Data-Driven Methods for Interactive Simulation of Complex Phenomena

Matthew Luchak Stanton

CMU-CS-14-136
September 30, 2014

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Adrien Treuille, Chair
Kayvon Fatahalian
Srinivasa Narasimhan
Doug James, Cornell University

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant numbers IIS-0953985, DGE-0750271, DGE-1252522, and DGE-0750271; Stanford University (National Institute of Health) under grant number 60211611102700A; Google; and the Okawa Foundation.

Keywords: data-driven simulation, data-driven animation, reduced models, games, galerkin projection, fluid simulation, solid-fluid coupling, liquid simulation, radiosity, domain decomposition, constraint reduction, crowdsourcing, player models

Abstract

Creating realistic virtual worlds requires fast, detailed physical simulations. Traditional simulation techniques based on discretization in time and space must trade speed for detail. Frequently, this tradeoff results in either coarse, unrealistic simulation, or slower-than-realtime response. Data-driven simulation techniques avoid this tradeoff by operating on compact representations of simulation state, which can be updated quickly due to their small size. These representations are learned from training simulations that resemble the runtime output we want the simulation to produce. In this thesis, we greatly expand the scope of data-driven simulation in practical applications by answering three important questions. First, how can we reconfigure simulation domains at runtime? While simple forms of data-driven simulation operate in a monolithic fashion, we show how one important data-driven simulation technique can be extended to create modular simulation tiles that can be rearranged at runtime. Second, how can we simulate a wide variety of phenomena? One popular data-driven simulation method, Galerkin projection, only works for simulations with polynomial dynamics. We present an extension of Galerkin projection to dynamics that include division and roots, enabling its application to new phenomena. Finally, how can we ensure that we select appropriate training data? Selecting good training data is critical to ensure good speed and realism from data-driven simulations. We describe a method for building continually-improving data-driven simulations that use recordings of user interactions to guide their selection of training data, guaranteeing that the simulation is trained with data appropriate to its actual runtime use. These contributions move us closer to the ability to create detailed, immersive, interactive simulations of any phenomenon.

Contents

Chapter 1

Introduction

Creating convincing virtual worlds requires high-quality simulation. For nearly as long as there have been graphical displays, there have been interactive simulations of physics. Over the past 50 years, interactive physical simulation has advanced from simple 2D projectile motion to beautiful, highly detailed simulations of phenomena including rigid bodies, smoke, water, and cloth. These simulations have enabled the creation of entirely new forms of art and genres of games.

Such simulations, while beautiful, are often incapable of running at interactive rates, especially at visually convincing resolutions. The short film *Geri's Game* [Pixar 1997] was released in 1997 and featured realistic simulated cloth; Quake 2 [id Software 1997], released just weeks later, featured low-polygon characters with clothing no more complex than texture maps. Meanwhile, interactive cloth simulation has only recently become practical, while the state of the art in offline simulation has continued to advance rapidly.

There are exceptions, such as rigid-body motion, and the number of such exceptions continues to grow as computing power increases, but simulation is in no danger of running out of room at the top: new simulation techniques to handle new phenomena and new interactions are constantly being developed, and these new techniques tend to be quite expensive. Furthermore, two simultaneous trends ensure that the gap between computing power available offline and computing power available in typical interactive contexts will remain large for the foreseeable future. The ready availability of huge amounts of cloud computation power makes enormous offline computations viable, and even inexpensive, and accessible to virtually anyone. However, the devices that people interact with directly are increasingly not PCs, but low-powered mobile devices or appliances. While the computational capabilities of these devices is rapidly improving over time, these capabilities will remain sharply limited compared to those of the cloud. Bringing detailed interactive simulation to these low-powered devices – decoupling simulation speed and detail from the resources available on the simulation platform – requires new simulation techniques.

One such class of simulation techniques are *data-driven simulations*, which use compact representations of simulation state in order to produce detailed interactive simulations. So why does data help?

The reason why we have to choose between speed and detail in most simulations is that most simulation techniques follow a standard rubric based on discretizing continuous equations from physics. We begin with a collection of partial differential equations from physics and discretize them both in time and in space, producing a system of ordinary differential equations. These equations can be integrated to obtain motion. Computing complex dynamics on low-powered devices, interactive performance requires that we

choose a coarse discretization, so that the number of equations to be integrated is small. Since most simulations operate on volumetric or surface data, the number of simulation elements in a given discretization has a quadratic or cubic dependence on the resolution. Coarse discretizations do not produce detailed or accurate motion, though, and increasing the resolution can often drop the simulation speed to slower than interactive rates. Adaptive discretization can partially combat this problem by discretizing finely where the additional detail is noticeable, however, the speed improvements that this produces are usually insufficient to bring an offline-only simulation reliably into real-time range. Choosing good discretizations, then, is not sufficient to achieve real-time simulation of complex phenomena.

Data-driven simulations take advantage of a technique called *model reduction* to avoid this dilemma. Creating detailed, interactive simulations requires algorithms that do not depend on fine discretization in order to provide detail, but instead use compact simulation representations, called *reduced representations*, that encode fine detail without explicitly storing information at high spatial resolution. Model reduction techniques allow simulations to perform their time updates directly on reduced representations, without ever returning to the full representation. Since well-designed reduced representations do not contain much data, they can be fast to process and update – but because they contain less information than a full representation, they must exclude the vast majority of possible simulation states. This exclusion is not necessarily problematic, since users will never experience the vast majority of possible states, but it is critical to choose the reduced representation such that it is capable of representing the simulation outputs that users will be interested in.

There are many possible structures for reduced representations. Most are based on combining or replaying snapshots of simulation states. Possible structures include: linear combinations of simulation snapshots, playback of recorded simulation trajectory snippets, and interpolation of long recorded simulation runs. Selecting a structure is usually not difficult given a good sense of the runtime constraints. However, in the complex simulation scenarios typical of graphics applications, it is not typically possible to find a good set of snapshots and trajectories using analytic methods.

Data-driven simulation techniques obtain these snapshots and trajectories by recording the output of offline training simulations. This empirical method for constructing reduced representations allows them to represent even complex behavior effectively. Larger volumes of training data lead to higher-quality reduced representations and thus more accurate simulations. Data-driven simulations, then, provide not only a way to generate fast, interactive simulations, but also a pathway for using ever-larger amounts of offline cloud computational power to directly improve interactive experiences.

1.1 Contributions

While data-driven simulations can achieve great speed, they come with some limitations that significantly limit their utility for practical applications. In this document, we answer three questions about how these limitations can be addressed:

1. **How can we reconfigure simulation domains at runtime?** One of the most important factors in setting up any simulation is the choice of simulation boundary conditions, such as the arrangement of obstacles in a fluid simulation. Data-driven simulation techniques can easily handle static boundary configurations – even very complex ones – but usually cannot handle dynamically changing boundaries. This inability to handle dynamic boundaries not only prevents the simulation of highly dynamic scenes and of scenes containing deformable objects, but also prevents the construction of

simulation domains larger than or with boundaries different from those contained in the training data. Chapter 3 shows how to overcome this limitation by constructing many simulation tiles, each containing a data-driven simulation, that can be coupled and reconfigured at runtime. By replacing or adding tiles, the boundaries can be rearranged or the simulation domain can be expanded at runtime. The key technical challenge in constructing a simulation from tiles is ensuring that invariants in the simulation dynamics are preserved across tile boundaries. By using a technique called *constraint reduction*, we can guarantee that tiles can be coupled at runtime while enforcing important variants and constraints. We also show how to use apply our modular simulation technique to fluid simulation in order to construct massive, reconfigurable simulations of fluid flow.

2. **How can we simulate a wider variety of phenomena?** Data-driven simulation techniques learn reduced representations for simulation dynamics, and typically rely on model reduction methods to perform the simulation directly on those reduced representations. One popular model reduction technique, Galerkin projection, provides a straightforward technique for converting simulations that use conventionally discretized simulation representations to use reduced representations instead. However, Galerkin projection works only for dynamics that can be represented as systems of polynomial equations. This constraint severely limits the types of simulations to which Galerkin projection can be applied. Many interesting phenomena are non-polynomial, especially in the presence of continuously deformable geometry. These phenomena include fluid dynamics, light transport, and inverse-polynomial force laws such as electromagnetism and gravitation. Chapter 4 describes an extension of Galerkin projection to non-polynomial simulation dynamics, along with applications of this *non-polynomial Galerkin projection* to global illumination and fluid flow in the presence of deforming geometry.

3. **How can we ensure that we select appropriate training data?** While data-driven simulation techniques allow us to find good reduced representations for complex simulations, choosing appropriate training simulations is often more art than science. We want the training data to resemble the simulation output that users will be interested in. This goal presents two major difficulties: first, complex simulation dynamics make it difficult to analyze ahead of time exactly how interesting simulation states will evolve, so while we may be able guarantee that we have captured a large set of interesting states, we cannot guarantee that we have captured all of their successors. Second, since we wish to construct interactive simulations, we cannot predict ahead of time exactly what inputs simulation users will provide. It is crucial to choose these training simulations well, since, if we choose them poorly, our data-driven simulation runtime will produce inaccurate results. Chapter 5 presents *self-refining games*, which record player interaction with a data-driven simulation and use the resulting data to continuously expand and improve the simulation's reduced representation over time. We show that this technique can be used to create a realistic 3D liquid simulation game with high-quality rendering, even on very low-powered hardware.

The answers to these three questions greatly expand data-driven simulation's versatility, greatly expanding the number of simulation applications and the complexity of the scenes to which it can be applied.

The next chapter reviews related work in data-driven simulation. Chapters 3 through 5 answer the three questions posed above, and Chapter 6 closes the document with some concluding thoughts on the current state and future of data-driven simulation.

Chapter 2

Related Work

This chapter provides context and a review of prior work for data-driven and interactive simulation techniques in graphics. §2.1 provides an overview of the types of data-driven simulation used in the graphics literature and §2.2 reviews other, non-data-driven, approaches to real-time simulation of the applications we show for the techniques presented in this document: fluid flow and light transport. Prior works more specifically related to the individual advances we present in this document are discussed in the appropriate chapters.

2.1 Data-Driven Simulation

A variety of different types of data-driven simulation have been applied to simulation problems in graphics. This section reviews some of the most prominent categories of these simulations, their characteristics, and their applications. The simulation techniques differ from each other in two principal ways: their reduced representations, and the methods by which they update their states.

Galerkin projection. Galerkin projection transforms simulation dynamics to operate on linear combinations of simulation snapshots. Typically, the reduced representation used with Galerkin projection is found by running PCA on a large number of simulation state snapshots generated using traditional simulation techniques. In graphics, it has been used to construct simulations of deformable solids [Baraff and Witkin 1992; Barbič and James 2005; Hauser et al. 2003; James and Pai 2002; Pentland and Williams 1989] and fluid flow [Treuille et al. 2006; Gupta and Narasimhan 2007; Barbič and Popović 2008; de Witt et al. 2012]. Outside of graphics, it is frequently used in many engineering applications, typically involving fluid dynamics [Ausseur et al. 2004; Couplet et al. 2005; Holmes et al. 1996; Lumley 1970; Marion and Temam 1989; Rowley et al. 2006; Sirisup et al. 2005; Sirovich 1987], but it has also been used for other purposes, such as control systems [Banks et al. 2000]. In engineering, when combined with the use of PCA to select the simulation subspace, it is also known as proper orthogonal decomposition (POD) or Karhunen-Loève decomposition.

Galerkin projection has historically only provided significant speedups when applied to simulations representible in terms of polynomial equations, or when it is combined with sampling-based approaches to function evaluation. In chapters Chapter 3 and Chapter 4, we extend Galerkin projection to allow Galerkin-projected simulations to be reconfigured and scaled to huge domains (Chapter 3), and to allow them to

accelerate simulation dynamics including division and roots (Chapter 4) without recourse to sampling.

Cubature. Like Galerkin projection, cubature (as a model reduction technique) relies on linear combinations of simulation snapshots to represent simulation state. It differs from Galerkin projection in that it does not transform the simulation equations by directly projecting them onto the new representation, but instead, at each timestep, exactly evaluates the simulation dynamics at a handful of points in the domain and uses them to estimate the value of the simulation state everywhere. Unlike Galerkin projection, cubature is not limited to polynomial simulation dynamics. However, for good accuracy, it requires not only a good selection of simulation snapshots to form its reduced representation (like Galerkin projection does), but also a good selection of points in space at which to evaluate the full simulation dynamics.

Cubature performs well when the simulation dynamics are complex but sparse, and has been used to model nonlinear deformable body dynamics [An et al. 2008] with online updating of the basis [Kim and James 2009] and domain decomposition [Kim and James 2011], thin shell dynamics [Chadwick et al. 2009], and fluid dynamics [Kim and Delaney 2013]. James et al. proposed a method for simulating sound synthesis and propagation that is conceptually similar to cubature [2006].

One substantial disadvantage of both Galerkin projection and cubature is that it is often difficult to place meaningful bounds on the error that a simulation user will perceive in the complex simulation scenarios typical of computer graphics. The self-refining games described in Chapter 5 enable the construction of data-driven simulations with bounded, predictable error. These games use a different underlying reduced representation: *state graphs*.

State graphs. This reduced model consists of a graph of simulation trajectories, which are replayed at runtime to produce fast, interactive simulation. Edges in the graph represent simulation trajectories, and vertices represent choice points, where different trajectories can be selected based on user input. Since the dynamics in this model are discrete, the only dynamics computation required at runtime is selecting which new edge (i.e., recorded simulation trajectory) to follow upon reaching a new vertex of the graph. Consequently, state graph-based models are capable of efficiently representing *any* simulation dynamics.

Kim et al. [2013] demonstrated how a state graph model could be used to represent cloth dynamics, by essentially "unrolling" a motion graph through simulation to form the state graph. Chapter 5 demonstrates the efficient construction of state graphs for liquid simulation, informed by user input, which allows us to guarantee that the data in these graphs is relevant to the simulation states that users will actually want to see.

