}\|)V_i$ where $V_i = m_i/\rho_i$ and $\rho_i = \sum_j \omega_h(\|\mathbf{x}_i - \mathbf{x}_j\|)\, m_j$	(26)	$\sum_i \Phi_i(\mathbf{x}) f_i$ where $\Phi_i(\mathbf{x}) = \omega_{h_i}(\|\mathbf{x} - \mathbf{x}_i\|)\mathbf{p}(\mathbf{x})^T \mathbf{M}(\mathbf{x})^{-1}\mathbf{p}(\mathbf{x}_i)$
$\langle \nabla f \rangle(\mathbf{x})$	(9)	$\sum_i f_i \nabla\omega_h(\|\mathbf{x}_i - \mathbf{x}\|)V_i$	(32)	$\sum_i \nabla\Phi_i(\mathbf{x}) f_i$
$\langle \nabla f \rangle(\mathbf{x}_i)$	(13)	$\sum_j (f_j - f_i)\nabla\omega_h(\|\mathbf{x}_j - \mathbf{x}_i\|)V_j$ (zero for $f_i = c$)	(32)	$\sum_j \nabla\Phi_j(\mathbf{x}_i) f_j$
	(19)	$\rho_i \sum_j (\frac{f_j}{\rho_j^2} - \frac{f_i}{\rho_i^2})\nabla\omega_h(\|\mathbf{x}_j - \mathbf{x}_i\|)V_j$ (symmetric)	(37)	$\left(\sum_j \omega_{h_j}(\|\mathbf{x}_{ij}\|)\mathbf{x}_{ij}\mathbf{x}_{ij}^T\right)^{-1} \sum_j \omega_{h_j}(\|\mathbf{x}_{ij}\|) f_{ij}\mathbf{x}_{ij}$
$\langle \nabla \cdot \mathbf{f} \rangle(\mathbf{x})$	(12)	$\sum_i \mathbf{f}_i \cdot \nabla\omega_h(\|\mathbf{x}_i - \mathbf{x}\|)V_i$	(33)	$\sum_i \mathbf{f}_i \cdot \nabla\Phi_i(\mathbf{x})$
$\langle \nabla \cdot \mathbf{f} \rangle(\mathbf{x}_i)$	(14)	$\sum_j (\mathbf{f}_j - \mathbf{f}_i) \cdot \nabla\omega_h(\|\mathbf{x}_j - \mathbf{x}_i\|)V_j$ (zero for $f_j = c$)	(33)	$\sum_j \mathbf{f}_j \cdot \nabla\Phi_j(\mathbf{x}_i)$
$\langle \Delta f \rangle(\mathbf{x})$	(11)	$\sum_i f_i \Delta\omega_h(\|\mathbf{x}_i - \mathbf{x}\|)V_i$	(34)	$\sum_i \Delta\Phi_i(\mathbf{x}) f_i$
$\langle \Delta f \rangle(\mathbf{x}_i)$	(15)	$\sum_j (f_j - f_i)\Delta\omega_h(\|\mathbf{x}_j - \mathbf{x}_i\|)V_j$ (zero for $f_j = c$)	(34)	$\sum_j \Delta\Phi_j(\mathbf{x}_i) f_j$

Table 1: *A summary of the most important equations for meshless approximation using the SPH and MLS frameworks. Given discrete samples $f_i = f(\mathbf{x}_i)$ of an unknown continuous function f, the table summarizes approximations of f and differential quantities of f.*

In each level of the hierarchy, each cell is divided once by an axis aligned plane. Since cell boundaries are axis-aligned, inside/outside tests for points are fast. Figure 2 (a) shows an illustration.

Given the set of sample locations, a kd-tree can be constructed by recursively splitting cells until a minimum number of samples remain in the cell. For each cell, a good separating plane has to be found. Most commonly, the orientation of the planes is simply cycled through, and the median of point positions within the cell is used to determine the position of the separating plane. Algorithm 1 is a pseudocode version of the basic construction algorithm. More sophisticated methods based on statistical analysis of the input points have been proposed for high-dimensional data [Qui83, WAD94]. The trade-off between construction time and query time justifies these more complex construction methods only if a kd-tree, once constructed, is used for significantly more queries than there are sample points.

The construction time for a kd-tree is $\mathcal{O}(n \log n)$. Adding, deleting, or moving samples will create an unbalanced tree and degrade search performance [FP86]. Since construction is relatively cheap, the kd-tree is typically rebuilt from scratch whenever the set of points changes.

To query a kd-tree, we find the cell containing the query point $\mathbf{x}$, and then compute all cells that are intersected by the sphere of radius r centered around $\mathbf{x}$ (see Figure 3 (a)). The relevant cells are found by backtracking up the hierarchy until a cell fully contains the sphere. All children of this cell are recursively tested for intersection with the query range. Points contained in intersecting cells are candidates to be returned by the query and have to be tested against the sphere [FBF77]. A pseudocode version of the algorithm is given in Algorithm 2. [AM93] contains a more comprehensive discussion.

Algorithm 1: Recursive construction of a kd-tree. $\mathcal{S}$ is a set of sample locations, and k denotes the dimension of embedding space: $\mathbf{x} \in \mathbb{R}^k \quad \forall\, \mathbf{x} \in \mathcal{S}$. s is the maximum number of samples in a leaf node.

```
Function SplitCell(axis, S)
  if |S| ≤ s then
    return LeafNode(S)
  else
    // find good separating plane
    split = median_{x∈S} x[axis]
    // sort points into half-spaces
    S_l = {x ∈ S : x[axis] < split}
    S_r = S \ S_l
    // cycle through splitting dimensions
    newaxis = (axis + 1) mod k
    return TreeNode(SplitCell(newaxis, S_l),
                    SplitCell(newaxis, S_r),
                    split, axis)

Function MakeTree(S)
  return SplitCell(0, S)
```

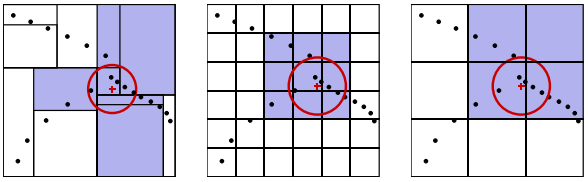
5.2. Spatial Hashing

Hash grids are regular grids whose contents are stored in a hash table. The embedding space is discretized into cells, and each cell is assigned an index. The cell indices are passed through a hash function, resulting in an index in a hash table. All data associated with the cell (such as the sample points it contains) is stored in this hash table entry. Hash

Algorithm 2: Querying a kd-tree T at $\mathbf{x}$ to find all neighboring samples $\mathcal{N}_r(\mathbf{x}) = \{\mathbf{x}_i : \|\mathbf{x}_i - \mathbf{x}\| < r\}$.

```
Function GatherSamples(T, x, r)
  if isLeaf(T) then
    return {x_i ∈ T.S : ||x − x_i|| < r}
  else
    S = ∅
    if IntersectsSphere(T.left, x, r) then
      S ← S ∪ GatherSamples(T.left, x, r)
    if IntersectsSphere(T.right, x, r) then
      S ← S ∪ GatherSamples(T.right, x, r)
    return S

Function TreeQuery(T, x, r)
  // find last node fully containing the query range
  while not isLeaf(T) do
    if ContainsSphere(T.left, x, r) then
      T = T.left
    else if ContainsSphere(T.right, x, r) then
      T = T.right
    else
      break
  // search all cells under T for sample points in range
  return GatherSamples(T, x, r)
```



(a) (b) (c)

Figure 3: *(a) Querying a kd-tree. The highlighted cells have to be searched for sample points within the query radius. (b) Query in a hash grid, where $d = r$. The 9 cells closest to the query point have to be searched (27 in three dimensions). (c) $d = 2r$. The 4 nearest cells (8 in three dimensions) need to be searched.*

grids provide fast access to stored data elements and are very easy to maintain. Since cells are not explicitly stored in memory, the searchable domain is not bounded. The memory requirement of a hash grid grows with the number of non-empty cells, but is independent of the number of empty cells. Figure 2 (b) shows an illustration.

To construct a hash grid, the search space is divided into an infinite regular grid. We will describe the construction for $\mathbb{R}^3$, but the procedure generalizes easily, and the pseudocode algorithms in this section work for arbitrary dimension.

We define a regular, axis-aligned grid with edge length d and a cell corner at the origin. For any point $\mathbf{x} = [x, y, z]$, we can compute an index tuple $\mathbf{I} = (i, j, k)$ that uniquely identifies the cell containing $\mathbf{x}$:

$$\mathbf{I}(\mathbf{x}) = (\lfloor x/d \rfloor, \lfloor z/d \rfloor, \lfloor z/d \rfloor), \tag{42}$$

We also require a hash function H which maps cell indices to entries of a hash table. A simple hash function for index tuples such as (42) has been proposed in [THM*03]:

$$H(i, j, k) = (ip_1 \textbf{ xor } jp_2 \textbf{ xor } kp_3) \textbf{ mod } s, \tag{43}$$

Where $p_{1,2,3}$ are prime numbers and s is the hash table size. In order to minimize collisions, s should also be prime or coprime to $p_{1,2,3}$. More advanced hash functions might result in fewer collisions for small hash table sizes s, however, they are typically more expensive to compute. The additional cost incurred by an occasional collision is low, and more complicated hash functions do not translate into performance gains.

Given the set of sample locations $\mathcal{S}$, points from P are sequentially added to the hash grid, and assigned to their respective hash table entries. The construction time of a hash grid for a point set of size n is $\mathcal{O}(n)$. Adding, deleting, or moving points in an existing hash grid requires only $\mathcal{O}(1)$ time. Algorithm 3 gives the construction algorithm in pseudocode.

Algorithm 3: Hash grid construction. Stores a point set $\mathcal{S} \subset \mathbb{R}^k$ in a hash table of size s.

```
Function Hash(I)
  Data: an array of prime numbers p
  H = 0
  for i ∈ {0...k−1} do
    H ← H + p[i]x[i]
  return H

Function MakeHashGrid(S, d)
  table = Array(s)
  forall x ∈ S do
    // compute cell index
    for i ∈ {0...k−1} do
      I[i] = ⌊x[i]/d⌋
    // add to hash table
    hash = Hash(I)
    table[hash].S ← table[hash].S ∪ x
  return table
```

To query the hash table, we find the hash cell (i, j, k) that contains the query point $\mathbf{x}$. All cells intersected by a sphere of radius r around $\mathbf{x}$ have to tested for points within the query range. The cell spacing d should be chosen to be either $d = r$ or $d = 2r$. In the former case, a total of 3^k cells have to be checked for points within the query radius. With $d = 2r$, only 2^k closest cells to $\mathbf{x}$ have to be considered (see Figure 3 for an illustration for $k = 2$). The average complexity of a query is linear in the number m of returned points. Algorithm 4 shows a pseudocode version of the query algorithm for $d = 2r$.

Algorithm 4: Hash grid query for a k-dimensional hash grid with spacing $d = r$. The query radius is fixed and therefore not passed as a parameter.

Function `HashGridQuery` (*table,* $\mathbf{x}$)
  // compute index and direction of neighboring cell
  for $i \in \{0 \dots k-1\}$ **do**
    $\mathbf{I}[i] = \lfloor \mathbf{x}[i]/d \rfloor$
    if $\mathbf{x}[i] - \mathbf{I}[i]d < d/2$ **then**
      $\mathbf{dI}[i] = -1$
    else
      $\mathbf{dI}[i] = 1$
  // find all points in the 2^k *closest neighboring cells*
  $\mathcal{C} = \emptyset$
  for $\mathbf{c} =$ `BinaryCount` (*0,* $2^k - 1$) **do**
    // compute modified cell index
    for $i \in \{0 \dots k-1\}$ **do**
      $\mathbf{J}[i] = \mathbf{I}[i] + \mathbf{c}[i]\mathbf{dI}[i]$
    $\mathcal{C} \leftarrow \mathcal{C} \cup table[\texttt{Hash}(\mathbf{J})].\mathcal{S}$
  return $\{\mathbf{x}_i \in \mathcal{C} : \|\mathbf{x}_i - \mathbf{x}\| < r\}$

5.3. Comparison

Both hash grids and kd-trees are commonly used for neighbor queries in meshless methods. Which data structure to use depends on several factors, and a final decision can often only be made after performance figures can be compared for representative test cases. There are, however, some general rules that help determine which method is most appropriate.