Other approaches. Of course, a wide variety of different reduced representations are possible, beyond those mentioned earlier in this section. Raveenendran et al. [2014] present a method for for interpolating liquid simulations parameterized by a low number of degrees of freedom, allowing many variations of a liquid simulation to be created from a coarse sampling of the parameter space. It is also possible to use data to assist in mapping from coarsely discretized underlying simulation representations to finely detailed output. For example, Chai et al. [2014] proposed a coarsely discretized hair simulation model with a data-driven refinement step to generate detailed hair motion.

2.2 Applications

This section discusses prior implementations of two major types of simulations that we use as sample applications in this document: fluid flow and global illumination.

2.2.1 Fluid Simulation

Fluid simulation in computer graphics has focused on three basic fluid representations: grid and mesh-based Eulerian simulations, meshless Lagrangian methods, and model reduction. An important step in Eulerian fluid simulation was the introduction of an unconditionally stable advection step by Stam [1999]. Improvements to this method based on hierarchical space decomposition [Losasso et al. 2004] and non-uniform meshes [Elcott et al. 2005; Feldman et al. 2005a] have produced impressive results, but do not typically allow real-time simulation. A number of Eulerian methods have been mapped to the GPU, including stable advection [Wu et al. 2005], pressure projection [Bolz et al. 2003; Goodnight et al. 2003; Krüger and Westermann 2003], Lattice Boltzmann methods [Li et al. 2003], and liquids using a variety of representations: regular grids [Crane et al. 2007], heightfields (shallow-water methods) [Chentanez and Müller 2010; Št'ava et al. 2008; Thurey et al. 2007], and hybrid techniques [Chentanez and Müller 2011]. These implementations enable real-time performance for medium-sized fluid domains. However, unlike the data-driven techniques we discuss in this document, they fundamentally have the same time complexity as their CPU variants.

Lagrangian particle representations such as vortex methods [Angelidis and Neyret 2005; Angelidis et al. 2006; Park and Kim 2005; Pfaff et al. 2012; Selle et al. 2005] and smoothed particle hydrodynamics [Adams et al. 2007; Keiser et al. 2005; Müller et al. 2003; Solenthaler and Pajarola 2009] do not depend on grid resolution, but the effective resolution of the fluid depends on the particle density, and detailed results are typically not feasible for real-time use. SPH has also been demonstrated to be feasible for real-time fluid simulation at moderate resolutions [Goswami et al. 2010; Macklin and Müller 2013]. However, significant computational power is still necessary to obtain detailed results.

A variety of hybrid methods also exist, perhaps the most popular being FLIP [Ando et al. 2013; Zhu and Bridson 2005], which performs advection using marker particles but solves the pressure equation on a grid, sharing the stability of grid-based methods with the simple advection and fine detail resolution of particle-based techniques. Other hybrid methods include Eulerian velocities with vortex particles [Golas et al. 2012], and a FLIP-spectral hybrid that enables the use of very large grid cells by representing pressure and velocity fields using high-order polynomials in each cell [Edwards and Bridson 2014].

Data-driven techniques for fluid simulation were introduced to graphics by Treuille et al. [2006]. Gupta and Narasimhan [2007] used a reduced-space representation to accelerate the simulation and rendering of dynamic participating media, such as smoke and snow, and Popović and Barbič [2008] used data-driven methods to control fluid dynamics. de Witt et al. [2012] suggested improved methods for basis construction for real-time simulation.

2.2.2 Global Illumination

Accurate illumination of dynamic scenes has been a long-standing goal in computer graphics. Existing reduced-order global illumination techniques, such as Precomputed Radiance Transfer [Sloan et al. 2002], cannot accurately model the effect of general continuous large-scale scene deformations on nonlocal radiance transfer. James and Fatahalian [2003a] allowed for deformable objects, but only for a certain set of physics-based deformations. Sloan et al. [2005] accounted for changes in local light transport arising from a more general set of deformations. More recent models allow discrete scenes to be composed without requiring additional precomputation [Loos et al. 2011; Loos et al. 2012], but do not allow for general continuous change in shape. Numerous methods [Hanrahan et al. 1991; Drettakis and Sillion 1997] have

been proposed to accelerate the computational speed of radiosity for dynamic scenes. However, most previous methods are limited to rigid transformations, such as inserting, deleting, and moving objects in the scene.

The application of data-driven techniques to global illumination is also not unprecedented. Many techniques have seen widespread use, including ambient occlusion [Zhukov et al. 1998; Ren et al. 2006], virtual point lights [Keller 1997], and hierarchical methods [Hanrahan et al. 1991; Durand et al. 1999]. However, these methods all have limitations: ambient occlusion accounts only for very local effects, achieving low error in complex scenes rapidly becomes very expensive with virtual point lights, and hierarchical methods do not handle large changes in the scene geometry well. There are also many reduced-order techniques for global illumination, such as Precomputed Radiance Transfer [Sloan et al. 2002; Sloan et al. 2005], however, they cannot accurately model the effect of general continuous large-scale scene deformations on nonlocal radiance transfer.

Chapter 3

Modular Simulations With Constraint Reduction

This chapter shows how data-driven simulations can be extended to reconfigurable domains on huge scales. For example, consider computing the detailed flow of wind through an entire city. Training simulations for a domain as extensive as an entire city would be prohibitively expensive, and would fix the simulation domain such that buildings could not be added, deleted, or moved after the fact. By allowing large simulations to be composed from multiple smaller simulations, it becomes possible to run large, interactive simulations that can be reconfigured at runtime.

This chapter uses fluid simulation on large domains as a motivating example. Large data-driven simulations are constructed from modular simulation *tiles* that capture fluid behavior given specific boundary conditions such as the presence of an obstacle. Each tile contains a data-driven liquid simulation, implemented using Galerkin projection (§2.1,§3.2) according to the method of Treuille et al. [2006] from high-resolution training data. Like most data-driven simulations, the simulations in these tiles are very fast and have runtime complexity independent of the grid resolution. Tiles can be assembled at runtime to simulate novel fluid configurations, and §3.7 shows results demonstrating that such tilings can scale to very large domains. We show that simulation operators can be precomputed, decomposed, and reconfigured at runtime based on the tiling, thus enabling novel tile configurations without additional precomputation. Simulation is fast because the dynamics operate entirely on the reduced representations of the data-driven simulations. We also show that the tiles can be constructed so as to maintain pairwise consistency between adjacent tiles at runtime.

The main technical difficulty we encounter in presenting this method is treating tile coupling correctly. In general, data-driven simulations have so few degrees of freedom that maintaining consistency quickly over-constrains the system. This can lead to severe artifacts at tile boundaries and unnatural behavior in the interior of the coupled tiles. We address this problem by introducing *constraint reduction*, an algorithm that modifies fluid tiles so that they can exactly fulfill a large number of linear constraints in the full-dimensional space. We show that this technique generalizes to arbitrary linear constraints; like simulation, constraint satisfaction can be solved entirely in the reduced space. These techniques enable flexible assembly of complex simulations on a scale previously unattainable in computer graphics.

Before discussing the details of modular simulations and our constraint reduction technique, this chapter presents a review of other work in creating modular data-driven simulations (§3.1) and of Galerkin

projection (§3.2), the data-driven simulation technique that forms the foundation for this chapter and the next.

3.1 Related Work

Our method is based on constructing reconfigurable fluid simulation tiles. Perhaps the best known examples of reconfigurable tiles in computer graphics are Wang tiles [Cohen et al. 2003] which can be arranged to produce non-periodic textures or complex geometric scenes. Similarly, for fluids, Chenney [2004] introduces flow tiles, which produce divergence-free flows so long as the tiles are appropriately combined. In contrast to flow tiles, which are static vector fields, our method enables dynamic simulation.

Our tile representation requires explicit consistency constraints. A similar need to maintain consistency across simulation domains is encountered in finite element simulations of fluids, where the technique is called *domain decomposition*. To enforce coupling constraints arising on the boundaries between subdomains, one can add penalty terms [Farhat et al. 2001; Farhat et al. 2003; Tezaur et al. 2008; Zhang et al. 2006], or enforce strict compliance via Lagrange multipliers [Babuška 1973; Farhat et al. 2000; Tezaur and Farhat 2006]. Toselli and Widlund [2005] provide a good overview of these techniques. Finite element methods use analytic basis functions which are specifically designed such that boundary constraints can be satisfied. By contrast, the basis vectors used in our model are more expressive, but do not satisfy coupling constraints by construction.

To our knowledge, there is little work on coupling model reduced fluid simulations. LeGresley and Alonso [2003] decompose the simulation domain into a model reduced fluid simulation but use full resolution fluid simulation to capture fine scale features in certain areas. However, this work does not address the case of coupling two separately computed reduced simulations. Borggaard et al. [2006] tackle the problem of performing singular value decomposition (SVD) on a very large fluid simulation spatially partitioned across a set of processors. Their intent is to produce a single basis from this data, not a set of coupled reduced models. Perhaps the closest work to our own is that of Lucia and King [2002], who perform domain decomposition to isolate regions that contain shockwaves and then combine a set of model reduced simulations in order to capture shockwaves in high-speed flow fields. They use a penalty-based method to enforce continuity across boundaries, but do not address the problem of creating bases that can be spatially reconfigured while fulfilling continuity constraints at runtime.

Domain decomposition has been applied to other model-reduced data-driven simulations in graphics. In particular, several works in graphics have coupled model-reduced simulations of deformable objects. Barbič and Zhao [2011] built reduced models for objects with tree-like topologies, and Zhao and Barbič [2013] used this technique to quickly construct reduced models of plants. Kim and James [2011] used penalty forces to couple multiple components of character models.

Contrary to existing work on reduced fluid models which relies on SVD to compute the basis, we use SVD-based models as a starting point, and then modify the basis to enable coupling of previously incompatible tiles. Our technique uses the data-driven simulation technique Galerkin projection as its foundation, so we review its basic operation in the next section.

3.2 Galerkin Projection

Galerkin projection is a popular data-driven simulation technique that translates polynomial dynamics directly into a chosen linear subspace. Since its mechanics are fundamental to the next two sections, we present here a basic overview of its operation.

Suppose we wish to evaluate a function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ where the input $\mathbf{x} \in \mathbb{R}^n$ and output $\mathbf{y} \in \mathbb{R}^m$ are very high-dimensional. We seek a reduced approximation $\hat{\mathbf{y}} = \hat{\mathbf{q}}(\hat{\mathbf{x}})$, where the reduced input $\hat{\mathbf{x}} \in \mathbb{R}^{\hat{n}}$ and output $\hat{\mathbf{y}} \in \mathbb{R}^{\hat{m}}$ lie in much lower-dimensional spaces: $\hat{n} \ll n$ and $\hat{m} \ll m$. The first step is to linearly *dimension-reduce* the state vectors, which means finding a pair of orthonormal bases $\mathbf{B}_x, \mathbf{B}_y$ which convert reduced vectors to full vectors: $\mathbf{x} = \mathbf{B}_x\hat{\mathbf{x}}$ and $\mathbf{y} = \mathbf{B}_y\hat{\mathbf{y}}$. We can project from the full to the reduced space through multiplication by the transpose: $\hat{\mathbf{x}} = \mathbf{B}_x^T\mathbf{x}$, and $\hat{\mathbf{y}} = \mathbf{B}_y^T\mathbf{y}$. The second step is to *model-reduce* the transformation $\mathbf{f}$, which means finding an efficient reduced approximation $\hat{\mathbf{q}} : \mathbb{R}^{\hat{n}} \to \mathbb{R}^{\hat{m}}$ operating entirely in the reduced space. The standard approach is *Galerkin projection*, which works well if $\mathbf{f}$ is polynomial but can be inefficient otherwise.

We will address these steps in order: first, we will sketch the process of basis construction, and then we will discuss how to use the bases to reduce simulation dynamics. We limit the class of reduced functions to the polynomials, since that limitation is inherent to traditional Galerkin projection. In Chapter 4 we discuss our work on lifting this limitation.

Notation. In the remainder of this document, scalars appear in lower case: x, vectors in bold lower case: $\mathbf{x}$, and matrices and tensors in bold upper case: $\mathbf{X}$. We write $\mathbf{Q} \otimes_a \mathbf{M}$ to denote tensor multiplication of the tensor $\mathbf{Q}$ by the matrix or vector $\mathbf{M}$ along the axis with index a, and $\mathbf{Q} \otimes_{a\ldots b} \mathbf{M}$ to denote the repeated tensor product $\mathbf{Q} \otimes_a \mathbf{M} \otimes_{a+1} \mathbf{M} \ldots \otimes_b \mathbf{M}$. We number tensor axes starting from 0. For clarity, we sometimes employ the matrix notation $\mathbf{B}^T\mathbf{Q}$ to denote multiplication along the zeroth axis, $\mathbf{Q} \otimes_0 \mathbf{B}$, and the notation $\mathbf{QB}$ to denote multiplication along the first axis, $\mathbf{Q} \otimes_1 \mathbf{B}$. If $\mathbf{Q}$ is a matrix, this notation preserves the usual meaning of $\mathbf{B}^T\mathbf{Q}$ and $\mathbf{QB}$. We refer to multiplication of a tensor by a vector as *tensor contraction*, an operation which reduces the tensor order by one.

3.2.1 Basis Construction

As mentioned above, to reduce a function $\mathbf{y} = \mathbf{f}(\mathbf{x})$, we require bases $\mathbf{B}_y$ and $\mathbf{B}_x$ for $\mathbf{y}$ and $\mathbf{x}$. We build these bases from snapshots of example states. To build bases for the dynamics $\mathbf{y} = \mathbf{f}(\mathbf{x})$, we collect a set of representative inputs $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]$, then evaluate $\mathbf{f}(\mathbf{x})$ for each of these inputs to obtain a set of representative outputs $\mathbf{Y} = [\mathbf{y}_1 \ldots, \mathbf{y}_n]$. We then construct the bases $\mathbf{B}_x$ and $\mathbf{B}_y$ using PCA. If we desire bases of ranks k_x and k_y, then $\mathbf{B}_x$ is the first k_x principal components of $\mathbf{B}_x$, and $\mathbf{B}_y$ is the first k_y principal components of $\mathbf{B}_y$.

This basis construction method has a number of important consequences. First, since the accuracy of the reduced dynamics is limited by the expressive power of the bases, it is critical that the example inputs $[\mathbf{x}_1, \ldots, \mathbf{x}_n]$ be chosen carefully. Second, since the method is snapshot-based, the dynamics are reduced *monolithically*: the system is treated as a whole, even if it may have recognizably loosely-coupled parts. This means that very large systems, or systems with a large range of behaviors, may require very large bases in order to give good performance.

3.2.2 Polynomial Galerkin Projection

The next step is to restate the simulation dynamics in terms of these bases. Traditional Galerkin projection is only efficient when applied to polynomial functions, so we will assume that our dynamics are given by a polynomial $\mathbf{y} = \mathbf{q}(\mathbf{x})$. A degree-$d$ polynomial $\mathbf{q}(\mathbf{x})$ can be represented as a $d+1$th-order tensor $\mathbf{Q}$. For example, if some polynomial $\mathbf{p}(\mathbf{x})$ has degree 3 and $\mathbf{x}$ is of length n, then we can write the components of $\mathbf{p}(\mathbf{x})$ as

$$p_i(\mathbf{x}) = \sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{\ell=1}^{n} c_{ijk\ell} x_j x_k x_\ell$$

where $c_{ijk\ell}$ are the coefficients of the polynomial $\mathbf{p}(\mathbf{x})$. The 4th-order tensor $\mathbf{P}$ simply consists of the polynomial coefficients: $P_{ijk\ell} = c_{ijk\ell}$, and we can write $\mathbf{p}(\mathbf{x}) = \mathbf{P} \otimes_{1\ldots3} \mathbf{x}$. Likewise, we can evaluate the degree-d polynomial $\mathbf{q}(\mathbf{x})$ by contracting its corresponding $d+1$th-order tensor $\mathbf{Q}$:

$$\mathbf{q}(\mathbf{x}) = \mathbf{Q} \otimes_{1\ldots d} \mathbf{x}.$$

To compute the Galerkin projection of the polynomial

$$\mathbf{y} = \mathbf{Q} \otimes_{1\ldots d} \mathbf{x}$$

we start by substituting the reduced variables:

$$\mathbf{B}_y \hat{\mathbf{y}} \approx \mathbf{Q} \otimes_{1\ldots d} \mathbf{B}_x \mathbf{x}$$

(We write $\approx$ to remind ourselves that $\hat{\mathbf{y}}$ has too few degrees of freedom to ensure that this equation has an exact solution.) We then multiply by $\mathbf{B}_y^T$:

$$\hat{\mathbf{y}} = \mathbf{B}_y^T \left(\mathbf{Q} \otimes_{1\ldots d} \mathbf{B}_x \mathbf{x}\right). \tag{3.1}$$

We can compute the value of $\hat{\mathbf{y}}$ at runtime quickly as $\hat{\mathbf{y}} = \hat{\mathbf{Q}} \otimes_{1\ldots d} \hat{\mathbf{x}}$, where

$$\hat{\mathbf{Q}} = \mathbf{B}_y^T \mathbf{Q} \otimes_{1\ldots d} \mathbf{B}_x, \tag{3.2}$$

and the product $\mathbf{B}_y^T\mathbf{Q}$ denotes the product of $\mathbf{Q}$ with $\mathbf{B}_y$ along the tensor axis corresponding to the result vector $\hat{\mathbf{y}}$. $\hat{\mathbf{Q}}$ is called the *Galerkin projection* of $\mathbf{Q}$. Intuitively, $\hat{\mathbf{Q}}$ transforms the reduced input $\hat{\mathbf{x}}$ into the full space using $\mathbf{B}_x$, applies $\mathbf{Q}$, then projects back into the reduced space using $\mathbf{B}_y^T$. The final projection incurs some accuracy cost, but it unambiguously specifies $\hat{\mathbf{y}}$, so we can write Eq. 3.2 with an equals sign. The projection is fast: evaluating $\hat{\mathbf{Q}}$ takes the same number of contractions as evaluating $\mathbf{Q}$, although each contraction is now with a vector of length $\hat{n}$ instead of length n.

3.3 Fluid Tiles

The central idea of the modular approach is to cover the simulation domain with a small set of simulation primitives, called *tiles*. Each tile consists of a velocity basis representing the possible flow within its subdomain. For example, if one tile represents the fluid flow around the Empire State Building, then its basis spans a subset of possible flows around this obstacle. The boundaries between subdomains correspond to tile *faces*. Associating a small set of *boundary bases* with the tile faces allows us to satisfy constraints that cross them. As long as adjacent faces share the same boundary basis, this construction guarantees that all constraints within the tiling can be satisfied.

3.3.1 Monolithic Fluid Reduction

In this algorithm, each tile corresponds to a spatially-localized linear model of fluid velocities. In this section, we briefly review the necessary basics of data-driven fluid simulation using Galkerin projection, and refer the reader to [Treuille et al. 2006] for more details.

First, consider a simulation with only one tile. This is simply a standard simulation implemented using Galerkin projection, where the entire domain is spanned by a single velocity basis. The full-dimensional simulation state is represented by a vector $\mathbf{u} \in \mathbb{R}^N$ consisting of the velocities defined at sample points. The reduced order model operates in an m-dimensional space spanned by basis states $\mathbf{B} = [\mathbf{b}_1 \dots \mathbf{b}_m]$.

As demonstrated in [Treuille et al. 2006], the Navier-Stokes equations can be reduced to

$$\frac{d\hat{\mathbf{u}}}{dt} = \left(\mu\hat{\triangle} + \sum_i \hat{\mathbf{A}}_i \hat{u}_i \right) \hat{\mathbf{u}}, \tag{3.3}$$

where μ is the diffusion coefficient, $\hat{\triangle}$ denotes the reduced diffusion operator, and $\hat{\mathbf{A}}_i$ is a reduced linear operator that advects $\hat{\mathbf{u}}$ using the velocity field $\mathbf{b}_i$. To render the advection operator amenable to Galerkin projection, consider the matrices $\hat{\mathbf{A}}_i$ as slices of an *advection tensor* $\hat{\mathbf{A}}$ of order three:

$$\frac{d\hat{\mathbf{u}}}{dt} = \left(\mu\hat{\triangle} + \hat{\mathbf{A}} \otimes_2 \hat{\mathbf{u}} \right) \hat{\mathbf{u}}. \tag{3.4}$$

Because the basis $\mathbf{B}$ spans only divergence-free velocity fields, incompressibility need not explicitly be enforced, unlike traditional Eulerian fluid simulations.

Treating the advection velocities as constant throughout each time step, Eq. 3.4 can be integrated analytically, leading to an integration that is both unconditionally stable and energy-preserving if the fluid is inviscid. Given a time step Δt, the next reduced state is computed as

$$\hat{\mathbf{u}}_{t+\Delta t} = e^{\Delta t\left(\mu\hat{\triangle} + \hat{\mathbf{A}} \otimes_2 \hat{\mathbf{u}}\right)} \hat{\mathbf{u}}_t. \tag{3.5}$$

This matrix-vector product can be computed using iterative Taylor or Padé approximation, making the integration fast even for high-dimensional reduced models.

3.3.2 Tiled Fluid Reduction

The monolithic reduction described above is fast but does not allow for domain reconfiguration. Even small changes to the simulation domain require complete recomputation of the model. Replacing the monolithic basis with a modular set of tiles that can be assembled at runtime enables reconfigurability through tile replacement and rearrangement. These tiles are obtained through decomposing the simulation domain. The fluid flow within each subdomain is computed using a standard Galerkin projected fluid simulation.

Fig. 3.1 shows an illustration of domain decomposition. Consider a discretized domain split into two parts A and B as shown in Fig. 3.1. The Navier-Stokes equations include the constraint that the divergence must be zero across the simulation domain. This constraint must still be enforced in the split domain. If cells are split by the decomposition, the divergence constraints lead to constraints involving all adjacent tiles (see Fig. 3.1, left). If the decomposition splits the domain such that velocities normal to the interface

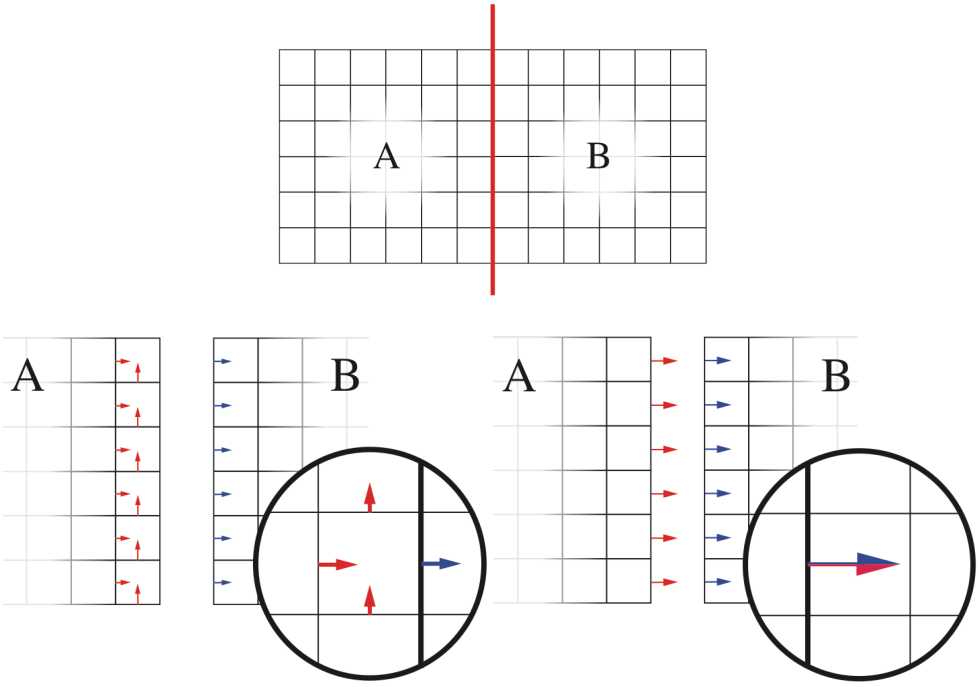


Figure 3.1: *Domain decomposition: Left: Decomposition into two domains, inducing divergence constraints in cells split between A and B. Right: Duplication of boundary values leads to equivalent equality constraints for duplicated values. The divergence of the cells can be enforced in each tile separately.*

are defined on the boundary, then boundary velocities can also be duplicated, as shown in Fig. 3.1, on the right. In this case, the aforementioned divergence constraints can be enforced in each tile separately. In order for the discretization to be consistent, the velocities twice defined on the boundary need to be equal, leading to equality constraints. Both interpretations are equivalent. The following sections will use the second convention, since it simplifies that algorithmic description. We will now discuss how to compute tensors for a tiled domain, before returning to the contraints in §3.4.

Computing the necessary tensors for simulation using tiles can be treated as a special case of standard, monolithic Galerkin projection. Consider two Galerkin-projected simulations A and B corresponding to domains $\mathcal{D}_A$ and $\mathcal{D}_B$ as in Fig. 3.1. A reduced basis $\mathbf{B}_A$ of A covers only $\mathcal{D}_A$, and it can be considered to be zero everywhere else. The same holds for $\mathbf{B}_B$. The two bases can be combined into a basis of size $m = m_A + m_B$:

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_A & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_B \end{bmatrix}. \tag{3.6}$$

Then, the state of the complete system can be written as $\hat{\mathbf{u}} = [\hat{\mathbf{u}}_A^T, \hat{\mathbf{u}}_B^T]^T$. Diffusion and advection operators can now be computed as before. In particular, for the basis in equation (3.6), the diffusion operator becomes

$$\triangle = \begin{bmatrix} \triangle^{AA} & \triangle^{AB} \\ \triangle^{BA} & \triangle^{BB} \end{bmatrix}, \tag{3.7}$$

where the *interior* terms $\triangle^{AA}$ and $\triangle^{BB}$ depend only on $\mathbf{B}_A$ or $\mathbf{B}_B$, respectively, while the *coupling* terms $\triangle^{AB}$ and $\triangle^{BA}$ depend on both $\mathbf{B}_A$ and $\mathbf{B}_B$.

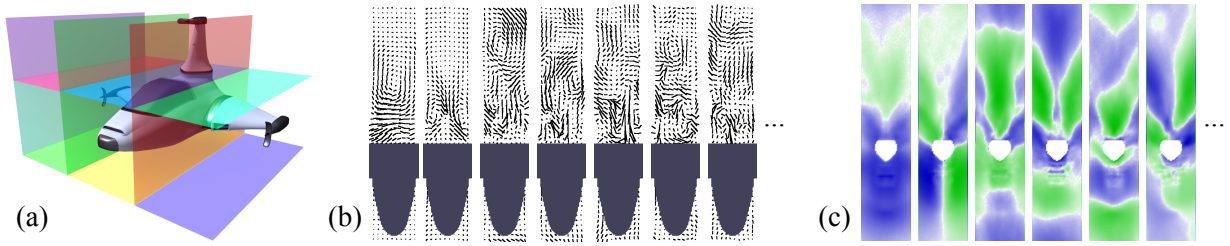


Figure 3.2: *(a) Decomposition of a spacecraft model. The domain is split into six subdomains. Two parts are empty, while the other subdomains contain the wings, the body and the tail, respectively. (b) Slices through the velocity basis of the spacecraft body. (c) Basis vectors from the boundary basis between body and tail of the spacecraft. Blue and green represent positive and negative flow across the boundary.*

The situation for the advection tensor $\hat{\mathbf{A}}$ is similar. However, since $\hat{\mathbf{A}}$ is a rank three tensor, it has eight components $\hat{\mathbf{A}}^{AAA}, \hat{\mathbf{A}}^{AAB}, \ldots, \hat{\mathbf{A}}^{BBB}$. The six blocks with mixed superscripts are coupling terms. Because the full-dimensional diffusion and advection operators are sparse and highly localized in space, the cost of computing the coupling terms is proportional only to the size of the interface, not the full dimension N.

This construction can be generalized to arbitrary tilings of the domain. Because the adjacency graph between spatial subdomains is sparse, the combined advection and diffusion operators are also block-sparse.