Most importantly, kd-trees are fully adaptive data structures, while hash grids discretize space uniformly. In situations with strongly varying sampling density, kd-trees offer advantages over hash grids since the spatial resolution adapts to the local spatial density of the point set. However, the adaptivity comes at a cost. Constructing a kd-tree takes $\mathcal{O}(n \log n)$ time as opposed to $\mathcal{O}(n)$ for hash grids. While construction cost is not an issue for static data, it can become a limiting factor in dynamic settings. Hash grids are most efficient with range queries of known constant radius. If the sampling density does not vary significantly throughout the domain, this leads to a retrieval in $O(1)$.

II. Meshless Simulation and Modeling

We now have all basic ingredients to implement various physical systems. We will treat fluid simulation using the SPH framework described in Section 2, as well as simulation of continuum elasticity using MLS interpolations as discussed in Section 3. We will see in Section 8 how the latter can be used for shape and motion modeling.

6. Fluid Simulation using SPH

The behavior of fluids is governed by the Navier-Stokes equations. In graphics, the incompressible version of these equations is most commonly used. Traditionally, it is discretized on either regular, adaptive, or tetrahedral meshes for both water and smoke simulations [Sta99, LGF04, FOK05, KFCO06].

An alternative to these Eulerian approaches is to discretize the material instead of its embedding space. Using a tetrahedral or hexahedral grid, this is the standard method for simulations involving continuum elasticity (see also Section 7). However, fluid simulations in computer graphics usually involve very turbulent behavior, as well as complex topological changes such as the formation of water droplets and spray.

The simplicity of restructuring in meshless discretization makes meshless Lagrangian approaches a viable alternative. We will use the SPH framework introduced in Section 2 to discretize the Navier-Stokes equations in a Lagrangian fashion. In this context, it is useful to think of the sample points as *particles* that carry the mass and other properties of the fluid, such as pressure and velocity. In principle, the simulation will then proceed by first computing forces acting on the particles using a discretization of the governing partial differential equations using the current particle positions. When all forces have been computed, they are integrated, and the particle velocities are updated accordingly. The particles are then moved according to their updated velocities.

The governing equations for fluids are the Navier-Stokes equations. They state the laws of momentum and mass conservation. The continuity equation, stating the conservation of mass, is not needed in the Lagrangian framework, since the mass is carried by the particles, and is hence preserved automatically if no particles are deleted or inserted, and the particle masses are not changed. For compressible fluids, the momentum equation can be written as a combination of pressure, viscosity, and external forces:

$$\frac{D\mathbf{v}}{Dt} = \frac{1}{\rho}\left(\mathbf{f}_\mathrm{p} + \mathbf{f}_\mathrm{v} + \mathbf{f}_\mathrm{e}\right), \tag{44}$$

Above, $\mathbf{v}$ denotes the velocity field, and $\mathbf{f}_\mathrm{e}$ are external forces acting on the fluid. The pressure force $\mathbf{f}_p$ is a function of the pressure field p

$$\mathbf{f}_\mathrm{p} = -\nabla p, \tag{45}$$

In computer graphics, viscosity effects due to compression are usually neglected, and the viscosity force (even for the compressible case that is assumed in SPH simulation) is typically modeled after the viscosity term that applies to incompressible fluids [MCG03]. Viscous forces smooth the velocity field, and are proportional to the Laplacian of the velocity field:

$$\mathbf{f}_\mathrm{v} = \mu\nabla \cdot \nabla\mathbf{v} = \mu\Delta\mathbf{v}. \tag{46}$$

Intuitively, Eq. 44 states that forces act to even out pressure differences ($\mathbf{f}_\mathrm{p}$) and velocity differences ($\mathbf{f}_\mathrm{v}$) within a fluid. The pressure p is a function of the density of the fluid.

A common choice for the pressure function is the Tait equation [Mon94]

$$p = K(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1). \tag{47}$$

ρ_0 denotes the rest density of the fluid. The parameters K and γ determine the compressibility of the fluid. Monaghan proposed $\gamma = 7$, whereas in computer graphics, a value of $\gamma = 1$ is typically used [DG96, MCG03]. Recently [BT07] propose a method for choosing sensible parameter values in (47). Low values of γ and K make the fluid more compressible, but allow for larger time steps.

$\frac{D\mathbf{v}}{Dt} = \frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v}$ is the *material derivative* of $\mathbf{v}$, i. e. the derivative of $\mathbf{v}$ at a point moving with the material. In a Lagrangian setting, the particles move with the material they represent. Thus, the material derivative can simply be computed as the time derivative of a value stored with a given particle.

Eq. 44 is a partial differential equation in the velocity field $\mathbf{v}$, and uses the pressure field p. We therefore discretize these fields using SPH, using the same sample points for both: We will have one value $\mathbf{v}_i$ and p_i per particle. Although it is beyond the scope of this tutorial, it is very easy to simulate inhomogeneous media using SPH. The varying property can simply be stored at each particle, and will automatically be advected with the fluid.

The most straightforward spatial discretization of (44) using SPH is

$$\frac{\partial \mathbf{v}_i}{\partial t} = -\frac{\langle \nabla p \rangle (\mathbf{x}_i)}{\rho_i} + \mu \langle \Delta \mathbf{v} \rangle (\mathbf{x}_i) + \frac{\mathbf{f}_e(\mathbf{x}_i)}{\rho_i}. \tag{48}$$

Below, we will discuss the pressure and viscosity terms of (48) in more detail, before we discuss the algorithmic details of the simulation.

6.1. Pressure Forces

In Eq. 48, we approximate the particle accelerations due to pressure forces as

$$\frac{\mathbf{f}_p(\mathbf{x}_i)}{\rho(\mathbf{x}_i)} = -\frac{\langle \nabla p \rangle (\mathbf{x}_i)}{\rho_i}. \tag{49}$$

An implementation of (49) using the gradient discretization (9) yields a working simulation. The explicit discrete expression is

$$\frac{\mathbf{f}_p(\mathbf{x}_i)}{\rho(\mathbf{x}_i)} = -\sum_j \nabla \omega_h^{ij} p_j \frac{m_j}{\rho_j \rho_i} = -\sum_j \mathbf{a}_p^{ji} \tag{50}$$

where we use the convenient shorthand $\omega_h^{ij} = \omega_h(\|\mathbf{x}_i - \mathbf{x}_j\|)$. It is useful to decompose the pressure interactions into pairwise accelerations $\mathbf{a}_{ij}$. We can see that $\mathbf{a}_p^{ji} + \mathbf{a}_p^{ij} \neq 0$. In other words, there is no guarantee that the pairwise influences cancel out. Therefore, it can not be guaranteed that linear and angular momentum are conserved. Especially for low-resolution simulations that are common in computer graphics, the lack of symmetry in the pairwise influences can lead to problematic visual artifacts.

A simple way to solve this problem is to symmetrize the pairwise interactions, for example by using $\mathbf{a}_p^{ji} = \nabla \omega_h^{ij} \frac{(p_i + p_j) m_j}{2 \rho_i \rho_j}$. A more elegant way of ensuring momentum conservation is using the alternative gradient approximation (19), which yields

$$\frac{\mathbf{f}_p(\mathbf{x}_i)}{\rho(\mathbf{x}_i)} = -\sum_j \nabla \omega_h^{ij} \left(\frac{p_j}{\rho_j^2} + \frac{p_i}{\rho_i^2}\right) m_j. \tag{51}$$

It is easy to check that this expression is indeed symmetric as defined above, implying that all pressure forces add up to zero (therefore conserving linear momentum). Because $\nabla \omega_h^{ij}$ is always parallel to $(\mathbf{x}_i - \mathbf{x}_j)$, no pairwise interaction introduces any angular momentum, and angular momentum is preserved as well.

6.2. Viscosity Forces

Discretizing the acceleration on particles due to viscosity leads to the expression

$$\frac{\mathbf{f}_v(\mathbf{x}_i)}{\rho_i} = \mu \sum_j \Delta \omega_h^{ij} (\mathbf{v}_j - \mathbf{v}_i) \frac{m_j}{\rho_i \rho_j} = \mu \sum_j \mathbf{a}_v^{ji}. \tag{52}$$

The force (52) is symmetric: $\mathbf{a}_v^{ji} + \mathbf{a}_v^{ij} = 0$.

There are several problems with this expression. The SPH approximation of higher-order derivatives depends strongly on the distribution of sample points and is therefore quite noisy, especially if the support radius of the kernel function is small and only few samples are considered. While choosing a larger support radius h can alleviate this problem, large values of h also introduce significant numerical diffusion on their own, which might be undesirable.

Numerical viscosity in SPH simulations is usually so high that additional viscosity is rarely necessary from an animation point of view. However, viscosity forces smooth the velocity field, and can greatly contribute to the stability of the simulation, allowing for larger timesteps. Creating a stable simulation with the viscosity (52) requires some parameter tuning, as high values of μ destabilize the simulation and can actually reduce that maximum admissible timestep.

Viscosity forces effectively perform a smoothing of the velocity field. We can use more stable smoothing operators to achieve a similar effect. Such methods are called *artificial viscosity*. One type of artificial viscosity is the XSPH technique [Mon92]. Here, after each time step, the velocity of each particle is modified in the direction of the average

velocity of its neighbors.

$$\begin{aligned}\tilde{\mathbf{v}}_i &= \xi \langle \mathbf{v} \rangle (\mathbf{x}_i) + (1-\xi)\mathbf{v}_i \\ &= \xi \left[\sum_j \omega_h^{ij} \mathbf{v}_j \frac{m_j}{\rho_j} \right] + (1-\xi)\mathbf{v}_i. \end{aligned} \tag{53}$$

Original XSPH uses the corrected velocities $\tilde{\mathbf{v}}_i$ only for moving the particles, but stores the originally computed velocities. If $\tilde{\mathbf{v}}_i$ is also stored and used in subsequent timesteps, the viscosity effect is stronger. In (53), $0 \leq \xi \leq 1$ determines how strong artificial viscosity should be. Even high values of ξ do not incur stability problems, on the contrary, stability increases as ξ gets closer to 1.

6.3. Boundary Effects

If the particle distribution is highly irregular, the SPH approximations of differential quantities can become unstable. In a fluid simulation, the forces push particles into semi-regular arrangements, such that these problems usually only occur on the boundary of the fluid. Figure 4 shows the effects of a free surface on the particle distribution. Near the surface, less particles are inside the support radius of the kernel function, leading to lower density estimates. The pressure forces try to maintain a constant density throughout the fluid and push particles on the surface closer together. By itself, this effect is not harmful, on the contrary, it can help extracting a smooth surface (see Section 6.5).

For simulations involving free boundaries, the pressure can be clamped to positive values, or reduced by a factor ζ for negative values:

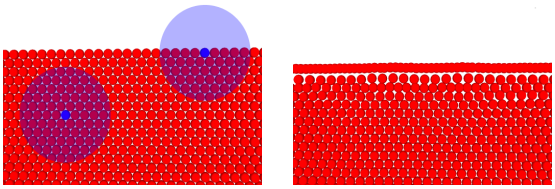
$$p' = \begin{cases} p & p \geq 0, \\ \zeta p & p < 0. \end{cases} \tag{54}$$

Clamping or reducing the pressure at the boundaries reduces the cohesive forces inside the fluid and makes splashes and spray more likely. Müller et al. [MCG03] have proposed to explicitly include surface tension forces, which have an effect similar to using low values of ζ.

Irregular sampling at the boundary can lead to problems for example for higher order derivative approximations needed for heat transfer computations. We can avoid irregular sampling and the associated issues using ghost particles. For free surfaces (fluid/air or fluid/vacuum interfaces), ghost particles have to be generated dynamically. This amounts to implementing a simulation of multiphase flow [MSKG05]. Rigid objects in contact with the fluid can be sampled statically. To implement interaction with a rigid body, the rigid body is sampled with ghost particles at positions $\mathbf{x}_k^g$, which are treated like regular fluid particles regarding density computation [MTHG04]. The density approximation then becomes

$$\rho_i = \sum_j m_j \omega_h^{ij} + \sum_k m_k^g \omega_h(\|\mathbf{x}_i - \mathbf{x}_k^g\|), \tag{55}$$

All other computations can proceed as before. If no-slip



(a) (b)

Figure 4: *Particle distribution close to the surface of the fluid. (a) The kernel function centered around a particle "sees" only about half the number of particles that are in range of a particle inside the fluid. (b) During the simulation, the particles close to the surface are pushed closer together to maintain constant density.*

(sticky) boundaries are desired, the ghost particles are assigned the velocity of the rigid body, and are included computation of the viscosity force

$$\begin{aligned}\frac{\mathbf{f}_v(\mathbf{x}_i)}{\rho_i} &= \mu \sum_j \Delta\omega_h^{ij}(\mathbf{v}_j - \mathbf{v}_i)\frac{m_j}{\rho_j} + \\ &\quad \mu \sum_k \Delta\omega_h(\|\mathbf{x}_i - \mathbf{x}_k^g\|)(\mathbf{v}_k^g - \mathbf{v}_i)\frac{m_k}{\rho_k}, \end{aligned} \tag{56}$$

or, for artificial viscosity

$$\begin{aligned}\tilde{\mathbf{v}}_i &= (1-\xi)\mathbf{v}_i + \\ &\quad \xi \left[\sum_j \omega_h^{ij} \mathbf{v}_j \frac{m_j}{\rho_j} + \sum_k \omega_h(\|\mathbf{x}_i - \mathbf{x}_k^g\|)\mathbf{v}_k^g \frac{m_k}{\rho_k} \right]. \end{aligned} \tag{57}$$

For two-way coupling, forces are computed for the ghost particles, however, they are not moved independently. Instead, forces acting on ghost particles are added to the corresponding rigid body, which is updated, moving all its particles and updating their velocities.

6.4. Time Discretization and Simulation Loop

SPH helps us discretize the Navier-Stokes equations in space, and leaves us with an ordinary differential equation for each particle which we can discretize in time. Each particle stores a position and a velocity. Eq. 44 tells us how the particle velocities change over time. Using a simple explicit integration scheme, we can integrate those changes over time

$$\mathbf{v}_i(t+\Delta t) = \mathbf{v}_i(t) + \Delta t \frac{D\mathbf{v}_i}{Dt} \tag{58}$$

$$\mathbf{x}_i(t+\Delta t) = \mathbf{x}_i(t) + \Delta t \mathbf{v}_i(t+\Delta t) \tag{59}$$

Whether or not rigid bodies are sampled with ghost particles, it is a good idea to prevent the particles from entering the solid by enforcing hard constraints. When moving particles according to (59), we can reflect them off solid boundaries, thus preventing them from entering solid objects.

Algorithm 5 gives the complete SPH fluid simulation algorithm using artificial viscosity. Boundary effects on the surface are ignored. In a first pass over the particles, we compute density estimates for all particles. Then, forces are computed. Finally, we integrate the forces and move the particles. The parameters passed to the simulation are the timestep Δt, the kernel radius h, artificial viscosity parameter ξ, stiffness (inverse compressibility) K, pressure exponent γ, rest density ρ_0, as well as pressure scaling for negative pressures ζ. It is important that the initial starting distribution is not too far from a rest state, i. e. , the pressures computed in the first step should be close to zero.

Algorithm 5: The SPH simulation loop. The kernels $\omega_h(r)$ are non-zero only for $r < h$. We use a hash grid for neighborhood computations.

Data:
$\mathbf{x}_i$ – particle positions
$\mathbf{v}_i$ – particle velocities
m_i – particle masses

Function `SPHStep` (Δt, h, ξ, K, γ, ρ_0, ζ)
// compute neighborhoods and density
H = `MakeHashGrid` ($\{x_i\}$, h)
forall i **do**
// compute neighborhood indices
$\mathcal{N}_i = \{ j : x_j \in$ `HashGridQuery` (H, $\mathbf{x}_i$, h) $\}$
// compute density
$\rho_i = \sum_{j \in \mathcal{N}_i} m_j \omega_h(\|\mathbf{x}_i - \mathbf{x}_j\|)$
// compute pressure
$p_i = K \left(\frac{\rho_i}{\rho_0} - 1 \right)^{\gamma}$
if $p_i < 0$ **then**
$p_i \leftarrow \zeta p_i$
// compute forces
forall i **do**
// acceleration due to pressure forces
$\mathbf{a}_i = -\sum_{j \in \mathcal{N}_i} \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|} \omega_h'(\|\mathbf{x}_i - \mathbf{x}_j\|)$
// average velocity for artificial viscosity
$\bar{\mathbf{v}}_i = \sum_{j \in \mathcal{N}_i} \mathbf{v}_j \frac{m_j}{\rho_j} \omega_h(\|\mathbf{x}_i - \mathbf{x}_j\|)$
// artificial viscosity and integration
forall i **do**
// artificial viscosity
$\mathbf{v}_i \leftarrow (1 - \xi)\mathbf{v}_i + \xi \bar{\mathbf{v}}_i$
// integrate velocity
$\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t \mathbf{a}_i$
// integrate positions
$\mathbf{x}_i \leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$
`EnforceConstraints` (i)

6.5. Surface Extraction

The SPH fluid simulation yields particle positions, as well as discretely sampled density and velocity fields. In Eulerian fluid simulations, a surface or smoke densities are advected using the computed velocity field. For SPH simulations, visualization is typically either based directly on the particle positions or the density field.

For gaseous phenomena such as clouds or smoke, particles are often rendered as semi-transparent spheres with a volumetric texture. The texture of the spheres can be chosen depending on density, temperature, or any other value from the underlying simulation (see for example [FOA03, HBSL03]).

For liquids, the interface of the fluid with the surrounding air or vacuum needs to be computed. The easiest and most common way to generate a surface around the particles is to extract an implicit surface from the density field. [Bli82]. Several variants to this approach have been proposed [ZB05, APKG07]. This isosurface can be rendered directly, for example using raytracing, or extracted using the Marching Cubes algorithm [LC87] or a variant thereof. The resulting triangle mesh can be rendered with standard rendering algorithms, in the case of transparent liquids, raytracing is the preferred solution for high-quality images, while hardware rendering can be used for simpler settings. Figure 5 shows some frames from an SPH fluid simulation result.

6.6. Comparison

A meshless fluid simulation has both advantages and disadvantages over the traditional mesh-based Eulerian approaches.

In general, boundary conditions are easier to enforce in particle-based methods than in Eulerian simulations. Boundary conditions in Eulerian simulation are easiest to implement if they are aligned with the simulation grid used to represent the domain. For performance reasons, these meshes are usually regular grids, thus leading to artifacts when representing boundaries. [FOK05] solve the problem by using adaptive meshes to discretize the simulation domain. [GSLF05] developed a method that couples a Eulerian fluid simulation to thin shells. The same technique could also be used to represent boundary conditions.

In Lagrangian methods, the material itself is discretized, and boundary conditions can be imposed without restrictions. In particular, in particle-based simulations, the boundary condition can be applied to individual particles. For two-way interaction between fluids and other objects, the forces or impulses used to enforce the boundary conditions on the particles can be applied to the boundary [MTHG04].

Another practical difficulty in Eulerian fluid simulations is the surface representation. Usually, the interface is tracked in the velocity field by integration. However, due to integration errors, the total volume of the fluid can change. In practice, this leads to mass loss, especially in thin sheets that cannot be resolved by the simulation grid. In contrast, in an

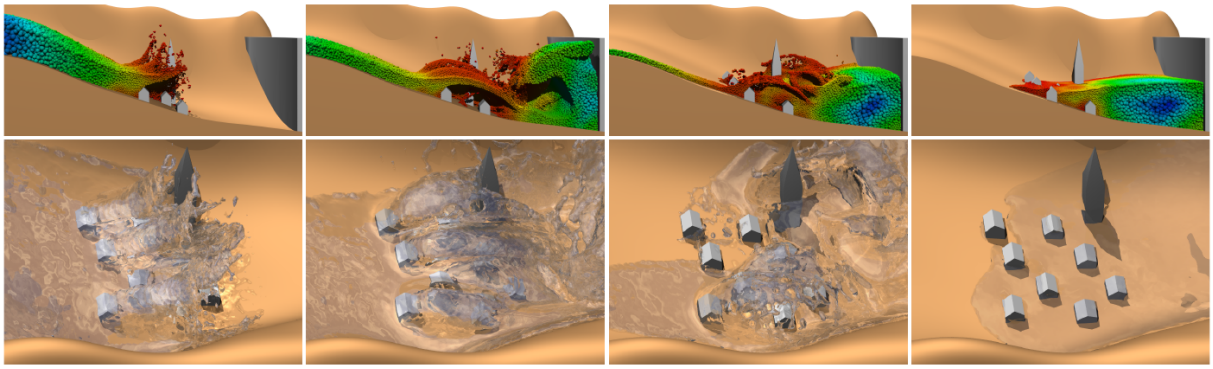


Figure 5: *The bottom row shows four frames of a fluid simulation obtained using the SPH algorithms discussed in this section. The top row shows the corresponding particles in a cross section view. This result is taken from [APKG07].*

SPH simulation, mass is carried by the particles, thus mass-preservation is guarantueed.

The flexibility that meshless discretizations offer makes them particularly well-suited for adaptive resampling. Adaptive discretization is beyond the scope of this tutorial and we refer the interested reader to [APKG07, KAG*06] for details.

A major disadvantage of SPH simulation especially for fluids like water is the inherent compressibility of the resulting material. In Eulerian simulations, it is relatively easy to enforce incompressibility by solving a global linear system. An analogous method exists for SPH simulation [KO96, PTB*03, CEL06]. However, due to the inconsistency of SPH operators, enforcing true incompressibility is hard for SPH methods. In practice, methods that use an auxiliary grid to solve for a divergence-free velocity field [Har64, BR86, ZB05] are more common.

7. Simulating Elastic Materials using MLS Approximations

The previous section discussed how the particle approximation method SPH can be used for the simulation of fluids. In this section we will discuss how the MLS method can be used for animation of elastically deformable objects. The basis of the material in this section was first presented in the graphics community by Müller and co-workers [MKN*04], building on recent trends in mechanical engineering [BKO*96].

Similar to the particle fluid discretization, we will sample the object's volume by particles and use a meshless continuum mechanics based model and the discussed MLS method to derive the elastic forces. The absence of explicit connectivity information entails similar advantages as discussed before. Particle resolutions can be easily adapted (e.g., coarsened or refined) during the simulation and therefore large deformations or even fracturing materials are easier to handle. On the downside, meshless methods are computationally more involved, since the particle connectivity is determined at run-time. However, for moderate elastic deformations, the computational burden can be reduced significantly as will be discussed in this section. Also, as will be shown, local caching schemes can be used in the case of fracturing materials.

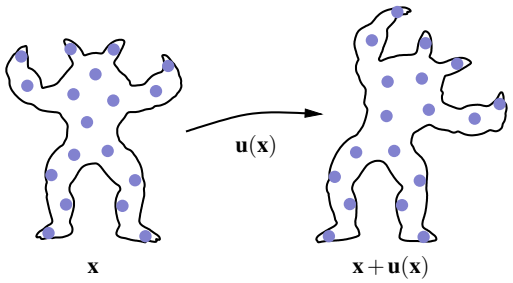


Figure 6: *Left: undeformed object in material coordinates* $\mathbf{x}$*. Right: deformed object in world coordinates* $\mathbf{x}' = \mathbf{x} + \mathbf{u}(\mathbf{x})$*. The displacement field* $\mathbf{u}(\mathbf{x})$ *defines a continuous mapping from material coordinates to world coordinates.*

We will first discuss the elasticity theory and the particle discretization of the governing equations as well as a simple explicit time stepping algorithm for the simulation of elastic solids in Section 7.1. Next, we will discuss two variants for surface deformation in a computer graphics context in Section 7.2. Finally, we discuss in Section 7.3 how plasticity and fracturing effects can be included without much effort.

7.1. Elastic Particle Based Solid Simulation

We start by discussing the continuum elasticity model in the continuous setting and derive how it can be adapted in the discrete setting to allow numerical point-based simulations.

Elasticity Model The continuum elasticity equations de-

scribe how to compute elastic stresses (and thus forces) inside an object given a deformation field. Assume that the undeformed object is defined by the *material* coordinates $\mathbf{x} = (x,y,z)^T$ and that the displacement field is defined as a continuous mapping $\mathbf{u}(\mathbf{x}) = (u,v,w)^T : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ which defines for each point $\mathbf{x}$ in material coordinates to which point $\mathbf{x}' = \mathbf{x} + \mathbf{u}(\mathbf{x})$ in *spatial* or *world* coordinates it is displaced (see Figure 6). Note that each of the scalar displacements $u = u(x,y,z), v = v(x,y,z)$ and $w = w(x,y,z)$ are function of the material coordinates. The material coordinate system is also often referred to as the reference system. The world coordinate system is often denoted as the displaced or deformed coordinate system.

The elastic strain ε is computed from the spatial derivatives of this displacement field $\mathbf{u}(\mathbf{x})$. In three dimensions the gradient of the displacement field is a 3×3 matrix:

$$\nabla \mathbf{u} = \begin{bmatrix} u_{,x} & u_{,y} & u_{,z} \\ v_{,x} & v_{,y} & v_{,z} \\ w_{,x} & w_{,y} & w_{,z} \end{bmatrix}, \tag{60}$$

where the index after the comma represents a spatial derivative.