3.4 Constraints

As described in 3.3.2, velocities are defined twice along the interface $\mathcal{F}$ between adjacent subdomains $\mathcal{D}_A$ and $\mathcal{D}_B$. Keeping the simulation consistent between tiles requires the enforcement of equality constraints

$$\mathbf{D}_A\mathbf{u}_A = \mathbf{D}_B\mathbf{u}_B \tag{3.8}$$

where $\mathbf{D}_A$ and $\mathbf{D}_B$ are matrices selecting only the elements of $\mathbf{u}_A$ and $\mathbf{u}_B$ that lie in $\mathcal{F}$. These constraints can be assembled into a constraint matrix $\mathbf{C}$ which is satisfied when $\mathbf{CB}\hat{\mathbf{u}} = 0$. The system is overconstrained as soon as the number of samples in $\mathcal{F}$, N_b, exceeds the number of combined degrees of freedom $m_A + m_B$. Solving for $\hat{\mathbf{u}}$ will therefore yield the trivial solution $\hat{\mathbf{u}} = 0$. A naïve solution turns these constraints into a penalty by solving for the updated state $\hat{\mathbf{u}}'$ that minimizes

$$\|\mathbf{CB}\hat{\mathbf{u}}'\|^2 + \alpha\|\hat{\mathbf{u}} - \hat{\mathbf{u}}'\|^2. \tag{3.9}$$

The regularization parameter α balances between constraint satisfaction and state modification. Large values of α allow inconsistent boundary velocities, leading to serious simulation artifacts. On the other hand, as $\alpha \to 0$, large corrections are applied to achieve an admissible state. If the tiles are too different, this leads to locking: only a very low-dimensional subspace of states can be represented by adjacent tiles, and the simulation will be *locked* into this subspace.

3.4.1 Constraint Reduction

To solve this problem, we introduce the *constraint reduction* method. This method modifies the basis vectors $\mathbf{B}$ to allow exact constraint satisfaction while preserving sufficient degrees of freedom for the

simulation.

Consider again the two adjacent tiles A and B. The mixed constraints between those tiles (i.e. the constraints depending on values from both A and B) are the equality constraints Eq. 3.8. These constraints can be written as:

$$\mathbf{CB}\hat{\mathbf{u}} = \mathbf{D}_A\mathbf{B}_A\hat{\mathbf{u}}_A - \mathbf{D}_B\mathbf{B}_B\hat{\mathbf{u}}_B = \mathbf{e}_A - \mathbf{e}_B = \mathbf{0}. \tag{3.10}$$

Both $\mathbf{e}_A$ and $\mathbf{e}_B$ are of dimension N_b, the number of velocity samples on the interface. If $\mathbf{e}_A$ and $\mathbf{e}_B$ lie in the same linear space $\mathcal{S}$ with low dimension $s < m$, only s degrees of freedom are needed to fulfill the constraints. This condition can be enforced for all basis vectors. To achieve this goal, construct a new basis $\tilde{\mathbf{B}}_A = [\tilde{\mathbf{b}}_1^A, \ldots, \tilde{\mathbf{b}}_{m_A}^A]$ such that

$$\mathbf{D}_A\tilde{\mathbf{b}}_i^A \in \mathcal{S} \qquad \forall i \in \{1 \ldots m_A\}, \tag{3.11}$$

and similarly for $\mathbf{B}_B$. In this case, the space $\mathcal{S}$ represents the allowed boundary states at the interface between A and B.

$\mathcal{S}$ is constructed as a low-dimensional model of observed boundary states. The boundary values of all bases that should be made compatible are extracted and stored in a database $\mathbf{E}$. In this example involving only two bases, the database is $\mathbf{E} = [\mathbf{D}_A\mathbf{b}_{1\ldots m_A}^A, \mathbf{D}_B\mathbf{b}_{1\ldots m_B}^B]$. Next, given $\mathbf{E}$, an $N_b \times s$ *boundary basis* can be computed for $\mathcal{S}$: $\mathbf{S} = \mathrm{pca}_s\{\mathbf{E}\}$, where pca_s takes the s leading singular vectors of its argument. See Fig. 3.2 (c) for an example. Often, more than two tiles share a compatible boundary. In these cases, $\mathbf{E}$ is assembled from all bases involved, or form a different set of examples representative of the flow patterns across the boundary.

Given the boundary basis, modified bases $\tilde{\mathbf{B}}_A$ and $\tilde{\mathbf{B}}_B$ can be computed as follows. For each basis vector, first project each of its faces into the appropriate boundary basis. Since this breaks the zero-divergence constraints inside the domain, the next step is to fix the boundaries and find the closest divergence-free field given the boundary conditions by Helmholtz-Hodge decomposition. Finally, reorthonormalize the basis using the standard Gram-Schmidt process.

This modified basis now satisfies Eq. 3.11, and so the constraint satisfaction problem can be solved. Since the constraint violations $\mathbf{e}_A$ and $\mathbf{e}_B$ lie in $\mathcal{S}$, they can be represented using a basis for $\mathcal{S}$ by rewriting Eq. 3.10 as

$$\mathbf{S}^T\mathbf{C}\tilde{\mathbf{B}}\mathbf{r} = \mathbf{M}\mathbf{r} = \mathbf{0}, \tag{3.12}$$

where the constraints are recombined into one system $\mathbf{C}$ using the combined modified basis $\tilde{\mathbf{B}}$. $\mathbf{M}$ is an $s \times m$ matrix, revealing the effective dimension of the constraints applied to the modified basis. Note that the constraints have not been compromised — all constraints are fulfilled exactly. Instead, the bases have been modified in order to allow for exact constraint satisfaction, potentially losing optimal reconstruction properties of the PCA-based construction.

A state that satisfies the constraints can now be found by solving Eq. 3.12. m and s are chosen such that enough degrees of freedom are left even if a tile is constrained from all sides. In a three dimensional tiling of space, there will be three constrained boundaries per tile. For m dimensions per tile and s dimensional boundary bases, it must be the case that $m > 3s$ to avoid locking. When building large scenes from many coupled tiles, the reduced constraint matrix $\mathbf{M}$ is block-sparse as only adjacent tiles have non-zero entries.

3.5 General Constraints

Before turning to the algorithmic details in §3.6, we discuss the case of general linear constraints. Assume a set of linear constraints involving a number of reduced models A_i with bases $\mathbf{B}_{A_i}$:

$$\sum_i \mathbf{D}_{A_i}\mathbf{B}_{A_i}\hat{\mathbf{x}}_{A_i} = \mathbf{0}. \tag{3.13}$$

As before, each basis $\mathbf{B}_{A_i}$ can be modified such that each $\mathbf{D}_{A_i}\mathbf{B}_{A_i}\hat{\mathbf{x}}_i$ lies in a small space $\mathcal{S}$ with dimension $s < m_{A_i}$. Note that the constraints do not need to be spatially localized or sparse. Potentially, all components of $\mathbf{B}_{A_i}\hat{\mathbf{x}}_{A_i}$ could be referenced in each constraint. However, it is crucial that the bases $\mathbf{B}_{A_i}$ can be modified such that the constraint violations for each reduced model A_i lie in a small subspace $\mathcal{S}$. In the general setting, the next step is to find bases $\tilde{\mathbf{B}}_{A_i} = [\tilde{\mathbf{b}}_1^{A_i} \dots \tilde{\mathbf{b}}_m^{A_i}]$ such that

$$\mathbf{D}_{A_i}\tilde{\mathbf{b}}_j^{A_i} \in \mathcal{S} \quad \forall j \in \{1 \dots m_{A_i}\}, \tag{3.14}$$

while minimizing the distortion to the bases:

$$\|\tilde{\mathbf{b}}_j^{A_i} - \mathbf{b}_j^{A_i}\|. \tag{3.15}$$

Note that $\mathbf{D}_{A_i}$ can include constraints only affecting a single basis $\mathbf{B}_{A_i}$, such as zero-divergence constraints.

The two-step technique described in §3.4.1 finds an approximate minimum of Eq. 3.15 while fulfilling the constraints exactly. Since the modifications to each basis vector are typically small, not exactly finding the global minimum does not lead to noticeable artifacts.

3.6 Algorithmic Details

Algorithm 1 summarizes the necessary steps to set up a tiled simulation. For each of the t tiles, first compute n_e examples by taking snapshots of a full simulation within the domain of the tile which has N degrees of freedom (line 2). Then, distill these examples into a raw basis of dimension m as described in §3.3.1 (line 4). Using a sampling-based approach, the principal components can be computed in time linear in n.

The boundary bases are computed in a similar fashion. Extract the boundary velocities relevant to each boundary type j from the raw bases and collect these boundary states in a matrix $\mathbf{E}_j$ which is of size $N_b \times \mathcal{O}(tm)$, where N_b is the number of samples in a boundary face. Then, use PCA to extract a boundary basis S_j of dimension m_b (line 7).

The raw bases and boundary bases are then combined into tiles. For each raw basis $\mathbf{B}_i$, choose boundary bases for each face that will be coupled to other tiles. Given these boundary bases, constraint reduction can be applied to compute a modified basis $\tilde{\mathbf{B}}_i$ (line 9). As described in §3.4.1, this process requires solving a sparse linear system for each basis vector, yielding a total cost of $\mathcal{O}(mN^{4/3})$ using conjugate gradients[1].

Finally, compute advection and diffusion operators for the tile bases $\tilde{\mathbf{B}}_i$ (line 11). Computing the interior advection tensors $\hat{\mathbf{A}}^{AAA}$ for each tile dominates the computational cost. The interior diffusion tensors

[1]The cost for solving a linear system of size N representing a 3D finite difference discretization of an elliptic boundary value problem can be solved in $\mathcal{O}(N^{4/3})$ time using conjugate gradients [Shewchuk 1994]. The systems treated herein are of this type.

	// Decompose domain, choose basis dimension m	
1	**forall the** *raw bases* $\mathbf{B}_i$ **do**	
2	Compute examples $\mathbf{U}_{\mathbf{B}_i}$	$\mathcal{O}(n_e N^{4/3})$
3	Compute raw basis $\mathbf{B}_i = \mathrm{pca}_m\{\mathbf{U}_{\mathbf{B}_i}\}$	$\mathcal{O}(n_e^2 N + n_e^3)$
4		
5	**forall the** *boundary types* $\mathbf{S}_j$ **do**	
	// Collect boundary states $\mathbf{E}_j$ *from all relevant* $\mathbf{B}_i$*, choose* s	
6	Compute boundary basis $\mathbf{S}_j = \mathrm{pca}_s\{\mathbf{E}_j\}$	$\mathcal{O}((tm)^2 N_b + (tm)^3)$
7		
8	**forall the** *tiles* k **do**	
	// Choose basis i *and boundary types* $j_1 \dots j_6$	
9	Compute modified basis $\tilde{\mathbf{B}}_k$	$\mathcal{O}(mN^{4/3})$
10	Compute interior tensor blocks $\hat{\mathbf{A}}^{AAA}$ and $\hat{\triangle}^{AA}$	$\mathcal{O}(m^3 N)$
11		
12	**forall the** *pairs of tiles* $(\tilde{\mathbf{B}}_1, \tilde{\mathbf{B}}_2)$ **do**	
13	Compute coupling tensors between $\tilde{\mathbf{B}}_1$ and $\tilde{\mathbf{B}}_2$	$\mathcal{O}(m^3 N_b)$

Algorithm 1: *Tile creation from examples.*

1	**if** *not initialized or connectivity changed* **then**	
2	Assemble tensors $\hat{\mathbf{A}}$ and $\hat{\triangle}$	$\mathcal{O}(k)$
3		
4	Contract advection tensor $\hat{\mathbf{M}}_A = \hat{\mathbf{A}} \otimes_2 \hat{\mathbf{u}}$	$\mathcal{O}(km^3)$
5	Assemble $\hat{\mathbf{M}} = \Delta t(\hat{\mathbf{M}}_A + \mu\hat{\triangle})$	$\mathcal{O}(km^2)$
6	Compute preliminary state $\hat{\mathbf{u}}' = e^{\hat{\mathbf{M}}}\hat{\mathbf{u}}$	$\mathcal{O}(km^2)$
7	Project state: solve Eq. 3.9 for new state $\hat{\mathbf{u}}$	$\mathcal{O}(k^{4/3}m^{8/3})$
8	Advect particles	$\mathcal{O}(n_p(m+k))$

Algorithm 2: *Computations performed in each time step.*

$\hat{\triangle}^{AA}$ require only $\mathcal{O}(m^2 N)$ time. The coupling terms must be computed for each pair of tiles that is to be coupled; for k tiles, there are $\mathcal{O}(k^2)$ such pairs. However, the coupling tensor computation involves iterating over the boundary region only, yielding the more favorable $\mathcal{O}(m^3 N_b)$ time complexity (line 13).

Once a good set of boundary bases has been computed, new tiles can be added to the library without having to touch existing tiles. After choosing appropriate existing boundary bases, constraint reduction and computation of interior tensors are performed *only* on the new tile (lines 9 and 11). Enabling coupling to all existing tiles requires the computation of $\mathcal{O}(k)$ coupling tensors (line 13).

Algorithm 2 summarizes the computations during runtime. Given the adjacency graph between tiles, the global tensors are assembled from their precomputed parts at the start of the simulation and whenever the adjacency graph changes (lines 1–2). Each time step begins with contracting the global advection tensor. Since each tile instance has only a fixed number of neighbors (six in three dimensions), the assembled global operators are block-sparse with $7m$ entries per row or column, leading to a total cost of $\mathcal{O}(km^3)$ for evaluating line 4. Time integration is performed using Eq. 3.5. The matrix-vector multiplication with the matrix exponential (line 6) is approximated using Taylor expansion. This requires only one sparse matrix-vector multiplication and one vector addition per iteration. Empirically, the Taylor approximation

of the matrix exponential converges to machine precision in fewer than 20 iterations.

After integration, the system is projected into the admissible space defined by the constraints. This requires the solution of the block-sparse system Eq. 3.12. Eq. 3.12 is underconstrained, and has an $m - s$ dimensional solution space. To disambiguate the system, a small regularization term α is added, as in Eq. 3.9: $\alpha = 10^{-8}$ for double and $\alpha = 10^{-4}$ for single precision. Since constraints are restricted to adjacent tile instances, the number of non-zero entries per row never exceeds $7m$, similar to the advection and diffusion tensors (line 7).

The resulting sequence of reduced states can be used for evaluation or visualization. In our experiments, we use massless marker particles for flow visualization. For each particle, we need to check which of the k tile instances currently affect it, and compute the velocity at its current position by computing a weighted sum of all m basis velocities at the particle position. For n_p particles, this leads to the total cost of $\mathcal{O}(n_p(m + k))$ for particle advection.

Note that particle advection is the only step that requires the full basis present in memory. All other computations require only reduced size structures (column "Tensors" in Table 3.1). Note also that particle advection is trivially parallelizable.

3.7 Evaluation

To evaluate our technique, we have conducted a variety of experiments and comparisons. These experiments demonstrate that our technique is fast, that it does not introduce much new error beyond standard model reduction, and that it can be used to scalably construct huge simulation domains.

Simulation Error. To measure our algorithm's approximation error, we perform simple tests in 2D and 3D. In both cases, we ran a simulation of horizontal wind evolving into vortices over 200 frames. We compare the full simulation at different resolutions to monolithic model reduction [Treuille et al. 2006] with 32 basis states (64 in 3D), and coupled model reduction over a pair of adjacent tiles with 16 basis states each (32 in 3D) and a 6-dimensional boundary basis. The results are summarized in Figure 3.3.

Averaging over the whole domain, the errors for tiled and monolithic simulations are similar. In terms of $\mathcal{L}^2$-error, the coupled reduced simulation outperforms a full-scale simulation that is downsampled by a factor of 16 (8 in 3D). Especially for larger domains, this indicates that reduced models are significantly faster than full simulations, even if we compare to a downsampled simulation incurring similar errors. These results measure error in cases where the test case is close to the training data.

2D Boxes. Modular reduced models show simulation errors similar to monolithic reduced models even in situations where tiles are combined in novel ways. To demonstrate this, we ran a 150×50 simulation of horizontal flow through a domain containing two square obstacles. We then split the domain into three 50×50 subdomains called, from left to right, A, B, and C. Both A and C contain obstacles; B does not. Finally, we discard B and compute tiles for A and C independently. The reduced models for A and C are then coupled to form a tiled simulation (see Fig. 3.4).

For comparison, we ran a full simulation using the same boundary conditions as in the coupled simulation seen in Fig. 3.4. We also compute a monolithic reduced model from this new full-dimensional simulation.

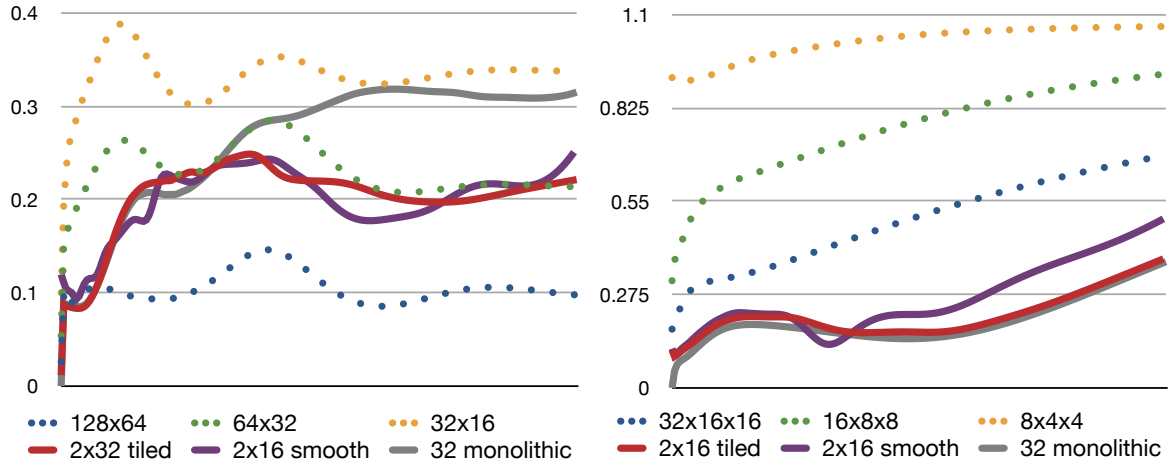


Figure 3.3: ***Simulation Error.*** *Compared to a ground truth simulation. Left: 2D test with* 256×128 *ground truth. Right: 3D test with* $64 \times 32 \times 32$ *ground truth. Shown is the relative* $\mathcal{L}^2$ *error:* $\delta(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||/||\mathbf{x}||$ *plotted against time for 200 frames.*

We then compare the tiled simulation and the monolithic reduced model against the new "ground truth" simulation. Note that in this experiment, the monolithic model is tested with its own training data, while the tiled model is not. Nevertheless, both techniques show similar errors. The relative $\mathcal{L}^2$ error approaches one because the reduced models dissipate energy faster than the full simulation (we calibrated the diffusion in all models be visually similar, leading to higher dissipation in the reduced models).

The $\mathcal{L}^2$ error is a crude measure of simulation quality, especially in the case of turbulent flows. Absent a good error measure, we have visually evaluated the performance of our approach by testing its ability to transfer vortices across tile boundaries. In order to highlight interesting areas of the flow, we advect a large number of particles and filter for a subset whose paths have high curvature. These particles tend to best show the important features of the flow.

Tiled simulation using boundary bases produces results that are far superior to simpler alternatives. Without coupling constraints, divergence along the boundary creates severe artifacts. Overlapping the two bases and blending between their velocities yields an approximately divergence-free flow, but flow features such as vortices are not transferred across the tile boundaries. Fulfilling the consistency constraints approximately without performing constraint reduction leads to a locked simulation (i.e. a small value for α in Eq. 3.9), or insufficient coupling and divergence artifacts (i.e. a large value for α in Eq. 3.9). Constraint reduction solves these issues by making the tiles compatible, thus allowing for information exchange across the boundary while fulfilling the consistency constraints exactly.

Spacecraft. This example demonstrates our algorithm's ability to recombine pieces of geometry simulated entirely independently. To compute the tiles for this scene, we ran a total of 15 full dimensional simulations of incompressible fluid flow on a three million voxel $248 \times 196 \times 71$ domain. We simulated five different "wind" directions for each of the three spacecraft designs. We then decomposed each spacecraft into wings, tail, and body. We thus obtained 14 tiles: three wing pairs, three body parts, three tails, and two additional tiles above the wings.

We created 16-dimensional boundary bases for each of the possible boundaries shown in Fig. 3.2 (a). This

			Reduced Sim				Runtime			Memory		
Name	**Shape**	**Voxels**	t	m	s	**Precomp.**	**Full**	**Reduced**	**Speedup**	**Full**	**Bases**	**Tensors**
Spacecraft	$248 \times 196 \times 71$	3.4M	14	64	16	33h	191s	0.024s	**7919**×	53MB	5.4GB	128MB
City 5x5	$285 \times 285 \times 146$	12M	7	72	12	26h	436s	0.108s	**4029**×	181MB	1.5GB	31MB
City 7x7	$399 \times 399 \times 146$	23M	7	72	12	26h	∼850s†	0.250s	∼3400×	355MB	2.3GB	73MB
City 16x16	$912 \times 912 \times 146$	121M	7	72	12	26h	∼4,400s†	1.626s	∼3900×	1.8GB	2.7GB	120MB

Table 3.1: ***Timing and memory summary***. *t, m and s denote the number of tiles, reduced dimension and the dimension of the boundary basis, respectively. "Precomp." is the precomputation time in hours, while the "Full" and "Reduced" runtimes are measured in seconds. Memory usage for the full simulation includes velocity and pressure fields, "tensors" includes all coupling terms (even those not used in the scene). All memory requirements assume single precision floating point storage. †Simulation time estimated by extrapolation.*

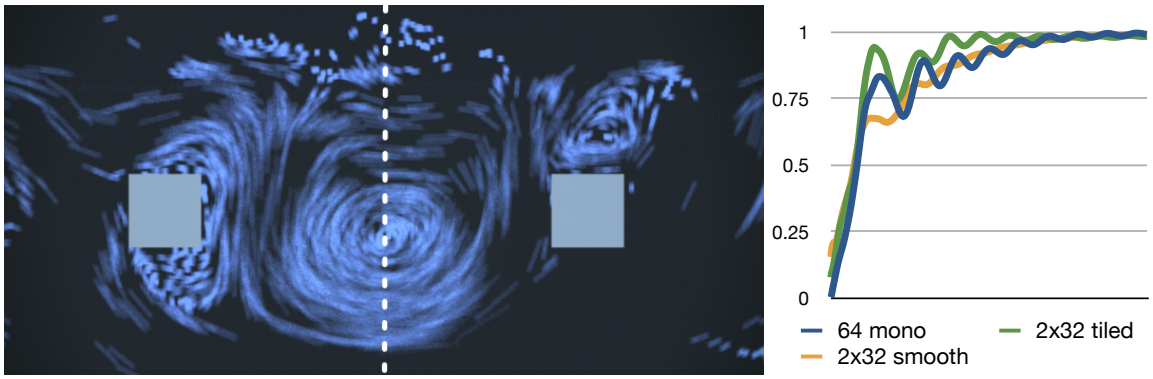


Figure 3.4: ***2D Boxes*** *example. Left: Vortices and other features can cross from one tile to another by virtue of the coupling basis. The dotted line indicates the tile boundary. The two squares are fixed obstacles. Right: Relative error of monolithic, tiled simulation, with and without smoothness constraints, plotted against time for 1000 frames.*

enables like parts to be interchanged at runtime. Finally, we created 64-dimensional simulation bases for each of the 14 tiles. This model can accomodate runtime part substitutions and wind variations within a $90°$ range. Our approach captures the turbulent wakes left by this obstacle even for tile combinations not originally simulated.

City. We simulated a large city, demonstrating that our approach enables interactive simulation on huge domains. We performed 90 fluid simulations for 200 frames of various 2×2 building configurations on a $114 \times 114 \times 146$ domain. The simulations were initialized with wind from one of the four compass directions or a turbulent initial state without net wind. For each building type, we extracted time series for a $57 \times 57 \times 146$ domain from the simulations. We then built two 12-dimensional coupling bases for the x and y faces, respectively, and computed 72-dimensional simulation bases for each building from these coupling bases. In this setting, every building can be coupled with every other building. We use these bases to build a set of cities ranging from 5×5 to 16×16 blocks. Again, we visualize flow using filtered particles. The results in the accompanying video show that our technique captures small scale simulation features, turbulent wakes that cross tile boundaries, and enables runtime part substitution and simulation modification such as the introduction of tornadoes or the removal or changing of tiles.

Table 3.1 shows statistics on scene preparation, precomputation times and runtime performance. Both the full-dimensional and coupled simulations were performed on a quad-core 1.1 GHz AMD Opteron with 16GB of RAM. The reported simulation times include simulation of the coupled reduced system and constraint handling. Timings do not include particle advection, particle filtering, and rendering. The speedups are approximate and we did not heavily optimize either our full dimensional or our coupled simulation runtime.

3.8 Summary

This chapter described a method for building scalable, reconfigurable data-driven simulations. The central idea is to distill high-resolution simulation data into simulation tiles that can be combined in a modular fashion. Our results demonstrate that tilings can scale to very large domains. Runtime complexity is low because dynamics and coupling operate entirely in the low-dimensional reduced space. We believe this technique brings us closer to complex virtual environments endowed with high resolution dynamics.

Our technical contributions relate to coupled simulation and constraint satisfaction across tiles. The tile representation induces simulation operators which can be precomputed and decomposed, enabling assembly and reconfiguration at runtime. Moreover, the approach enables extensible libraries of tiles: we can introduce new tiles without recomputing information about existing tiles.

Another contribution is constraint reduction, which enables tile coupling by allowing constraints to be enforced across tile boundaries. We modify the simulation bases so that they can satisfy a large set of consistency constraints while preserving sufficient freedom in the representation to prevent locking. This technique generalizes to arbitrary linear equality constraints on any reduced model. Beyond enforcing consistency, the method opens up interesting possibilities for adapting reduced models to linear constraints in a post-process.

Using our method, tiles can be assembled at runtime to produce fluid simulations with obstacles not contained in the original data. This modularity overcomes one of the principal limitations of data-driven simulation: the inability to adapt the reduced model once it has been computed.

Otherwise, our approach shares some of the limitations of monolithic data-driven simulation. Depending on the size of the domain, the memory cost of storing bases can be high. Also, representational limits of the basis incur greater accuracy costs than full simulation. As expected from a data-driven technique, these errors are particularly noticeable when the reduced system is presented with inputs far from the training data. Finally, like other reduced order techniques for fluid simulation, our method cannot be used to simulate multi-phase flows. In particular, fluids with free surfaces cannot be properly handled.

Generating the simulation data to construct fluid tiles requires some care. It is possible to create fundamentally incompatible tiles which lose much of the representational power during constraint reduction. This suggests a difficult but exciting open problem: can we generate bases that preserve their full representational power when subject to coupling?

While our constraint reduction technique is in principle general, and we would like to see it applied to a wide range of phenomena, in this chapter we have only applied it to fluid simulation. One factor that limits how widely constraint reduction can be applied is the inflexibility of Galerkin projection: Galerkin projection in its usual formulation works only for polynomial functions. In the next chapter, we show how

Galerkin projection can be extended to handle divisions and square roots, allowing for the simulation of a variety of phenomena – including fluids – in the presence of continuously deforming geometry.

Chapter 4

Non-Polynomial Galerkin Projection

While the previous chapter focused on allowing discrete reconfigurability of Galerkin-projected simulations, it did nothing to lift one of the most onerous constraints on the use of Galerkin projection: the need for the simulation dynamics it treats to be expressible in polynomial form in order to achieve substantial speedups. This chapter addresses the latter concern with an efficient extension to any function composed of *elementary algebraic operations* – the four operations of arithmetic plus rational roots – thus expanding the applicability of this data-driven simulation technique across graphics.

We illustrate the utility of this approach by showing applications to two strikingly different problems: radiosity rendering and fluid simulation. Although both of these phenomena can be expressed in polynomial form on a fixed mesh, in this chapter we show that allowing geometric deformation requires non-polynomial operations to express changes to the dynamics and appearance. While applying standard Galerkin projection to such functions is possible, it is a futile exercise, since it does not yield any runtime speed improvement. One workaround is to abandon the analytic projection used in creation of the reduced tensors in favor of sampling-based projection approaches (e.g. [Carlberg 2011]). This new technique, by contrast, can efficiently model these complex non-polynomial systems without abandoning analytic projection. Similar to standard analytic Galerkin projection, this approach not only preserves key optimality guarantees, but also generates a compact, analytic model in the reduced space.