A popular choice in computer graphics is to use Green-Saint-Venant's nonlinear strain tensor:

$$\varepsilon = \frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^T + \nabla \mathbf{u}^T \nabla \mathbf{u}), \tag{61}$$

which is a symmetric 3×3 matrix (i.e., $\varepsilon^T = \varepsilon$).

To come to stress σ, one can simply use Hooke's law which states that stress and strain are linearly related:

$$\sigma = \mathbf{E}\varepsilon, \tag{62}$$

where $\mathbf{E}$ is a $3 \times 3 \times 3 \times 3$ rank four tensor. For isotropic materials, the coefficients of this tensor only depend on Young's Modulus E and Poisson's Ratio ν [NMK*05].

The elastic body forces can then be computed via the strain energy density [MKN*04]:

$$U = \frac{1}{2}\varepsilon \cdot \sigma = \frac{1}{2}\sum_{i=1}^{3}\sum_{j=1}^{3} \varepsilon_{ij}\sigma_{ij}. \tag{63}$$

The elastic force per unit volume is then computed as the negative gradient of the strain energy density with respect to the displacement field $\mathbf{u}$ (computed as the directional derivative):

$$\mathbf{f}^{\text{elastic}} = -\nabla_{\mathbf{u}} U. \tag{64}$$

Volume conservation forces are added to avoid undesirable shape inversions

$$\mathbf{f}^{\text{volume}} = -\frac{k_v}{2}\nabla_{\mathbf{u}}(|\mathbf{I} + \nabla \mathbf{u}(\mathbf{x})| - 1)^2, \tag{65}$$

with k_v a user-defined constant.

By applying Newton's second law of motion, we come to the partial differential equation (PDE) governing dynamic elastic materials:

$$\rho \frac{\partial^2 \mathbf{x}'}{\partial t^2} = \rho \frac{\partial^2 \mathbf{u}}{\partial t^2} = \mathbf{f}^{\text{elastic}} + \mathbf{f}^{\text{volume}} + \mathbf{f}^{\text{body}}, \tag{66}$$

where ρ is the material density and $\mathbf{f}^{\text{body}}$ are external forces such as gravity or collision forces.

Meshless Discretization To discretize the force distribution, the displacement field is approximated using the MLS approximation scheme discussed in Section 3 as

$$\mathbf{u}(\mathbf{x}) = \sum_i \Phi_i(\mathbf{x})\mathbf{u}_i, \tag{67}$$

where $\mathbf{u}_i$ are the displacement vectors at a discrete set of particles and Φ_i are shape functions associated with these particles. In the work of [MKN*04], the particles are sampled uniformly in space over the volume of the object. However, the lack of a consistent mesh and connectivity information also facilitates adaptive sampling as will be discussed below for plastic and fracturing materials.

Discretization of the above PDE is straightforward and MLS approximations are only necessary for the computation of the gradient of the displacement field in the strain equation. Müller et al. [MKN*04] use the first order approximation as given by Eq. 37, however, one can equally well use Eq. 32.

As discussed before, for moderate elastic deformations the computation of the shape functions can be performed only once for each particle in the reference system. This corresponds to using Lagrangian kernels and the shape functions are then a function of the material coordinates, and not a function of the displaced or world coordinates. Thus, in each animation time step, the gradient of the displacement field for a particle can be simply computed by iterating over its neighbors and evaluating the appropriate linear combination.

Simulation Loop The elastic animation framework proceeds in each time step as follows:

$$\underbrace{\mathbf{u}_t}_{\texttt{displacements}} \rightarrow \underbrace{\nabla \mathbf{u}_t}_{\texttt{derivatives}} \rightarrow \underbrace{\varepsilon_t}_{\texttt{strains}} \rightarrow \underbrace{\sigma_t}_{\texttt{stresses}} \rightarrow \underbrace{\mathbf{f}_t}_{\texttt{forces}} \rightarrow \underbrace{\mathbf{u}_{t+\Delta t}}_{\texttt{integration}}$$

with Δt the time step interval. First, the derivatives of the displacements are computed using the MLS approximation method. This yields elastic strains (Eq. 61) which in turn yield elastic stresses according to Hooke's law (Eq. 62). From the stresses, one can compute the elastic body forces acting on the particles using Eq. 64 which are then used to integrate the particle positions forward in time using Eq. 66. These new particle positions then define the shape of the deformed object at the beginning of the next time step.

Time integration can be performed using explicit

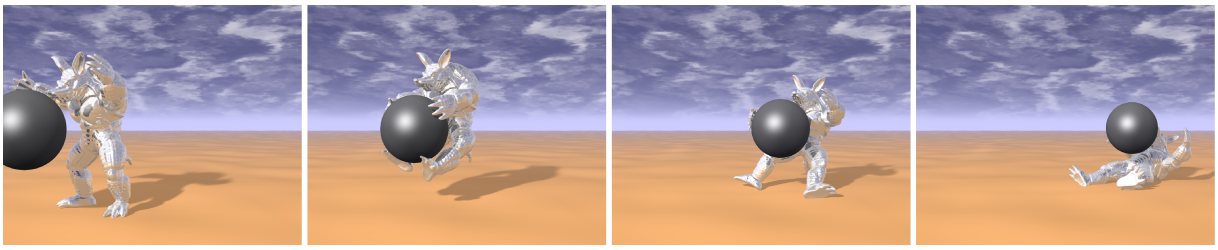


Figure 7: *Meshless simulation of an elastically deforming armadillo model.*

schemes, if small time steps are taken. However, Müller et al. also provide the tangent stiffness matrix derived from the elastic forces for implicit integration [MKN*04].

7.2. Surface Animation

The deformation field defined by the particles can be evaluated using the MLS approximation at any point in the particles' support region. Hence, any explicit surface representation that surrounds the particles can be used. In the following we will limit the discussion to polygon meshes, but the algorithms extend trivially to other surface representations such as point set surfaces. Note that there is no tight coupling between the surface and the deformation field discretization which is important in a computer graphics context where typically one requires a very fine surface resolution, while a coarse deformation field representation is often sufficient to obtain the desired deformable solid behavior.

Deforming the shape's surface can be done in a straightforward and very efficient manner. We discuss two alternatives that were proposed in the literature.

Alternative 1 The first approach [AOW*08] directly uses the MLS approximation of the displacement field. In a preprocessing step, one has to compute for each vertex $\mathbf{x}$ the set of particles that have non-zero support at the vertex. Given these particles, the shape functions $\Phi_i(\mathbf{x})$ and the gradient of the shape functions $\nabla\Phi_i(\mathbf{x})$ are computed using Eq. 27 and Eq. 32 respectively. This computation is only done once in material coordinates. During simulation, the deformed vertex position $\mathbf{x}'$ is computed using Eq. 67 as $\mathbf{x}' = \mathbf{x} + \mathbf{u}(\mathbf{x})$. Note that this simply amounts to computing a linear combination of the neighboring particles' deformation vectors using the precomputed shape functions.

Similarly, the updated (unnormalized) vertex normal $\mathbf{n}'(\mathbf{x})$ can be approximated from the local gradient of the deformation field as $\mathbf{n}'(\mathbf{x}) = \mathbf{n}(\mathbf{x}) + \nabla\mathbf{u}(\mathbf{x})\mathbf{n}(\mathbf{x})$. Here, a computational trick reduces the matrix-vector multiplications to scalar-vector multiplications by noting that this expression is equivalent to $\mathbf{n}'(\mathbf{x}) = \mathbf{n}(\mathbf{x}) + \sum_i (\nabla\Phi_i^T(\mathbf{x})\mathbf{n}(\mathbf{x}))\mathbf{u}_i$, where the scalars $\nabla\Phi_i^T(\mathbf{x})\mathbf{n}(\mathbf{x})$ are constant and can be precomputed. Again, this amounts to adding to the undeformed normal a weighted sum of the displacement vectors $\mathbf{u}_i$ where the weights are the precomputed $\nabla\Phi_i^T(\mathbf{x})\mathbf{n}(\mathbf{x})$.

Computing the updated vertex position and normal can be efficiently performed on the GPU as follows. For each vertex $\mathbf{x}$ the indices to the particle neighbors and the accompanying scalars $\Phi_i(\mathbf{x})$ and $\nabla\Phi_i^T(\mathbf{x})\mathbf{n}(\mathbf{x})$ are stored in GPU texture memory. During animation, the computed particle displacement vectors $\mathbf{u}_i$ are sent to the graphics board, which is several orders of magnitude smaller than the number of vertices. Using multiple render passes and fragment shaders one can easily compute and write for each vertex its deformation for the position and normal to intermediate texture memory. Then, in a final render pass, the vertex position and normal are computed in a vertex shader using two final texture lookups to retrieve the respective information.

Alternative 2 Instead of using the MLS approximation at the vertices, Müller et al. [MKN*04] propose a different surface animation approach that is less accurate, but avoids the storage of per-vertex shape functions. They reuse the approximation $\nabla\mathbf{u}$ computed at the particles and derive a first order accurate approximation for the displacement vectors at the surface points:

$$\tilde{\mathbf{u}}(\mathbf{x}) = \sum_j \bar{\omega}_{ij} \left(\mathbf{u}_j + \nabla\mathbf{u}(\mathbf{x}_j)^T (\mathbf{x} - \mathbf{x}_j) \right), \qquad (68)$$

where $\bar{\omega}_{ij} = \omega_{ij} / \sum_j \omega_{ij}$ is a normalized compactly supported weighting function evaluated using the distance between the vertex and the respective particle. This alternative surface deformation algorithm avoids the per-vertex storage of precomputed shape functions, but implementing this deformation algorithm on the GPU would require sending more information (the vectors $\mathbf{u}_j$ and matrices $\nabla\mathbf{u}(\mathbf{x}_j)$) to the graphics board. Evaluation also involves more computations as compared to Alternative 1.

Figure 7 illustrates the meshless elasticity animation algorithm.

7.3. Extension for Plasticity and Fracturing

The basic elastic simulation framework is extended in [MKN*04] to include plastic deformations and in [PKA*05] to incorporate fracturing for graphical applications. We first

discuss the extension for plasticity using plastic strain state variables and then discuss how discontinuities in the deformation field are modeled to enable fracture simulation. Finally, adaptive sampling for large plastic deformations and near propagating cracks ensures an adequate sampling density at all times.

Plasticity Plasticity effects are easily modeled by using strain state variables $\varepsilon_i^{\text{plastic}}$ stored per particle as proposed by [OBH02] and [MKN*04]. The actual strain used for computing the elastic forces is then

$$\varepsilon_i^{\text{elastic}} = \varepsilon_i - \varepsilon_i^{\text{plastic}}, \tag{69}$$

where ε_i is computed using Eq. 61.

At every time step, the plastic strain is updated using following procedure [GP07]

$$\begin{aligned}
\varepsilon_i^{\text{elastic}} &\leftarrow \varepsilon_i - \varepsilon_i^{\text{plastic}} \\
\text{if } \|\varepsilon_i^{\text{elastic}}\| > c_{\text{yield}} \text{ then } \varepsilon_i^{\text{plastic}} &\leftarrow \varepsilon_i^{\text{plastic}} + c_{\text{creep}} \cdot \varepsilon_i^{\text{elastic}} \\
\text{if } \|\varepsilon_i^{\text{plastic}}\| > c_{\text{max}} \text{ then } \varepsilon_i^{\text{plastic}} &\leftarrow \varepsilon_i^{\text{plastic}} \cdot c_{\text{max}} / \|\varepsilon_i^{\text{plastic}}\|
\end{aligned}$$

A small value for the parameter c_{creep} gives only little plastic behavior, while a value of 1 immediately absorbs all elastic strain. The parameter c_{max} defines the maximum plastic strain a particle can store.

For large deformations, the reference shape should be updated after each time step while storing the plastic strain state variables:

$$\begin{aligned}
\varepsilon_i^{\text{plastic}} &\leftarrow \varepsilon_i^{\text{plastic}} - \varepsilon_i, \\
\mathbf{x}_i &\leftarrow \mathbf{x}_i + \mathbf{u}_i, \\
\mathbf{u}_i &\leftarrow \mathbf{0}.
\end{aligned}$$

After doing so, one also has to recompute the shape functions for all the particles. This can make simulating plastic materials expensive, especially when a large number of particles and/or when high order basis functions are used. Note also that highly plastic deformations can result in poor and degenerate particle sampling distributions. Below we will discuss an adaptive sampling strategy that alleviates this problem. An example of plastic deformations is shown in Figure 8.