4.1 Related Work

In numerical analysis, Galerkin projection has primarily been used for linear and rational functions. Rational Krylov methods approximate a rational transfer function in the frequency domain using moment matching [Olssen 2005; Ebert and Stykel 2007; Gugercin et al. 2006] to find good bases for linear time-invariant (LTI) systems and extensions thereof. Alternatives include rational function fitting (also known as multipoint methods) [Liua et al. 2008; Gallivan et al. 1996; Grimme 1997] and balanced truncation [Li 2000; Zhou 2002; Gugercin and Antoulas 2004]. The key difference between these works and the method presented in this section is that their goal is to reduce linear, time-invariant systems (LTIs) by analyzing their rational transfer functions, whereas we are interested in the reduction of non-polynomial functions for their own sake.

Because of the restrictive nature of LTIs, extensions have been proposed for time-varying systems, both

linear [Phillips 1998; Sandberg and Rantzer 2004; Chahlaoui and van Dooren 2005; Hossain and Benner 2008] and multilinear [Savas and Eldén 2009; Carlberg 2011]. The work closest to this chapter is Farhood and Dullerud [2007], who apply rational Krylov methods to linear systems rationally dependent on time-varying parameters. However, unlike our work, theirs does not guarantee preservation of polynomial degree for arbitrary compositions of elementary algebraic operations and therefore can become computationally intractable for complex phenomena. Using an algebraic approach similar to that of Debusschere et al. [2004] for probabilistic dynamics, we alter Galerkin projection to enable order-preserving reduction for arbitrary compositions of elementary algebraic operations. To our knowledge, the extension presented in this chapter is the first work to present a general framework for analytic Galerkin projection of arbitrary compositions of elementary algebraic operations while preserving polynomial degree, an essential property for real-time graphics, and the first work to simulate fluid flow or radiosity on deforming meshes.

Reduced-order fluid models. Previous reduced fluid models in graphics and numerical analysis have been unable to model complex, continuously-deforming boundaries. A straightforward solution is to construct separate bases for each possible boundary configuration [Schmit and Glasuer 2002]. However, this approach cannot scale to continuously deforming boundaries. For flows with periodic boundaries, and with inherent symmetries within the flow dynamics, it is possible to remove uniform translation modes [Rowley et al. 2003; Rowley and Marsden 2000]. For a single rotating object, the basis can be constructed in the object's frame of reference and then simulated at various angles [Ausseur et al. 2004]. Treuille et al. [2006] enabled the insertion of rigidly moving boundaries, and, in Chapter 3, we demonstrated how to allow discrete boundary reconfiguration at runtime. Perhaps the work closest to that presented in this chapter is that of Fogleman et al. [2004], which enabled linear deformation along a single axis to model reduce piston and combustion simulation.

We enable continuous boundary motion by embedding the fluid in a tetrahedral mesh, similar to Elcott et al. [2007], and deforming the mesh along with the boundary. This requires a full-dimensional fluid solver that works for tetrahedral meshes. We considered several classes of solvers including finite element methods (e.g. [Feldman et al. 2005a; Feldman et al. 2005b]), methods based on discrete exterior calculus [Mullen et al. 2009; Pavlov et al. 2011], and ALE methods (such as [Klingner et al. 2006]). We chose the residual distribution scheme [Sewall et al. 2007; Dobes and Deconinck 2006; Deconinck and Ricchiuto 2007], which is akin to a finite-difference fluid approximation [Foster and Metaxas 1996], but generalized to a tetrahedral mesh, due to its amenability to our non-polynomial Galerkin projection and the fact that it can be stably integrated in the reduced space.

Kim et al. [2013] apply a similar extension to Galerkin projection in order to construct a reduced fluid model, although the details of their simulation technique differ from the one we present in this chapter.

4.2 Method

While §3.2.2 describes the application of Galerkin projection to polynomial functions, it is more difficult to see how one can efficiently apply Galerkin projection to non-polynomial functions. For example, consider the rational function $\mathbf{y} = \mathbf{f}(\mathbf{x})$, where $\mathbf{x} = [x_1, x_2]^T$, $\mathbf{y} = [y_1, y_2]^T$, and:

$$y_1 = \frac{x_1 x_2 + x_2^2}{x_1^2} \qquad y_2 = \frac{x_1^2}{x_2^2}. \tag{4.1}$$

One could compute the Galerkin projection of this function by transforming reduced vectors into the full space, applying the full space equations, and projecting the results back to the reduced space, but

this would yield no speed advantage. Instead, one can use two tensors to express $\mathbf{f}(\mathbf{x})$, rewriting it as a matrix-vector product:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1^2 & 0 \\ 0 & x_2^2 \end{bmatrix}^{-1} \begin{bmatrix} x_1 x_2 + x_2^2 \\ x_1^2 \end{bmatrix}$$

Both the matrix and the vector are polynomial (specifically, quadratic) in $\mathbf{x}$. Therefore one can evaluate the matrix using a 4th-order tensor $\mathbf{Q}_1$, and the vector using an 3rd-order tensor $\mathbf{Q}_2$:

$$\begin{bmatrix} x_1^2 & 0 \\ 0 & x_2^2 \end{bmatrix} = \mathbf{Q}_1 \otimes_2 \mathbf{x} \otimes_3 \mathbf{x}$$

$$\begin{bmatrix} x_1 x_2 + x_2^2 \\ x_1^2 \end{bmatrix} = \mathbf{Q}_2 \otimes_1 \mathbf{x} \otimes_2 \mathbf{x}$$

where $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are (labeling each tensor slice by its associated polynomial term):

$$\begin{aligned} \mathbf{Q}_1 \otimes_2 \mathbf{x} \otimes_3 \mathbf{x} = \; & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} x_1^2 + \quad \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} x_1 x_2 + \\ & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} x_2 x_1 + \quad \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} x_2^2 \end{aligned}$$

and

$$\mathbf{Q}_2 \otimes_1 \mathbf{x} \otimes_2 \mathbf{x} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} x_1^2 + \begin{bmatrix} 1 \\ 0 \end{bmatrix} x_1 x_2 + \begin{bmatrix} 0 \\ 0 \end{bmatrix} x_2 x_1 + \begin{bmatrix} 1 \\ 0 \end{bmatrix} x_2^2$$

To evaluate $\mathbf{f}(\mathbf{x})$, contract $\mathbf{Q}_1$ and $\mathbf{Q}_2$ as shown above, then invert the matrix $\mathbf{Q}_1 \otimes_2 \mathbf{x} \otimes_3 \mathbf{x}$, and finally compute the matrix-vector product. In the form of a single equation:

$$\mathbf{y} = (\mathbf{Q}_1 \otimes_2 \mathbf{x} \otimes_3 \mathbf{x})^{-1} \otimes_1 (\mathbf{Q}_2 \otimes_1 \mathbf{x} \otimes_2 \mathbf{x}) \tag{4.2}$$

This equation is completely equivalent to Eq. 4.1. This form is preferred because it makes the tensor structure of the computation clear.

More generally, to reduce a non-polynomial function $\mathbf{f}(\mathbf{x})$, write $\mathbf{f}(\mathbf{x})$ in terms of a collection of tensors $\mathbf{Q}_1 \ldots \mathbf{Q}_k$, to which we apply a series of tensor contractions and matrix operations. This technique can reduce functions expressible using only the following operations:

(i) $\mathbf{Q} \otimes \mathbf{X}$, tensor product.

(ii) $\mathbf{Q}^{-1}$, matrix inverse.

(iii) $\mathbf{Q}^{\frac{1}{n}}$, matrix root. $\mathbf{Q}$ is a symmetric positive semidefinite matrix and $\mathbf{Q}^{\frac{1}{n}}$ is uniquely identified as the positive semidefinite matrix such that $(\mathbf{Q}^{\frac{1}{n}})^n = \mathbf{Q}$.

As §3.2.2 demonstrated, polynomial functions require only one tensor and repeated applications of operation (i) for evaluation. However, as shown in Eq. 4.1, there are many cases in which it may be necessary to use multiple tensors and operations beyond (i) to evaluate $\mathbf{f}(\mathbf{x})$.

Once one has expressed a function $\mathbf{f}(\mathbf{x})$ in terms of tensors $\mathbf{Q}_1, \ldots, \mathbf{Q}_k$ and our allowed matrix operations, the function can be reduced. As a first step, this requires finding bases for every axis of every tensor $\mathbf{Q}_i$. For example, a polynomial $\mathbf{y} = \mathbf{q}_i(\mathbf{x}_1, \ldots, \mathbf{x}_d)$ requires separate bases for $\mathbf{y}, \mathbf{x}_1, \ldots, \mathbf{x}_d$. After finding these bases, one can pre-multiply each tensor by its associated bases, as in Eq. 3.2. Computing $\hat{\mathbf{f}}(\hat{\mathbf{x}})$ then follows exactly the sequence of operations used to compute $\mathbf{f}(\mathbf{x})$, except that each tensor $\mathbf{Q}_i$ is replaced with its reduced counterpart $\hat{\mathbf{Q}}_i$. That is, each operation transforms as follows:

	Operation	Full Form	Reduced Form
(i)	Tensor Product	$\mathbf{Q} \otimes_k \mathbf{X}$	$\hat{\mathbf{Q}} \otimes_k \hat{\mathbf{X}} = \mathbf{B}_q^T\mathbf{Q} \otimes_k \mathbf{B}_x\hat{\mathbf{X}}$
(ii)	Matrix Inverse	$\mathbf{Q}^{-1}$	$\hat{\mathbf{Q}}^{-1} = (\mathbf{B}_q^T\mathbf{Q}\mathbf{B}_q)^{-1}$
(iii)	Matrix Root	$\mathbf{Q}^{\frac{1}{n}}$	$\hat{\mathbf{Q}}^{\frac{1}{n}} = (\mathbf{B}_q^T\mathbf{Q}\mathbf{B}_q)^{\frac{1}{n}}$

While each reduction is straightforward, these reductions do have implications for basis selection, as shown in the last column of the table. Since every tensor must be reduced along all of its axes, a basis is required for each axis of each tensor. Operations (ii) and (iii) require that $\mathbf{Q}$ be reduced using the same basis along both axes. §4.2.2 explains this restriction. If $\hat{\mathbf{Q}}$ is small, which is the case for typical choices of basis size, then $\hat{\mathbf{Q}}^{\frac{1}{n}}$ can be computed efficiently using eigen-decomposition.

4.2.1 Reduction Example

To reduce the example function (Eq. 4.1) from the previous section, begin by inspecting the tensor form (Eq. 4.2) to see which bases are required. Reading from right to left, it appears that one requires a basis $\mathbf{B}_x$ for $\mathbf{x}$, a basis $\mathbf{B}_n$ for $\mathbf{Q}_2 \otimes_1 \mathbf{x} \otimes_2 \mathbf{x}$, and a basis $\mathbf{B}_y$ for $\mathbf{y}$. Due to the restriction that an inverted matrix must be reduced by the same basis along both axes, $\mathbf{B}_y = \mathbf{B}_n$, and we will refer to both as $\mathbf{B}_y$. The reduction of $\mathbf{Q}_1$ and $\mathbf{Q}_2$ is then:

$$\begin{aligned}\hat{\mathbf{Q}}_1 &= \mathbf{B}_y^T\mathbf{Q}_1 \otimes_1 \mathbf{B}_y \otimes_2 \mathbf{B}_x \otimes_3 \mathbf{B}_x \\ \hat{\mathbf{Q}}_2 &= \mathbf{B}_y^T\mathbf{Q}_2 \otimes_1 \mathbf{B}_x \otimes_2 \mathbf{B}_x\end{aligned}$$

and to evaluate $\hat{\mathbf{f}}(\hat{\mathbf{x}})$ at runtime, perform:

$$\hat{\mathbf{y}} = (\hat{\mathbf{Q}}_1 \otimes_2 \hat{\mathbf{x}} \otimes_3 \hat{\mathbf{x}})^{-1} \otimes_1 (\hat{\mathbf{Q}}_2 \otimes_1 \hat{\mathbf{x}} \otimes_2 \hat{\mathbf{x}})$$

The only data required to perform this computation are the two reduced tensors $\hat{\mathbf{Q}}_1$ and $\hat{\mathbf{Q}}_2$. Displaying the full space result of the computation at runtime will also require $\mathbf{B}_y$ in order to compute $\mathbf{y} = \mathbf{B}_y\hat{\mathbf{y}}$.

4.2.2 Properties

This choice of reduction is not the only one possible for non-polynomial functions. It would have been possible, for example, to replace each full-space operation with something more complex than the same operation performed on reduced tensors. The method proposed here does, however, have three principal advantages. First, it is *simple*: once a function has been expressed in terms of tensors, the reduction consists only in replacing full-dimensional tensors with corresponding reduced-dimensional tensors. Second, it is *efficient*: tensor order is preserved by the reduction, which is important because both tensor storage costs and evaluation time complexity are exponential in the tensor order. Third, it is *optimal*: each reduction rule is constructed to minimize some measure of error (similar to [Carlberg et al. 2011]), given a particular decomposition into tensors of the target function. We discuss the optimality properties of each of these operations in turn.

Tensor product. Given a single tensor $\mathbf{Q}$ of order $d+1$ which is contracted d times by a fixed vector $\mathbf{x}$: $\mathbf{y} = \mathbf{Q} \otimes_{1\ldots d} \mathbf{x}$, $\min_{\hat{\mathbf{y}}} ||\mathbf{B}_y\hat{\mathbf{y}} - \mathbf{Q} \otimes_{1\ldots d} \mathbf{B}_x\hat{\mathbf{x}}||$ is minimized at $\hat{\mathbf{y}} = \mathbf{B}_y^T\mathbf{Q} \otimes_{1\ldots d} \mathbf{x}$, exactly as in the case of the ordinary polynomial reduction (Eq. 3.1) discussed in §3.2.2.