Modeling Discontinuities As is clear from the shape function definition Eq. 27, neighboring particles always interact, as long as they are in each other's support. Although this is a reasonable assumption when animating elastic materials, this approach is clearly insufficient when animating fracturing materials. Indeed, when a crack propagates through the material, it should cut the interaction between the particles on opposite sides of the crack surfaces. To enable this separation, Belytschko et al. [BLG94] introduced a visibility criterion, where particles can only interact with each other, if the ray connecting the two particle centers does not intersect a boundary surface. The adaptation can be easily made by modifying the weight function ω to incorporate this visibility

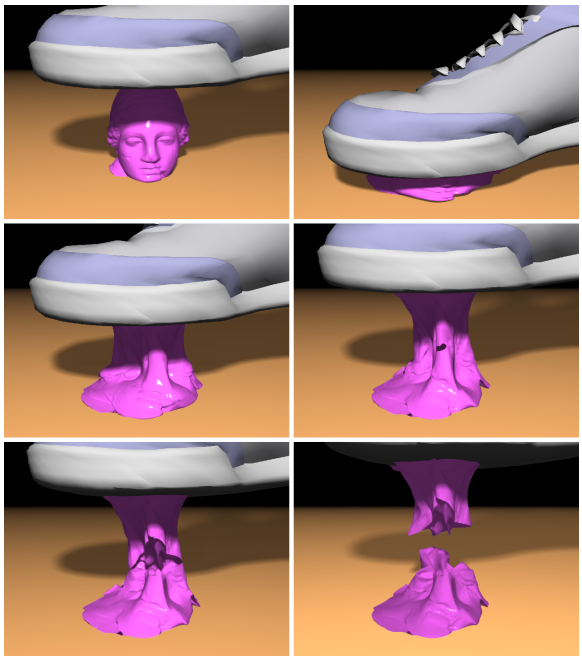


Figure 8: *Highly plastic deformations and ductile fracture using the meshless simulation method discussed in this section.*

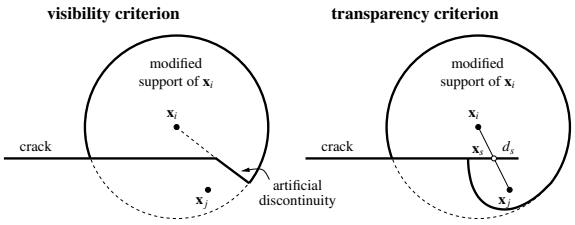


Figure 9: *Comparison of visibility criterion (left) and transparency criterion (right) for the modeling of discontinuities. The visibility criterion introduces an artificial line of discontinuity.*

criterion (see Figure 9, left). However, considering only this line-of-sight constraint can cause undesirable discontinuities within the simulation domain. This can adversely affect the accuracy of the physical simulation, but also has a significant impact on the smoothness of the displacement field used for surface animation (cf. Section 7.2). Indeed, spatially adjacent vertices can be deformed very differently because a particular simulation particle is visible for one vertex, but not for its neighboring vertex, even though the surface points lie on the same side of the crack. Thus, using the visibility criterion would degrade the surface quality, especially in the neighborhood of propagating cracks.

To alleviate this problem, Pauly et al. [PKA*05] use the

transparency method proposed by Organ et al. [OFTB96] to allow partial interaction of points in the vicinity of the crack front. Suppose the ray between two points $\mathbf{x}_i$ and $\mathbf{x}_j$ (this can be a surface point and a simulation particle or two simulation particles) intersects a crack surface at a point $\mathbf{x}_s$ (see Figure 9, right). Then the weight function $\omega_{ij} = \omega(\|\mathbf{x}_j - \mathbf{x}_i\|/h_i)$ is adapted to

$$\omega'_{ij} = \omega(\|\mathbf{x}_j - \mathbf{x}_i\|/h_i + (d_s/\kappa h)^2), \tag{70}$$

where d_s is the distance between $\mathbf{x}_s$ and the closest point on the crack front, h is the average particle spacing in the vicinity of $\mathbf{x}_i$ and κ controls the opacity of the crack surfaces. Effectively, a crack passing between two points lengthens the interaction distance of the points until eventually, in this adapted distance metric, the points will be too far apart to interact.

A comparison between the visibility criterion and the transparency method is shown in Figure 9 and the effect on the weight and shape functions of both methods is illustrated in Figure 10.

Note that the only adaptation which has to be made to the elastic solid animation framework of Section 7.1 for the animation of fracturing materials is the modified distance metric given by Eq. 70. However, this also increases the computational complexity as visibility tests between simulation particles (and surface points) have to be performed and ray-surface intersection tests are necessary to compute the modified transparency weights. However, this additional computational complexity can be reduced significantly by using efficient caching schemes.

As discussed before weight functions are evaluated in the reference (or material coordinate) system. This means that weights between interacting points do not change under deformation. Weights can thus only be affected by propagating cracks. There are two situations where the weight between neighboring points $\mathbf{x}_i$ and $\mathbf{x}_j$ can change. Firstly, if newly added crack surfaces block the line of sight between the previously visible $\mathbf{x}_i$ and $\mathbf{x}_j$. This means that the weight $\omega'_{ij} = \omega_{ij}$ at time t is modified to a weight $\omega'_{ij} < \omega_{ij}$ at time $t + \Delta t$. Secondly, when two points $\mathbf{x}_i$ and $\mathbf{x}_j$ have a transparency weight $\omega'_{ij} < \omega_{ij}$, i.e., the ray connecting the points is already blocked by a crack surface, crack propagation will alter the distance d_s (cf. Eq. 70) and change the transparency weight at time step $t + \Delta t$ to $\omega'_{ij}(t + \Delta t) < \omega'_{ij}(t)$.

To summarize, transparency weights can only change when a crack surface cuts the line between two neighboring points or when a crack already passing between the two points propagates further away. Typically only a few cracks propagate through the material at a time and only a small number of points are affected. Hence, in each time step only a small number of shape functions have to be re-evaluated. Pauly et al. discuss in detail how the affected points can be identified and updated.

Figure 8 shows the animation of plastic material including fracturing. Figure 12 shows the result of brittle fracturing of a very stiff material. For a detailed discussion on when, where and how to initiate and propagate crack surfaces, we refer to [PKA*05, Ada06, Kei06].

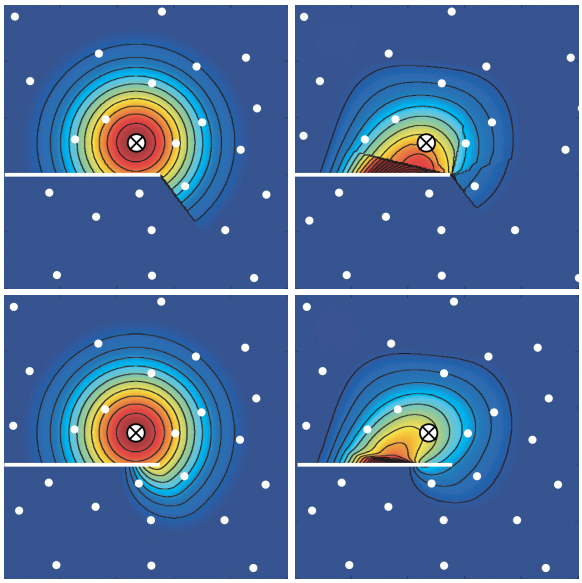


Figure 10: *Comparison of visibility criterion (top) and transparency method (bottom) for an irregularly sampled 2D domain. The effect of a crack, indicated by the horizontal white line, on weight function ω_i and shape function Φ_i is depicted for particle $\mathbf{x}_i$ marked by the cross.*

Adaptive Sampling During fracture simulation, the particle sampling needs to be adapted. Without dynamic resampling, frequent fracturing would quickly degrade the numerical stability of the simulation even for an initially adequately sampled model. New particles need to be inserted in the vicinity of the crack surfaces and in particular around the crack front. At the same time, strong deformations of the model can lead to a poor spatial discretization of the simulation volume, which also requires a dynamic adaptation of the sampling resolution. This is particularly important for highly plastic materials, where the deformed shape can deviate significantly from its original configuration. Note of course that resampling due to strong deformations only makes sense when Eulerian kernels are used.

Pauly et al. propose to use a simple local criterion to determine under-sampling at a particle $\mathbf{x}_i$. Let

$$\Omega_i = \sum_j \frac{\omega'_{ij}}{\omega_{ij}} \tag{71}$$

be the normalized sum of transparency weights cf. Eq. 70. Without visibility constraints, Ω_i is simply the number of simulation particles in the support of $\mathbf{x}_i$. During simulation Ω_i decreases, if fewer neighboring particles are found if the

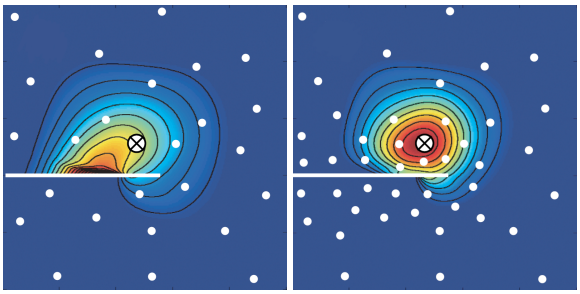


Figure 11: *Effect of resampling on the shape functions. Left: no resampling. Right: resampling.*

transparency weights become smaller due to a crack front passing through the solid or due to severe stretching in plastic materials. If Ω_i drops below a threshold $\Omega_{\min}$ (One typically uses $\Omega_{\min} = 10$), a total of $\lceil \Omega_{\min} - \Omega_i \rceil$ new particles are inserted using an approach similar to [DC99].

The mass associated with the old particles is distributed evenly among the new ones and their support radius is adapted to keep the overall material density constant. Note that mass will not be strictly preserved locally in the sense that the mass distribution of nodes after fracturing will not precisely match the correct distribution according to the separated volumes created by the fracture surface sheets. However, mass will be preserved globally and the local deviations are sufficiently small to not affect the simulation noticeably (cf. [MBF04]).

To prevent excessive resampling for particles very close to a fracture boundary, particle splitting is restricted by prescribing a minimal support radius. Resampling due to fracturing is triggered by the cracks passing through the solid, similar to adapting the visibility weights. Performing these checks comes essentially for free, since all the required spatial queries are already carried out during visibility computation.

The effect of crack propagation and adaptive upsampling on the shape functions is illustrated for a simple 2D example in Figure 11.

7.4. Summary

This section introduced a particle animation framework based on MLS approximations for the simulation of elastic materials. We showed how the material behavior can be simulated using discrete particles and MLS approximations of the elastic forces. We discussed two alternatives for animating the surface mesh along with the particles and showed how the basic framework can be extended to incorporate the simulation of plasticity and fracturing.

One of the nice properties of this animation approach is that the surface and volume representation are decoupled

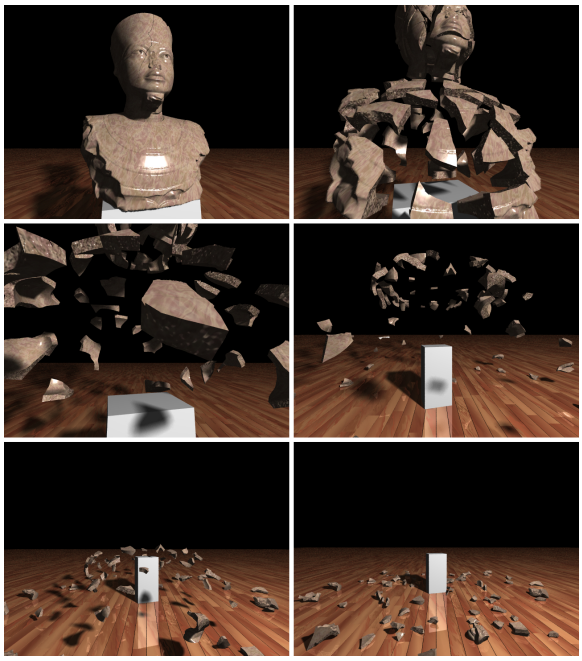


Figure 12: *Brittle fracture of a hollow stone sculpture using the meshless simulation method discussed in this section.*

which allows the animation of highly detailed surfaces without increasing the complexity of the physical representation and simulation. This allows separating visual quality from physical accuracy which is important in computer graphics applications. Müller et al. for example report interaction rates of 27 FPS (on a P4 2.8 GHz laptop) of a model sampled with 200 particles and 10k surface points. This clearly shows the strength of the point-based animation framework: It allows high visual quality with low simulation complexity. Moreover, if a higher surface quality is required, the same animation can be replayed with a high resolution surface wrapped around the simulation points. This is an important feature for example in production environments.

With the addition of plastic materials, the proposed framework allows the animation of a wide range of materials. Moreover, the continuum mechanics based elasticity model has the advantage that the parameters (i.e., Young's Modulus and Poisson's Ratio) have a real physical meaning and can thus be looked up in a mechanics text book for example.

The proposed system has a few limitations though. First, the MLS approximation assumes that each particle has enough neighbors in its support at non-degenerate locations. This means that the approach presented in this section does not work for 2D layers (e.g., thin shells) or 1D strings of particles. Other point-based simulation algorithms have been proposed in the computer graphics community for the animation of for example thin shells [WSG05, GLB*06].

The absence of connectivity information requires the computation of neighboring particles. When animating moderately elastic materials, static neighborhoods can be used. However, when plastic materials are added, these neighborhoods have to be recomputed in each animation frame because the reference system is adapted and therefore the choice of a proper data structure is of key importance.

The extension to fracturing shows the key strengths of the meshless method. Instead of maintaining a consistent volumetric mesh using continuous cutting and re-structuring of finite elements, the method dynamically adjusts the weight functions based on simple visibility constraints and dynamically adapts the particle distribution where needed by simple resampling.

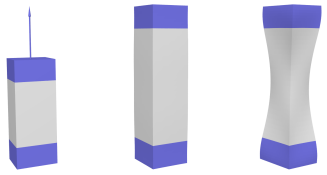


Figure 13: *Illustration of the effect of the different shape modeling constraints. Left: handle constraints are specified to fix the bottom of the box and to move the top to the desired position. Middle: with only the rigidity constraint the total volume is increased by 53%. Right: the total volume remains within 3% of the original when the volume constraint is added.*

8. Geometric Modeling using Physical Metaphors

Another application of the MLS particle approximation method was proposed by Adams et al. [AOW*08] for interactive shape deformation modeling and the design of smooth keyframe animations for deformable objects. The modeling applications presented in this section are based on the same physical models as used for the elastic solid simulation of Section 7 and hence result in realistic deformations. We first discuss the shape modeling algorithm and then detail the extension to deformable animation design.

8.1. Shape Deformations

The meshless deformation field representation used for elastic solid simulation, can be equally well used for shape deformation modeling. Energy terms can be defined and optimized that penalize non-rigid deformations and changes in the shape's volume, while enforcing the user's input constraints. The resulting deformation framework can be used to interactively model complex shapes. It will also serve as the core component in the deformable animation design framework that we will discuss in Section 8.2.

Again, the object is sampled using a set of particles and the continuous deformation field is found using MLS approximation of the displacements at the particles as

$$\mathbf{u}(\mathbf{x}) = \sum_i \Phi_i(\mathbf{x})\mathbf{u}_i. \tag{72}$$

Adams et al. [AOW*08] propose to use a complete linear basis of order $n = 1$ ($\mathbf{p}(\mathbf{x}) = [1\ x\ y\ z]^T$, and $\mathbf{x} = [x\ y\ z]^T$) in the shape function construction. This allows reconstructing rigid motions exactly. Eq. 72 maps every point $\mathbf{x}$ in the undeformed shape to its image $\mathbf{x}+\mathbf{u}(\mathbf{x})$ in the deformed shape. For ease of notation this mapping is denoted in this section as

$$\mathbf{f}(\mathbf{x}) = \mathbf{x}+\mathbf{u}(\mathbf{x}). \tag{73}$$

Given the above formulation, the goal is to find the particle displacement vectors $\mathbf{u}_i$ so that the resulting continuous deformation field $\mathbf{u}(\mathbf{x})$ (or equivalently $\mathbf{f}(\mathbf{x})$) fulfills desirable properties (see also Figure 13).

Handle Constraints Handle constraints restrict the movement of certain points of the shape. For example, the user may want to fix the legs while pulling one of the arms of the model to deform its shape. Thus, a handle constraint simply states that the deformation field $\mathbf{f}(\mathbf{x}_k)$ should move a given point $\mathbf{x}_k$ to a prescribed target position $\mathbf{x}'_k$. Given a set of K handle constraints $(\mathbf{x}_k, \mathbf{x}'_k)$, the energy to minimize is:

$$E_{\text{handle}} = \sum_{k=1}^{K} \|\mathbf{f}(\mathbf{x}_k) - \mathbf{x}'_k\|^2. \tag{74}$$

Rigidity The deformation field is completely rigid if at all points $\nabla\mathbf{f}^T(\mathbf{x})\nabla\mathbf{f}(\mathbf{x}) = \mathbf{I}$. Hence, to obtain as-rigid-as-possible shape deformations, following energy should be minimized:

$$E_{\text{rigidity}} = \int_{\mathbf{x}\in\mathcal{V}} \|\nabla\mathbf{f}^T(\mathbf{x})\nabla\mathbf{f}(\mathbf{x}) - \mathbf{I}\|_F^2 d\mathbf{x}, \tag{75}$$

where the integration is over the (undeformed) shape's volume $\mathcal{V}$ and $\|\cdot\|_F$ is the Frobenius norm. Note the similarity between Eq. 75 and Eq. 61. To facilitate optimization, it is often sufficient to only penalize non-rigid behavior at the particle positions. This leads to the discretized equation

$$E_{\text{rigidity}} = \sum_i V_i \|\nabla\mathbf{f}^T(\mathbf{x}_i)\nabla\mathbf{f}(\mathbf{x}_i) - \mathbf{I}\|_F^2. \tag{76}$$

Here, $\nabla\mathbf{f}(\mathbf{x}) = \mathbf{I}+\nabla\mathbf{u}(\mathbf{x})$, where $\nabla\mathbf{u}(\mathbf{x})$ is computed using the analytic derivative formula of Eq. 32. The scaling by the particle volume $V_i = 4/3\pi h_i^3$ can be omitted when using uniform particle radii ($h_i = h$). In the following we will directly write down the discretized equation (cf. Eq. 76) and leave out the continuous one (cf. Eq. 75) for the sake of brevity.

Volume Preservation The deformation field preserves the shape's volume if and only if $|\nabla\mathbf{f}(\mathbf{x})| = 1$ over the whole shape. Thus, the deformed shape's volume matches its orig-

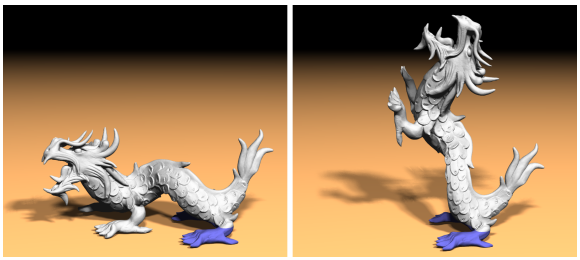


Figure 14: *Deformation of the dragon model obtained using a coarse set of only* 60 *particles. The particle deformations are computed on the CPU, while the high resolution surface is deformed faithfully on the GPU. The interaction was performed at a rate of* 55 *fps for the model with* 100*k vertices and* 10 *fps for the model with* 500*k vertices.*

inal volume as closely as possible if one minimizes

$$E_{\text{volume}} = \sum_i V_i (|\nabla \mathbf{f}(\mathbf{x}_i)| - 1)^2. \tag{77}$$

Again, this equation corresponds to the volume preservation force used for elastic solid simulation given in Eq. 65.

The optimal deformation field $\mathbf{f}(\mathbf{x})$ can now be found by minimizing the total sum of constraint energies:

$$E = \lambda_1 E_{\text{handle}} + \lambda_2 E_{\text{rigidity}} + \lambda_3 E_{\text{volume}}, \tag{78}$$

where the parameters λ_1, λ_2 and λ_3 vary the contribution of each of the different constraints. It can be easily seen from Eq. 74, Eq. 76 and Eq. 77 that E is a multivariate polynomial of total degree 6 in the unknowns (the particle displacements $\mathbf{u}_i$). Minimizing Eq. 78 hence requires a non-linear solver. Note however, that taking analytic derivatives with respect to the unknowns is straightforward.

Adams et al. also discuss a particle sampling algorithm that distributes particles uniformly over the shape's volume and defines the coupling between the particles by computing distances within the shape instead of using simple Euclidean distances. During interactive modeling, the surface is deformed on the GPU using Alternative 1 described in Section 7.2. Even though the resulting optimization problem is non-linear, interactive deformations for highly detailed geometric shapes are possible on standard hardware.

An example of a deformation of a high resolution dragon model is shown in Figure 14.

8.2. Deformable Shape Motions

The shape modeling framework can be easily extended into an animation design framework for deformable objects. In such a framework, the user specifies keyframe poses and the algorithm computes an optimal interpolatory deformable motion, while satisfying certain constraints such as shape preservation and collision avoidance. Again, by sampling the deformation field only at a sparse discrete number of time instances and by using the meshless shape approximation scheme of Section 3, a continuous time dependent deformation field is obtained over the whole time interval. An adaptive temporal sampling strategy limits the number of unknowns and allows rapid motion path modeling.

Each particle $\mathbf{x}_i$'s motion path is now sampled at T discrete times t_j, and defined by the deformation vectors $\mathbf{u}_{i,t_j}$ (see Figure 15). These discrete time representations are called *frames*. If each frame was treated separately, the deformation at $\mathbf{x}$ would be represented using Eq. 72 as

$$\mathbf{u}_{t_j}(\mathbf{x}) = \sum_i \Phi_i(\mathbf{x}) \mathbf{u}_{i,t_j}. \tag{79}$$

By assigning a support radius h_{t_j} to each frame t_j, we can obtain a smooth time dependent deformation at position $\mathbf{x}$ and time t by MLS approximation over the neighboring frames

$$\mathbf{u}(\mathbf{x},t) = \sum_{j=1}^{T} \Phi_j(t) \mathbf{u}_{t_j}(\mathbf{x}). \tag{80}$$

Here the shape functions are one-dimensional and defined over the time domain. Second order consistency is obtained by using the complete polynomial basis $\mathbf{p}(t) = [1\ t\ t^2]$. This allows reconstructing most deformable motions with only a small number of frames.

Substituting Eq. 79 in Eq. 80 yields the final expression

$$\begin{aligned} \mathbf{u}(\mathbf{x},t) &= \sum_{j=1}^{T} \Phi_j(t) \sum_i \Phi_i(\mathbf{x}) \mathbf{u}_{i,t_j} \\ &= \sum_{j=1}^{T} \sum_i \Phi_j(t) \Phi_i(\mathbf{x}) \mathbf{u}_{i,t_j}. \end{aligned} \tag{81}$$

Hence, the deformation at position $\mathbf{x}$ at time t is a linear combination of the deformations $\mathbf{u}_{i,t_j}$ of the spatially neighboring particles $\mathbf{x}_i$ at neighboring frames. These particle deformations are the unknowns that are solved for.

Note that space and time are decoupled and separate MLS shape functions are used to approximate the displacement field within the shape and to define the continuous displacement field over time.

Within a frame t_j, the rigidity and volume preserving penalties as defined before for the shape modeling algorithm are used. To solve for a temporally changing deformation field, additional constraints are added (see also Figure 16).

Keyframes The user can specify the desired position for the shape at certain keyframe times $t_k, k \in 1 \ldots K$. Keyframes are typically defined at the beginning and end of a motion, but can also constrain shape poses at intermediate times. These keyframes are converted into handle constraints by specifying one handle constraint for each particle in each keyframe. Note that the user can specify the shape in a keyframe in a *deformed* pose or can only constrain a subset of the shape in a keyframe. The resulting energy function is denoted as E_{keyframe} in the following.

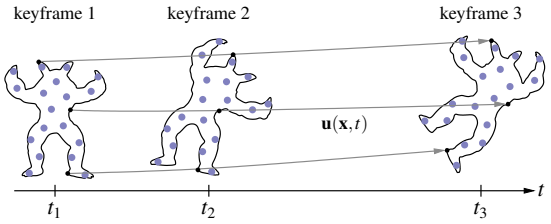


Figure 15: *The goal is to find a smooth motion of the deformable shape that interpolates the keyframes. The continuous time dependent deformation field is defined from the frames as* $\mathbf{u}(\mathbf{x},t) = \sum_{j=1}^{T}\sum_i \Phi_j(t)\Phi_i(\mathbf{x})\mathbf{u}_{i,t_j}$ *where the unknowns* $\mathbf{u}_{i,t_j}$ *are the nodal displacements we will solve for. There is one displacement vector* $\mathbf{u}_{i,t_j}$ *for each particle* i *in each frame* t_j.