Matrix inverse. For the matrix inverse, operation (ii), let $\mathbf{y} = \mathbf{Q}^{-1}\mathbf{x}$. $\min_{\hat{\mathbf{y}}} ||\mathbf{B}_x\hat{\mathbf{y}} - \mathbf{Q}^{-1}\mathbf{B}_x\hat{\mathbf{x}}||_{\mathbf{Q}}$ is minimized at $\hat{\mathbf{y}} = (\mathbf{B}_x^T\mathbf{Q}\mathbf{B}_y)^{-1}$. Note that this is not the same as the ordinary Galerkin projection of $\mathbf{Q}^{-1}$, which would be $\mathbf{B}_x^T\mathbf{Q}^{-1}\mathbf{B}_x$. One could also allow the bases for $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ to differ, in which case the error would be minimized at $\hat{\mathbf{y}} = (\mathbf{B}_x^T\mathbf{Q}\mathbf{B}_y)^{-1}\mathbf{B}_y^T\mathbf{B}_x$. However, setting $\mathbf{B}_x = \mathbf{B}_y$ has significant benefits in practice, such as energy conservation in the fluids application (§4.4.3).

Matrix root. Matrix roots, operation (iii), approximate $\mathbf{Q}^{\frac{1}{n}}$ by first finding the reduced matrix $\hat{\mathbf{Q}}$ that best approximates $\left(\mathbf{Q}^{\frac{1}{n}}\right)^n = \mathbf{Q}$, computing $\min_{\hat{\mathbf{Q}}} ||\mathbf{B}_x\hat{\mathbf{Q}}\mathbf{x} - \mathbf{Q}\mathbf{B}_x\mathbf{x}||$. This reduced matrix is $\hat{\mathbf{Q}} = \mathbf{B}_x^T\mathbf{Q}\mathbf{B}_x$. One can then use its nth root, $\hat{\mathbf{Q}}^{\frac{1}{n}}$, to approximate $\mathbf{Q}^{\frac{1}{n}}$. Multiplying $\mathbf{Q}$ by the same basis along both axes ensures that $\hat{\mathbf{Q}}$ is also symmetric and positive semidefinite. Again, our reduction differs from the ordinary Galerkin projection of $\mathbf{Q}^{\frac{1}{n}}$, which is $\mathbf{B}_x^T\mathbf{Q}^{\frac{1}{n}}\mathbf{B}_x$.

Speed-optimality tradeoff. We restricted the optimality discussion for operation (i) to the case where the function $\mathbf{f}(\mathbf{x})$ is represented using a single tensor. However, in many cases it is possible to represent a polynomial using multiple tensors. This alternative representation exchanges optimality for speed by composing polynomials. Suppose $\mathbf{q}(\mathbf{x}) = \mathbf{q}_1(\mathbf{q}_2(\mathbf{x}))$, where $\mathbf{q}$ has degree $d = ab$, $\mathbf{q}_1$ has degree a, and $\mathbf{q}_2$ has degree b. One can choose to express $\mathbf{q}(\mathbf{x})$ as either $\mathbf{Q} \otimes_{1\ldots d} \mathbf{x}$ or $\mathbf{Q}_1 \otimes_{1\ldots a} (\mathbf{Q}_2 \otimes_{1\ldots b} \mathbf{x})$. In the full space, these expressions are identical. When reduced, however, these expressions become $(\mathbf{B}_y^T\mathbf{Q} \otimes_{1\ldots d} \mathbf{B}_x) \otimes_{1\ldots d} \hat{\mathbf{x}}$ and $(\mathbf{B}_y^T\mathbf{Q}_1 \otimes_{1\ldots a} \mathbf{B}_z) \otimes_{1\ldots a} ((\mathbf{B}_z^T\mathbf{Q}_2 \otimes_{1\ldots b} \hat{\mathbf{x}})$. The second case can also be written as:

$$\mathbf{B}_y^T\mathbf{Q}_1 \otimes_{1\ldots e} (\mathbf{B}_z\mathbf{B}_z^T\mathbf{Q}_2 \otimes_{1\ldots g} \mathbf{B}_x\hat{\mathbf{x}})$$

which is equivalent to:

$$\mathbf{B}_x^T\mathbf{q}_1(\mathbf{B}_z\mathbf{B}_z^T\mathbf{q}_2(\mathbf{B}_x\hat{\mathbf{x}})).$$

Notice that the composition introduces an extra projection $\mathbf{B}_z\mathbf{B}_z^T$, which reduces the accuracy of the reduced result. On the other hand, the reduced composition is faster to compute than the reduced original polynomial: the composition replaces one reduced tensor containing $\hat{n}^{eg+1}$ elements with two much smaller reduced tensors, one containing $\hat{n}^{e+1}$ elements, and the other containing $\hat{n}^{g+1}$.

4.2.3 Summary

We have demonstrated that any function constructable from tensor contraction, matrix inversion, and matrix roots can be easily model-reduced using our non-polynomial Galerkin projection method. Reducing such a function $\mathbf{f}(\mathbf{x})$ can be accomplished by constructing it using a collection of tensors $\mathbf{Q}_1, \ldots, \mathbf{Q}_k$ and applying our tensor and matrix operations. The reduced counterpart $\hat{\mathbf{f}}(\hat{\mathbf{x}})$ can then be computed by simply replacing the tensors $\mathbf{Q}_1, \ldots, \mathbf{Q}_k$ with the tensors $\hat{\mathbf{Q}}_1, \ldots, \hat{\mathbf{Q}}_k$, and maintaining exactly the same sequence of operations. This method is simple, efficient, and optimal in the sense described in §4.2.2.

4.3 Fluid Model

As an example of this non-polynomial Galerkin projection method, we describe the reduction of fluid flow on a deforming tetrahedral mesh. In this system, movement of the mesh boundary exerts force on the fluid, but forces from fluid motion do not cause the mesh boundary to move. This method attempts to keep the fluid motion as independent as possible of the motion of the interior of the mesh. The simulation state consists of fluid velocities $\mathbf{u}$ and fluid momenta $\mathbf{p}$, located at the centroid of each element, and of fluid

	begin fullSimStep($\mathbf{u}_t$, $\mathbf{f}_t$, $\mathbf{g}_t$, $\mathbf{g}_{t+1}$):	**begin** reducedSimStep($\hat{\mathbf{u}}_t$, $\hat{\mathbf{f}}_t$, $\hat{\mathbf{g}}_t$, $\hat{\mathbf{g}}_{t+1}$)
flux combination	$\dot{\mathbf{g}} \leftarrow \mathbf{g}_{t+1} - \mathbf{g}_t$ $\mathbf{f}' \leftarrow \mathbf{f}_t - \mathbf{h}(\dot{\mathbf{g}})$	$\hat{\mathbf{f}}' \leftarrow \hat{\mathbf{f}}_t - \mathbf{B}_f^T \mathbf{B}_h \hat{\mathbf{h}}_t$
advection (§4.3.1) *diffusion (§4.3.2)*	$\mathbf{u}' \leftarrow \mathbf{u}_t + \int_t^{t+1} [\mathbf{V}^{-1}(\mathbf{g}_t)(\mathbf{A} \otimes_2 \mathbf{f}' - \mu\Delta)]\mathbf{u}_t$	$\hat{\mathbf{u}}' \leftarrow \exp[\Delta t \hat{\mathbf{V}}^{-1}(\hat{\mathbf{g}}_t)(\hat{\mathbf{A}} \otimes_2 \hat{\mathbf{f}}' - \mu\hat{\Delta})]\hat{\mathbf{u}}_t$
pressure projection (§4.3.3)	$\mathbf{f}_{t+1} \leftarrow \min \|\mathbf{u}' - \mathbf{V}^{-1}(\mathbf{g}_{t+1})(\mathbf{P} \otimes_2 \mathbf{g}_{t+1})\mathbf{f}\|_{\mathbf{V}(\mathbf{g}_{t+1})}$ s. t. $\mathbf{Df} - \mathbf{Dh}(\dot{\mathbf{g}}) = 0$ and $\nabla \cdot \mathbf{f} = 0$	$\hat{\mathbf{f}}_{t+1} \leftarrow \min_{\hat{\mathbf{f}}} \|\hat{\mathbf{u}}' - \hat{\mathbf{V}}^{-1}(\hat{\mathbf{g}}_{t+1})(\hat{\mathbf{P}} \otimes_2 \hat{\mathbf{g}}_{t+1})\hat{\mathbf{f}}\|_{\hat{\mathbf{V}}(\hat{\mathbf{g}}_{t+1})}$ s. t. $\hat{\mathbf{D}}\hat{\mathbf{f}} - \hat{\mathbf{D}}_h \hat{\mathbf{h}}_t = 0$
convert to velocity	$\mathbf{u}_{t+1} \leftarrow \mathbf{V}^{-1}(\mathbf{g}_{t+1})(\mathbf{P} \otimes_2 \mathbf{g}_{t+1})\mathbf{f}_{t+1}$	$\hat{\mathbf{u}}_{t+1} \leftarrow \hat{\mathbf{V}}^{-1}(\hat{\mathbf{g}}_{t+1})(\hat{\mathbf{P}} \otimes_2 \hat{\mathbf{g}}_{t+1})\hat{\mathbf{f}}_{t+1}$
	end	**end**

Figure 4.1: *Algorithmic summary of the full-dimensional and reduced time steps. Note the close correspondence.*

fluxes $\mathbf{f}$ through each face. (Ordinarily, one would use only one of these descriptions of fluid flow. As we shall see, however, non-polynomial Galerkin projection demands separate treatment of these quantities.) The mesh topology must remain constant, but continuous deformation of the positions of the mesh vertices denoted, $\mathbf{g}$ (Fig. 4.2), is allowed.

Since the fluid is represented on a moving tetrahedral mesh, and not a fixed rectangular grid as in Chapter 3, a different fluid discretization and full-space fluid simulation method is required. We begin from the incompressible Navier-Stokes momentum equation:

$$\dot{\mathbf{u}} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nu\nabla^2\mathbf{u} + \nabla p + \mathbf{e}, \tag{4.3}$$

where p denotes pressure, ν viscosity, and $\mathbf{e}$ external forces, and discretize this equation using the residual distribution scheme of Dobes, Deconinck, and Ricchiuto [2006; 2007], which is essentially a finite-differencing method applied to tetrahedral meshes. Then, using "operator splitting", described in [Stam 1999], the velocity update is divided into advection, diffusion, and pressure projection steps. The algorithm is summarized in Fig. 4.1, where we integrate the full-space equations using a 4th order Runge-Kutta integrator. We now describe each step in detail.

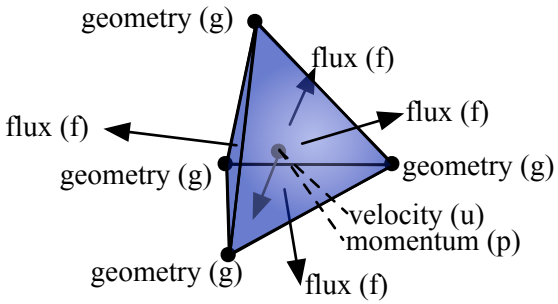


Figure 4.2: *Geometric layout of the simulation variables on a tetrahedral element.*

4.3.1 Advection

The advection step transports quantities through the mesh. Each of the x, y, and z components of velocity is treated separately and transported between mesh elements according to the flux $\mathbf{f}$. This transport is described by the advection tensor $\mathbf{A}$, which interpolates velocities onto the mesh faces, multiplies the interpolated velocities by $\mathbf{f}$ to find the rate of momentum transport over each face, and finally sums the

momentum transported over a cell's faces to find the momentum derivative at that cell as $\dot{\mathbf{p}} = (\mathbf{A} \otimes_2 \mathbf{f})\mathbf{u}$. For a cell i, $\mathbf{A}$ computes:

$$\dot{\mathbf{p}}_i = \sum_e \mathbf{f}_{f_{ie}} \frac{1}{2}(\mathbf{u}_e + \mathbf{u}_i), \tag{4.4}$$

where the sum index e runs over the four face-adjacent cells to i, and f_{ie} denotes the index of the (oriented) face between i and e.

However, the quantity of interest is not momentum $\dot{\mathbf{p}}$, but velocity $\dot{\mathbf{u}}$. Assuming a constant density (incompressible) fluid[1], $\mathbf{u}$ and $\mathbf{p}$ are related by volume. The volume of each cell is a cubic polynomial in the vertex positions, which must be evaluated to produce a diagonal matrix whose elements are cell volumes. This evaluation can be achieved using the 5th-order *volume tensor* $\mathbf{V}$, which computes a matrix-valued cubic polynomial in the vertex locations:

$$\mathbf{V} \otimes_{2\ldots4} \mathbf{g} = \begin{bmatrix} v_1(\mathbf{g}) & & \\ & \ddots & \\ & & v_n(\mathbf{g}) \end{bmatrix} \tag{4.5}$$

where $v_k(\mathbf{g})$ is the volume of cell k as a function of the vertex positions. For clarity, we will write $\mathbf{V}(\mathbf{g})$ in place of $\mathbf{V} \otimes_{2\ldots4} \mathbf{g}$.

Given $\mathbf{V}$, momentum can be computed: $\mathbf{p} = \mathbf{V}(\mathbf{g})\mathbf{u}$. This gives us the advection equation:

$$\dot{\mathbf{u}} = \mathbf{V}^{-1}(\mathbf{g})(\mathbf{A} \otimes_2 \mathbf{f})\mathbf{u}. \tag{4.6}$$

Note that this equation depends on the geometry $\mathbf{g}$. If $\mathbf{g}$ were constant, $\mathbf{V}$ would simply be a constant matrix, and we could precompute $\mathbf{V}^{-1}$ and absorb it into $\mathbf{A}$. Because we want to simulate the fluid behavior in the presence of changing geometry, we must explicitly represent $\mathbf{V}$ as a tensor which we can contract to a matrix and invert. This requires our non-polynomial Galerkin projection technique.