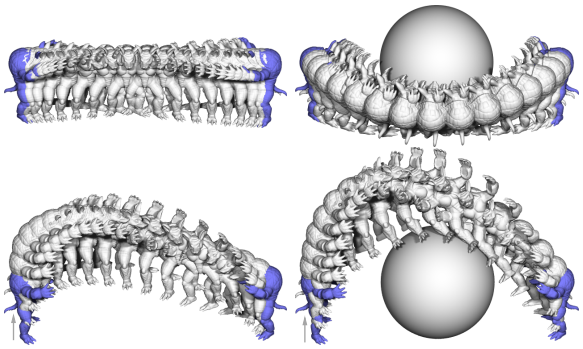


Figure 16: *Illustration of the effect of the different temporal constraints. Top left: smooth interpolation between two keyframes using the keyframe and acceleration constraints. Lower left: result after prescribing the velocity in the first frame (gray arrow). Top right: result after adding an obstacle. Bottom right: result after adding the velocity and obstacle constraints.*

Velocity Constraints Along with specifying the shape's keyframe poses, the user can specify a velocity $\mathbf{v}_k$ that the deformation field should satisfy at the keyframes t_k. This velocity should match the temporal derivative of the shape's deformation field at time t_k, for all points in the shape. For all keyframes together and discretized at the the particles, this gives the constraint

$$E_{\text{velocity}} = \sum_{k=1}^{K}\sum_i V_i \left\| \frac{\partial \mathbf{u}}{\partial t}(\mathbf{x}_i,t_k) - \mathbf{v}_k \right\|^2. \qquad (82)$$

The analytic time derivatives of the deformation field are computed from the shape function's time derivatives $\partial\Phi_j(t)/\partial t$ using Eq. 30.

Acceleration To obtain smooth motion, the shape's acceleration is bounded. Very similar to the above velocity constraint this yields the energy penalty

$$E_{\text{acceleration}} = \sum_{j=1}^{T}\sum_i h_{t_j} V_i \left\| \frac{\partial^2 \mathbf{u}}{\partial t^2}(\mathbf{x}_i,t_j) \right\|^2, \qquad (83)$$

where h_{t_j} is the support radius of frame t_j. The second derivatives of the displacement field with respect to time can be computed analytically by computing $\partial^2\Phi_j(t)/\partial t^2$ as detailed in [FM03]. Note again that the resulting expression for the acceleration is a simple linear combination of the particle displacements, i.e., the unknowns $\mathbf{u}_{i,t_j}$.

Obstacle Avoidance A final penalty function prevents penetration of the deforming object with possible obstacles in the scene. Assuming that obstacles can be represented by a (time dependent) distance field $d(\mathbf{x},t)$ and that a point $\mathbf{x}$ is penetrating at time t if $d(\mathbf{x},t) \geq 0$, the following collision avoidance energy is obtained

$$E_{\text{obstacles}} = \sum_{j=1}^{T}\sum_i h_{t_j} V_i d^2(\mathbf{f}(\mathbf{x}_i,t_j),t_j), \qquad (84)$$

where $\mathbf{f}(\mathbf{x},t) = \mathbf{x} + \mathbf{u}(\mathbf{x},t)$. Note that although this energy penalizes collisions for the particles at the frames, it does not prevent collisions of all points at all times. To prevent artifacts rising from only constraining the particles, the particles are fattened to spheres (by using their support radii h_i) and constrained to be outside the obstacles (hence, $d(\mathbf{f}(\mathbf{x}_i,t_j),t_j) + h_i$ is used instead of $d(\mathbf{f}(\mathbf{x}_i,t_j),t_j)$ in the above equation). If the union of these spheres covers the whole shape, this adequately prevents penetrations at the frames t_j. However, nothing restricts the shape from colliding with obstacles at other time instances, as the interpolation scheme is collision oblivious. Adaptive temporal sampling solves this issue and will be explained below.

Given the discrete spatial and temporal sampling, the optimal time dependent deformation field $\mathbf{f}(\mathbf{x},t) = \mathbf{x} + \mathbf{u}(\mathbf{x},t)$ can now be found by minimizing the total energy

$$\begin{aligned} E \;=\; & \lambda_1 E_{\text{keyframe}} + \lambda_2 E_{\text{rigidity}} + \lambda_3 E_{\text{volume}} \\ & \lambda_4 E_{\text{velocity}} + \lambda_5 E_{\text{acceleration}} + \lambda_6 E_{\text{obstacles}}, \end{aligned} \qquad (85)$$

where λ_1 to λ_6 are again parameters to modify the contribution of the various constraints. Similar to the energy function of Eq. 78, the above equation is a polynomial of total degree 6 in the unknown nodal displacements $\mathbf{u}_{i,t_j}$.

We now discuss how the deformation field is temporally discretized by iteratively creating and solving for new frames.

Adaptive Temporal Sampling In the (single frame) deformation modeling part, the total number of unknowns to solve for is $3N$, where N is the number of particles. In the motion planning setting, the total number of unknowns multiplies to $3NT$, where T is the number of frames. To keep this number sufficiently low, Adams et al. propose an adaptive time sampling strategy that introduces frames iteratively in problematic regions. Initially, all frames t_j correspond to keyframes

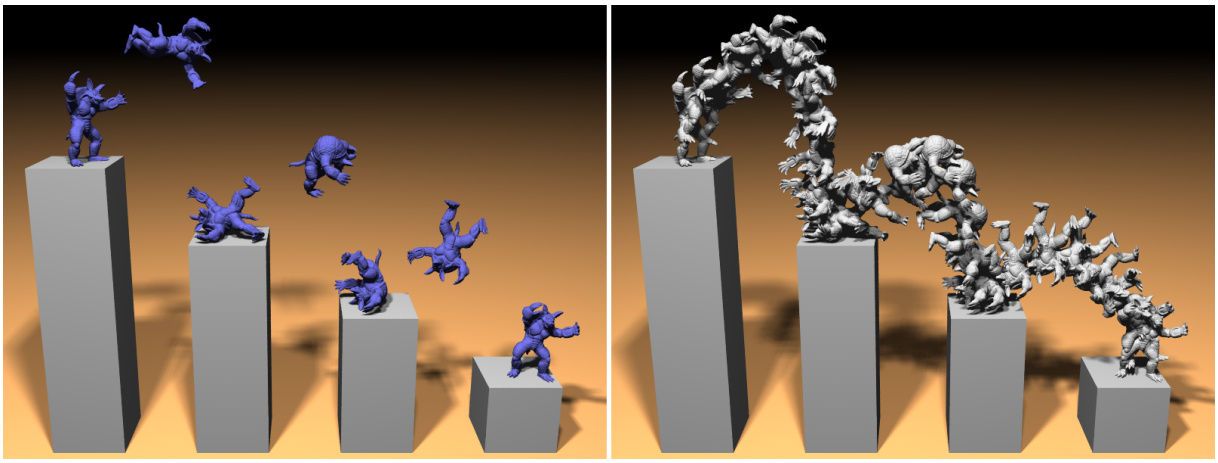


Figure 17: *Left: keyframe poses obtained using the meshless shape deformation algorithm. Right: smooth interpolated motion obtained using the shape interpolation algorithm.*

specified by the user. The displacement field is optimized as discussed above and the error is evaluated at a dense number of frames in between the frames t_j (typically evaluation is done at 10 intermediate frames). A new frame t_j is introduced at the time instance t_{max} where the error is maximal. The algorithm then proceeds by solving again and iterating until a desired accuracy is obtained. When a new frame is introduced at time t_{max}, the particle deformation vectors at the new frame are initialized kas $\mathbf{u}_{i,t_{max}} = \sum_{t=1}^{T} \Phi_j(t_{max})\mathbf{u}_{i,t_j}$. This yields a good initial guess for the subsequent solve.

The proposed adaptive sampling strategy greatly reduces the number of unknowns and introduces frames only at problematic regions, for example when there is high acceleration, or when the deforming shape is penetrating an obstacle. Thanks to the second order consistency in the temporal shape functions $\Phi_j(t)$, most motions can be represented by only a very low number of frames. The frames' support radii h_{t_j} (and hence weight functions $\omega_{t_j}(t)$) are adapted to the frame spacing to ensure that every time instance t is covered by at least 3 neighboring frames. This guarantees non-singular moment matrices and safe computation of the temporal shape functions. If only two frames are present, simple linear interpolation defines the shape's deformation field.

An example of interpolated motion of a deformable object is shown in Figure 17.

8.3. Summary

This section presented a framework and algorithms for interactive shape and motion modeling. Using energies derived from physical metaphors similar to the forces used for elastic solid simulations, realistic shape deformations are obtained. A particle discretization and the approximation using the MLS method enables interactivity as it allows using only few particles to adequately deform high resolution surfaces.

The shape modeling framework is easily extended to a motion planning or keyframe interpolation algorithm for deformable objects, by using the MLS approximation to interpolate between adjacent frames. Note that this is the only application presented in this tutorial that uses a particle approximation scheme where the domain is time and not the Euclidean space. A drawback of using the MLS approximations to interpolate motion is that, contrary to translations, rigid rotations are not recovered exactly. Indeed, even though a quadratic basis is used, rotations cannot be represented using simple polynomials in time.

9. Conclusion

This tutorial treated meshless methods for animation and modeling. In order to offer a principled and comprehensive perspective on the material, we have first introduced the function approximation methods that form the foundation of these methods, before discussing their applications in computer graphics. We have seen how the requirements of the simulation determine the choice of approximation method: fluid simulation requires frequent recomputation of the neighborhood structure and shape functions, and a low-cost, low-accuracy approximation like SPH appears optimal. Conversely, continuum elasticity requires the reconstruction of linear functions, and therefore demands the more accurate MLS approximation.

Similarly, the choice of approximation method influences the simulation algorithm. The most striking example of this is the near-universal absence of mesh-based Lagrangian fluid simulations form the graphics literature. Since fluids are often highly turbulent, permanently recomputing the mesh

structure is impractical, and mesh-based fluid simulations discretize the embedding space instead. The same holds for elasticity computations: Since meshless sampling makes it easy to resample, adaptive sampling is much more attractive for meshless methods.

When approaching a problem in simulation or modeling, it is a good idea to analyze the problem and consider which advantages and problems a solution using both mesh-based and meshless approaches might have. Meshless approaches require only the neighborhood graph, which is easy to compute and maintain. Resampling or topological changes in the material that require restructuring the neighborhood information are therefore relatively simple operations in a meshless setting. On the other hand, mesh-based approaches offer significantly more mathematical structure to be exploited by the function approximation mechanism. Examples are the consistency of differential operators (for fluid simulation) or the exact conservation of integral properties (for continuum elasticity). Sometimes, careful consideration might lead to a hybrid technique combining the strength of both approaches, successful examples of this are the PIC and FLIP techniques for incompressible SPH [Har64, BR86, ZB05], or the particle level set techniques for interface tracking in Eulerian fluid simulations [EMF02, ELF05].

We hope to have given the reader a useful introduction to meshless methods in computer graphics.

Appendix A: SPH Kernel Functions

A good polynomial kernel function has been introduced in [MCG03], here shown normalized for use in three dimensions:

$$\omega_h(d) = \begin{cases} \frac{315}{64\pi h^3}\left(1-\frac{d^2}{h^2}\right)^3 & d<h, \\ 0 & \text{otherwise.} \end{cases} \tag{86}$$

This since d is only used squared, (86) can be evaluated without using a square root. The first and second derivatives are:

$$\omega_h'(d) = \begin{cases} -\frac{945}{32\pi h^3}\frac{d}{h^2}\left(1-\frac{d^2}{h^2}\right)^2 & 0<d<h, \\ 0 & \text{otherwise,} \end{cases} \tag{87}$$

$$\omega_h''(d) = \begin{cases} \frac{945}{32\pi h^3}\frac{1}{h^2}\left(1-\frac{d^2}{h^2}\right)\left(5\frac{d^2}{h^2}-1\right) & d<h, \\ 0 & \text{otherwise.} \end{cases} \tag{88}$$

For function approximation in 2D, the normalization has to be adjusted, yielding:

$$\omega_h(d) = \begin{cases} \frac{4}{\pi h^2}\left(1-\frac{d^2}{h^2}\right)^3 & d<h, \\ 0 & \text{otherwise,} \end{cases} \tag{89}$$

with derivatives

$$\omega_h'(d) = \begin{cases} -\frac{24}{\pi h^2}\frac{d}{h^2}\left(1-\frac{d^2}{h^2}\right)^2 & 0<d<h, \\ 0 & \text{otherwise,} \end{cases} \tag{90}$$

$$\omega_h''(d) = \begin{cases} \frac{24}{\pi h^2}\frac{1}{h^2}\left(1-\frac{d^2}{h^2}\right)\left(5\frac{d^2}{h^2}-1\right) & d<h, \\ 0 & \text{otherwise.} \end{cases} \tag{91}$$

[MCG03] also introduce a spiky kernel $\hat{\omega}_h(d)$ to avoid clumping in low resolution fluid simulations. Its gradient does not go smoothly to zero at $d=0$, thus preventing a force-free state in which particles clump at a single position. For 3D simulation, it is given by

$$\hat{\omega}_h(d) = \begin{cases} \frac{15}{\pi h^3}\left(1-\frac{d}{h}\right)^3 & d<h, \\ 0 & \text{otherwise,} \end{cases} \tag{92}$$

with first and second derivatives

$$\hat{\omega}_h'(d) = \begin{cases} -\frac{45}{\pi h^3}\frac{1}{h}\left(1-\frac{d}{h}\right)^2 & 0<d<h, \\ 0 & \text{otherwise,} \end{cases} \tag{93}$$

$$\hat{\omega}_h''(d) = \begin{cases} \frac{90}{\pi h^3}\frac{1}{h^2}\left(1-\frac{d}{h}\right) & d<h, \\ 0 & \text{otherwise.} \end{cases} \tag{94}$$

In 2D, the normalization yields

$$\hat{\omega}_h(d) = \begin{cases} \frac{10}{\pi h^2}\left(1-\frac{d}{h}\right)^3 & d<h, \\ 0 & \text{otherwise,} \end{cases} \tag{95}$$

with first and second derivatives

$$\hat{\omega}_h'(d) = \begin{cases} -\frac{30}{\pi h^2}\frac{1}{h}\left(1-\frac{d}{h}\right)^2 & 0<d<h, \\ 0 & \text{otherwise,} \end{cases} \tag{96}$$

$$\hat{\omega}_h''(d) = \begin{cases} \frac{60}{\pi h^2}\frac{1}{h^2}\left(1-\frac{d}{h}\right), & \\ 0 & \text{otherwise.} \end{cases} \tag{97}$$

References

[Ada06] ADAMS B.: *Point-Based Modeling, Animation and Rendering of Dynamic Objects*. PhD thesis, Katholieke Universiteit Leuven, May 2006.

[AM93] ARYA S., MOUNT D. M.: Algorithms for fast vector quantization. In *Proceedings of the IEEE Data Compression Conference '93* (1993), pp. 381–390.

[AOW*08] ADAMS B., OVSJANIKOV M., WAND M., SEIDEL H.-P., GUIBAS L. J.: Meshless modeling of deformable shapes and their motion. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2008).

[APKG07] ADAMS B., PAULY M., KEISER R., GUIBAS L. J.: Adaptively sampled particle fluids. In *Proc. SIGGRAPH '07* (2007).

[Ben75] BENTLEY J. L.: Multidimensional binary search tree used for associative searching. *Communications of the ACM 18* (1975), 509–517.

[BF79] BENTLEY J. L., FRIEDMAN J. H.: Data structures for range searching. *ACM Computing Surveys 11* (1979), 397–409.

[BKO*96] BELYTSCHKO T., KRONGAUZ Y., ORGAN D., FLEMING M., KRYSL P.: Meshless methods: An overview and recent developments. *Comp. Meth. in Appl. Mech. Eng. 139*, 3 (1996).

[BLG94] BELYTSCHKO T., LU Y., GU L.: Element-free galerkin methods. *Int. J. Numer. Meth. Engng 37* (1994), 229–256.

[Bli82] BLINN J. F.: A generalization of algebraic surface drawing. *ACM Transactions on Graphics 1*, 3 (1982), 235–256.

[BR86] BRACKBILL J. U., RUPPEL H. M.: Flip: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *J. Comput. Phys. 65*, 2 (1986), 314–343.

[BT07] BECKER M., TESCHNER M.: Weakly compressible sph for free surface flows. In *Proc. SCA 07* (2007), pp. 63–72.

[CBP05] CLAVET S., BEAUDOIN P., POULIN P.: Particle-based viscoelastic fluid simulation. In *Proceedings of the Symposium on Computer Animation'05* (2005), pp. 219–228.

[CEL06] COLIN F., EGLI R., LIN F. Y.: Computing a null divergence velocity field using smoothed particle hydrodynamics. *J. Comput. Phys. 217*, 2 (2006), 680–692.

[DC99] DESBRUN M., CANI M.-P.: *Space-Time Adaptive Simulation of Highly Deformable Substances*. Tech. rep., INRIA Nr. 3829, 1999.

[DG96] DESBRUN M., GASCUEL M.-P.: Smoothed particles: A new paradigm for animating highly deformable bodies. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation'96* (1996), pp. 61–76.

[Dil99] DILTS G. A.: Moving least-squares particle hydrodynamics i: consistency and stability. *International Journal for Numerical Methods in Engineering 44*, 8 (1999), 1115–1155.

[Dil00] DILTS G. A.: Moving least-squares particle hydrodynamics ii: conservation and boundaries. *International Journal for Numerical Methods in Engineering 48*, 10 (2000), 1503–1524.

[ELF05] ENRIGHT D., LOSASSO F., FEDKIW R.: A fast and accurate semi-lagrangian particle level set. *Computers and Structures 83* (2005), 479–490.

[EMF02] ENRIGHT D., MARSCHNER S., FEDKIW R.: Animation and rendering of complex water surfaces. In *ACM Transactions on Graphics (SIGGRAPH 2002 Proceedings)* (2002), pp. 736–744.

[FBF77] FRIEDMAN J. H., BENTLEY J. L., FINKEL R. A.: An algorithm for finding best matches in logarithmic expected time. *ACM Transactions in Mathematical Software 3* (1977), 209–226.

[FM03] FRIES T.-P., MATTHIES H. G.: *Classification and Overview of Meshfree Methods*. Tech. rep., TU Brunswick, Germany Nr. 2003-03, 2003.

[FOA03] FELDMAN B. E., O'BRIEN J. F., ARIKAN O.: Animating suspended particle explosions. *ACM Trans. Graph. 22*, 3 (2003), 708–715.

[FOK05] FELDMAN B. E., O'BRIEN J. F., KLINGNER B. M.: Animating gases with hybrid meshes. In *Proceedings of SIGGRAPH'05* (2005), pp. 904–909.

[FP86] FLAJOLET P., PUECH C.: Partial match retrieval of multidimensional data. *Journal of the ACM 33* (1986), 371–407.

[GLB*06] GUO X., LI X., BAO Y., GU X., QIN H.: Meshless thin-shell simulation based on global conformal parameterization. *IEEE Transactions on Visualization and Computer Graphics 12*, 3 (2006).

[GM77] GINGOLD R. A., MONAGHAN J. J.: Smoothed particle hydrodynamics: Theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society 181* (1977), 375–389.

[GP07] GROSS M., PFISTER H.: *Point-Based Graphics (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[GSLF05] GUENDELMAN E., SELLE A., LOSASSO F., FEDKIW R.: Coupling water and smoke to thin deformable and rigid shells. *ACM Trans. Graph. 24*, 3 (2005), 973–981.

[Har64] HARLOW F. H.: The particle in cell computing methods for fluid dynamics. *Methods of Computational Physics 3* (1964), 319–343.

[HBSL03] HARRIS M., BAXTER W., SCHEUERMANN T., LASTRA A.: Simulation of cloud dynamics on graphics hardware. In *Proc. Graphics Hardware* (2003).

[KAG*06] KEISER R., ADAMS B., GUIBAS L., DUTRÉ P., PAULY M.: *Multiresolution Particle-Based Fluids*. Tech. Rep. 520, ETH Zürich, 2006.

[Kei06] KEISER R.: *Meshless Lagrangian Methods for Physics-Based Animations of Solids and Fluids*. PhD thesis, ETH Zürich, 2006.

[KFCO06] KLINGNER B. M., FELDMAN B. A., CHENTANEZ N., O'BRIEN J. F.: Fluid animation with dynamic meshes. In *Proceedings of SIGGRAPH'06* (2006), pp. 820–825.

[KO96] KOCHIZUKA S., OKA Y.: Moving particle semi-implicit method for fragmentation of incompressible fluid. *Nuclear Science Engineering 123* (1996), 421–434.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), ACM Press, pp. 163–169.

[LGF04] LOSASSO F., GIBOU F., FEDKIW R.: Simulating water and smoke with an octree data structure. In *Proceedings of SIGGRAPH'04* (2004), pp. 457–462.

[LL03] LIU G.-R., LIU M.: *Smoothed Particle Hydrodynamics*. World Scientific, 2003.

[Luc77] LUCY L. B.: A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal 82*, 12 (1977), 1013–1024.

[MBF04] MOLINO N., BAO Z., FEDKIW R.: A virtual node algorithm for changing mesh topology during simulation. *ACM Trans. Graph. 23*, 3 (2004), 385–392.

[MCG03] MÜLLER M., CHARYPAR D., GROSS M.: Particle-based fluid simulation for interactive applications. In *Proceedings of the Symposium on Computer Animation'03* (2003), pp. 154–159.

[MKN*04] MÜLLER M., KEISER R., NEALEN A., PAULY M., GROSS M., ALEXA M.: Point based animation of elastic, plastic and melting objects. *Proceedings of 2004 ACM SIGGRAPH Symposium on Computer Animation* (2004).

[Mon92] MONAGHAN J. J.: Smoothed particle hydrodynamics. *Annu. Rev. Astron. Physics 30* (1992), 543.

[Mon94] MONAGHAN J. J.: Simulating free surface flows with SPH. *J. Comput. Phys. 110*, 2 (1994), 399–406.

[Mon05] MONAGHAN J. J.: Smoothed particle hydrodynamics. *Reports on Progress in Physics 68* (2005), 1703–1759.

[MSKG05] MÜLLER M., SOLENTHALER B., KEISER R., GROSS M.: Particle-based fluid-fluid interaction. In *Proceedings of the Symposium on Computer Animation'05* (2005), pp. 237–244.

[MTHG04] MÜLLER M., TESCHNER M., HEIDELBERGER B., GROSS M.: Interaction of fluids with deformable objects. In *Proceedings of the Conference on Computer Animation and Social Agents'04* (2004), pp. 159–171.

[NMK*05] NEALEN A., MÜLLER M., KEISER R., BOXERMAN E., CARLSON M.: Physically based deformable models in computer graphics. In *Eurographics 2005 State of the Art Report* (2005).

[OBH02] O'BRIEN J. F., BARGTEIL A. W., HODGINS J. K.: Graphical modeling and animation of ductile fracture. In *Proceedings of SIGGRAPH 2002* (2002), Computer Graphics Proceedings, Annual Conference Series, ACM, ACM Press / ACM SIGGRAPH, pp. 291–294.

[OFTB96] ORGAN D., FLEMING M., TERRY T., BELYTSCHKO T.: Continuous meshless approximations for nonconvex bodies by diffraction and transparency. *Computational Mechanics 18* (1996), 1–11.

[PKA*05] PAULY M., KEISER R., ADAMS B., DUTRÉ P., GROSS M., GUIBAS L. J.: Meshless animation of fracturing solids. *ACM Trans. Graph. 24*, 3 (2005), 957–964.

[PTB*03] PREMOZE S., TASDIZEN T., BIGLER J., LEFOHN A., WHITAKER R.: Particle based simulation of fluids. In *Proceedings of Eurographics '03* (2003), pp. 401–410.

[Qui83] QUINLAN J. R.: Learning efficient classification procedures and their applications to chess endgames. In *Machine Learning* (1983), Michalski R., Carbonell J., Mitchell T., (Eds.).

[Ros56] ROSENBLATT M.: Remarks on some nonparametric estimates of a density function. *Annals of Mathematical Statistics 27* (1956), 832–835.

[Saa86] SAAD Y.: On the condition numbers of modified moment matrices arising in least squares approximation in the complex plane. *Journal of Numerical Mathematics 48* (1986), 337–347.

[Sam05] SAMET H.: *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2005.

[She68] SHEPARD D.: A two dimensional interpolation function for irregular spaced data. In *Proc. 23rd Nat. Conf. ACM* (1968), pp. 517–524.

[Sta99] STAM J.: Stable fluids. In *Proceedings of SIGGRAPH'99* (1999), pp. 121–128.

[THM*03] TESCHNER M., HEIDELBERGER B., MÜLLER M., POMERANERTS D., GROSS M.: Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of Vision, Modeling, and Visualization (VMV)'03* (2003), pp. 47–54.

[WAD94] WESS S., ALTHOFF K.-D., DERWAND G.: Using k-d trees to improve the retrieval step in case-based reasoning. In *Selected papers from the European Workshop on Case Based Reasoning '93* (1994), pp. 167–181.

[WSG05] WICKE M., STEINEMANN D., GROSS M.: Efficient animation of point-based thin shells. In *Proceedings of Eurographics '05* (2005), pp. 667–676.

[ZB05] ZHU Y., BRIDSON R.: Animating Sand as a Fluid. In *Proceedings of SIGGRAPH'05* (2005), pp. 965–972.