4.3.2 Diffusion

We discretize viscosity as follows:

$$\dot{\mathbf{u}} = -\mu \mathbf{V}^{-1}(\mathbf{g}) \triangle \mathbf{u}, \tag{4.7}$$

where μ is a viscosity coefficient, $\triangle$ is the graph Laplacian $(\triangle \mathbf{u})_i = -|\mathcal{N}_i|\mathbf{u}_i + \sum_{j \in \mathcal{N}_i} \mathbf{u}_j$, $\mathcal{N}_i$ are the (usually four) tetrahedra neighboring tetrahedron i, and $\mathbf{V}$ is the volume tensor. We have found that this simple, geometric approximation to diffusion is sufficient in both the full and reduced spaces. After computing the contribution of both advection and diffusion to $\dot{\mathbf{u}}$, we use an explicit time integration scheme to update the velocity.

4.3.3 Projection

Given a velocity $\mathbf{u}$, the projection step first generates a flux $\mathbf{f}$ that is close to $\mathbf{u}$ and satisfies the incompressibility constraint $\nabla \cdot \mathbf{f} = 0$. We can then convert $\mathbf{f}$ back to a velocity that corresponds exactly to the incompressible flux.

[1] Specifically, a fluid where the density is 1.

To perform the conversion from flux to velocity, we introduce the tensor $\mathbf{P}$, which sums the volume-weighted directed fluxes of a cell to obtain the momentum of the cell: $\mathbf{p} = (\mathbf{P} \otimes_2 \mathbf{g})\mathbf{f}$. Thus, the velocity corresponding to $\mathbf{f}$ is given by $\mathbf{u} = \mathbf{V}^{-1}(\mathbf{g})(\mathbf{P} \otimes_2 \mathbf{g})\mathbf{f}$. Note that this relation only holds if the fluxes $\mathbf{f}$ are divergence-free.

Using the flux-to-velocity conversion and a velocity field $\mathbf{u}$, we can find the flux field $\mathbf{f}$ that fulfills our incompressibility constraint $\nabla \cdot \mathbf{f} = 0$, while minimizing the energy of the difference between its corresponding velocity $\mathbf{V}^{-1}(\mathbf{g})(\mathbf{P} \otimes_2 \mathbf{g})\mathbf{f}$ and $\mathbf{u}$. The energy to be minimized is

$$\frac{1}{2}||\mathbf{u} - \mathbf{V}^{-1}(\mathbf{g})(\mathbf{P} \otimes_2 \mathbf{g})\mathbf{f}||^2_{\mathbf{V}(\mathbf{g})}, \tag{4.8}$$

where $\|\mathbf{x}\|^2_{\mathbf{M}} = \mathbf{x}^T\mathbf{M}\mathbf{x}$. We solve the minimization using Uzawa's method as described in [Benzi et al. 2005].

Again, since we need to perform flux-to-velocity conversions, we have to perform divisions by cell volumes $\mathbf{V}(\mathbf{g})$. In this case, these divisions appear in the objective of our optimization.

4.3.4 Fluid-Geometry Coupling

To allow deforming objects to exert forces on the fluid around them, we modify both the advection and projection steps of our simulation. To model the effect of the moving mesh on advection, we compute and then subtract the flow $\mathbf{h}$ induced by the motion of the mesh. This ensures that velocities do not translate simply because their discretization element moves through space. We also modify the projection step to ensure that we never advect fluid across a moving domain boundary.

The movement of face i induces a flux $h_i(\dot{\mathbf{g}})$ through that face equal to

$$h_i = A_i\dot{\mathbf{c}}_i \cdot \mathbf{n}_i, \tag{4.9}$$

where A_i is the area of the face, $\mathbf{n}_i$ its normal, and $\dot{\mathbf{c}}_i$ the velocity of its centroid. To compensate for mesh movement, we can simply subtract $\mathbf{h}(\dot{\mathbf{g}})$ wherever we use $\mathbf{f}$.

In particular, in the advection step we subtract the effect of advection due to induced fluxes from the effect of advection due to fluid fluxes: $\dot{\mathbf{p}} = \mathbf{A} \otimes_2 \mathbf{f} - \mathbf{A} \otimes_2 \mathbf{h}$. In the projection step, we enforce that there is no flow across the boundary by adding constraints $\mathbf{Df} + \mathbf{Dh} = 0$ to the projection, where $\mathbf{D}$ is an operator which selects the boundary faces. These modifications ensure that the flow inside of the domain is independent of the movement of the mesh, and that there is no flow across (possibly moving) domain boundaries.

4.3.5 Stability

Our discretization of the fluid equations is *stable*, meaning that the discrete versions of the partial differential equation do not inherently gain energy. To see why, let us consider the advection, diffusion, and projection steps separately. Energy is given by $\mathrm{E}(\mathbf{u}) = \frac{1}{2}||\mathbf{u}||^2_{\mathbf{V}(\mathbf{g})}$ and its time derivative is $\dot{\mathrm{E}} = \mathbf{u}^T\mathbf{V}(\mathbf{g})\dot{\mathbf{u}}$. Noting that we use oriented fluxes in Eq. 4.4, it is easy to see that the advection matrix $\mathbf{A} \otimes_2 \mathbf{f}$ is anti-symmetric. Substituting for $\dot{\mathbf{u}}$ according to Eq. 4.6, we can see that the advection step exactly conserves energy.

$$\dot{\mathrm{E}} = \mathbf{u}^T(\mathbf{A} \otimes_2 \mathbf{f})\mathbf{u} = 0 \tag{4.10}$$

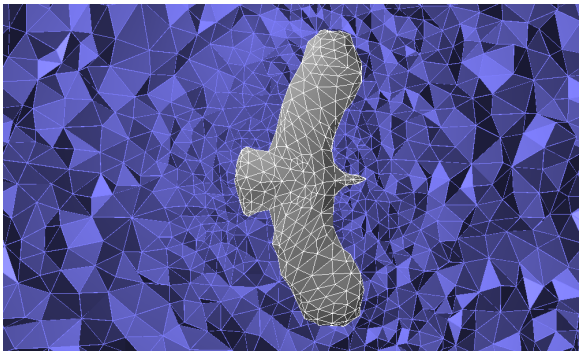


Figure 4.3: *A cutaway view of a tetrahedral mesh used in our fluid simulation application.*

The graph Laplacian $\triangle$ has only positive eigenvalues, so substituting Eq. 4.7 for $\dot{\mathbf{u}}$ gives us

$$\dot{\mathrm{E}} = \mathbf{u}^T \Big(-\mu \mathbf{V}^{-1}(\mathbf{g}) \triangle \Big) \mathbf{u} < 0. \tag{4.11}$$

Finally, in the projection step, we minimize the energy difference between the current velocities and the velocities corresponding to new, divergence free fluxes. Let $\mathbf{u}$ be the velocities before projection, $\mathbf{u}' = \mathbf{V}^{-1}(\mathbf{g})(\mathbf{P} \otimes_2 \mathbf{g})\mathbf{f}$ be the divergence-free velocities after projection, and $\mathbf{u}_\perp$ be their difference: $\mathbf{u} = \mathbf{u}' + \mathbf{u}_\perp$. (This is known as the Helmholtz-Hodge decomposition [Stam 1999].) We use $\| \cdot \|_{\mathbf{V}(\mathbf{g})}$ in our objective function Eq. 4.8, meaning $\mathbf{u}'$ and $\mathbf{u}_\perp$ are orthogonal in *energy space*. So the triangle inequality is tight: $\mathrm{E}(\mathbf{u}) = \mathrm{E}(\mathbf{u}') + \mathrm{E}(\mathbf{u}_\perp)$, which implies $\mathrm{E}(\mathbf{u}) \geq \mathrm{E}(\mathbf{u}')$, ensuring that the projection never gains energy.

Note that this does not mean that the method is unconditionally stable independent of the time integration method and time step chosen. However, as we show in §4.4.3, our reduced simulation is in fact unconditionally stable.

4.4 Reduced Fluids

We construct the reduced simulation by applying our non-polynomial reduction rules (§4.2) to the fluid simulation method from the previous section.

In order to reduce the governing equations, we have to reduce the tensors $\mathbf{V}$, $\mathbf{P}$, $\mathbf{A}$, $\triangle$, and $\mathbf{D}$. Because we require bases for each axis of each of these tensors (last paragraph before §4.2.1), we will need a flux basis $\mathbf{B}_f$, velocity basis $\mathbf{B}_u$, and momentum basis $\mathbf{B}_p$. Since $\mathbf{V}$ and $\mathbf{P}$ depend on the geometry, we need a geometry basis $\mathbf{B}_g$. We also need a basis $\mathbf{B}_h$ for the fluxes induced by mesh motion and a basis $\mathbf{B}_d$ for boundary fluxes.

4.4.1 Basis Construction

The quality of the runtime simulation depends significantly on our choice of bases. We build $\mathbf{B}_f$, $\mathbf{B}_h$, $\mathbf{B}_u$, and $\mathbf{B}_p$ using a method similar to [Treuille et al. 2006]. We run a set of full-dimensional simulations and collect snapshots of simulation quantities into large matrices, where each column represents a simulation frame. We create bases by running out-of-core Singular Value Decomposition (SVD) on these matrices

using the method of James and Fatahalian [2003a]. After computing the momentum and velocity bases independently, we concatenate these bases and re-orthonomalize. This process ensures that $\mathbf{B}_u = \mathbf{B}_p$, which ensures energy preservation in the reduced space (§4.4.3) and complies with the requirements of matrix inversion reduction (§4.2).

Creating the geometry basis $\mathbf{B}_g$ is more difficult. We begin with a sequence of triangle meshes animating changes to the boundary conditions, such as the flapping wings of a bird. We select an intermediate pose for the boundary mesh and construct a tetrahedral *base mesh* discretizing the simulation domain. For each deformed state of the surface model, we then use Laplacian deformation transfer [Sorkine 2006] to deform the base mesh so that the embedded surface matches the deformed surface model. Unfortunately, this step can lead to inverted elements, which would be fatal to the simulation. To fix inversions, we interleave the following two steps. First, we increase the weights around inverted elements to increase rigidity during Laplacian deformation and thus avoid inversion. Second, we improve the quality of the mesh by running Stellar [Klingner and Shewchuk 2007], which we modified to leave surface vertices unchanged. Unfortunately, the latter step can lead to discontinuities in the animation sequence, where the configuration of internal vertices rapidly changes between simulation frames. This popping artifact, which can be seen in our example video, is not fatal, but removing it would likely improve simulation quality and is an open question for future research. Once we have an inversion-free tetrahedral mesh animation, we run SVD to create $\mathbf{B}_g$.

4.4.2 Tensor Reduction

Using these bases, we follow the procedure described in §4.2, turning the full space tensors into their reduced equivalents:

[MS: center this]

	Tensor	Galerkin Projection
Advection	$\mathbf{A}$	$\hat{\mathbf{A}} = \mathbf{B}_p^T \mathbf{A} \otimes_1 \mathbf{B}_u \otimes_2 \mathbf{B}_f$
Induced Advection	$\mathbf{A}_h$	$\hat{\mathbf{A}}_h = \mathbf{B}_p^T \mathbf{A} \otimes_1 \mathbf{B}_u \otimes_2 \mathbf{B}_h$
Diffusion	$\triangle$	$\hat{\triangle} = \mathbf{B}_u^T \triangle \mathbf{B}_u$
Flux to momentum	$\mathbf{P}$	$\hat{\mathbf{P}} = \mathbf{B}_p^T \mathbf{P} \otimes_1 \mathbf{B}_f \otimes_2 \mathbf{B}_g$
Volume	$\mathbf{V}$	$\hat{\mathbf{V}} = \mathbf{B}_p^T \mathbf{V} \otimes_1 \mathbf{B}_u \otimes_{2\ldots4} \mathbf{B}_g$
Boundary	$\mathbf{D}$	$\hat{\mathbf{D}} = \mathbf{B}_d^T \mathbf{D} \mathbf{B}_f$
Induced Boundary	$\mathbf{D}_h$	$\hat{\mathbf{D}}_h = \mathbf{B}_d^T \mathbf{D}_h \mathbf{B}_h$

The reduced space simulation procedure is nearly identical to the full space one (Fig. 4.1), although we do make several small changes. First, we store values of $\hat{\mathbf{h}}$ along deformation trajectories that we will use at runtime and replay these trajectories to find $\hat{\mathbf{h}}$, rather than computing $\mathbf{h}$ from $\hat{\mathbf{g}}$. Second, we integrate advection and diffusion by matrix exponentiation, instead of explicitly as we do in the full space [Treuille et al. 2006]. In the full space, integration by exponentiation would make sure that the simulation is stable for any time step; the fact that $\mathbf{B}_u = \mathbf{B}_p$ ensures that this stability result can be carried over into the reduced space (§4.4.3).

4.4.3 Reduced Stability

The definition of energy in the full space $\mathrm{E}(\mathbf{u}) = ||\mathbf{u}||^2_{\mathbf{V}(\mathbf{g})}$ leads naturally to a definition of reduced energy $\hat{\mathrm{E}}(\hat{\mathbf{u}}) = ||\hat{\mathbf{u}}||^2_{\hat{\mathbf{V}}(\hat{\mathbf{g}})}$, and all our stability arguments from §4.3.5 carry over directly with one exception: advection. In order to ensure energy-preserving advection, we must be careful in basis selection. Constructing the reduced equivalent of Eq. 4.10, we see that the derivative in energy due to reduced-space advection is given by

$$\dot{\mathrm{E}} = \hat{\mathbf{u}}^T \mathbf{B}_p^T \left((\mathbf{A} \otimes_2 \mathbf{B}_f) \otimes_2 \hat{\mathbf{f}} \right) \mathbf{B}_u \hat{\mathbf{u}}. \tag{4.12}$$

If we set $\mathbf{B}_u = \mathbf{B}_p$, then $\hat{\mathbf{u}}^T \mathbf{B}_p^T = (\mathbf{B}_u \hat{\mathbf{u}})^T$, and since the full space matrix $(\mathbf{A} \otimes_2 \mathbf{B}_f) \otimes_2 \hat{\mathbf{f}}$ itself is antisymmetric, we once again are in possession of an energy-conserving discretization. Combined with analytic integration using matrix exponentiation, this basis choice results in an unconditionally stable system. To achieve $\mathbf{B}_u = \mathbf{B}_p$, we first compute the momentum and velocity bases independently. We then concatenate them and re-orthogonalize the result. We use this combined basis for both $\mathbf{B}_u$ and $\mathbf{B}_p$ in our fluid simulations, which guarantees that the simulations will preserve energy.

4.4.4 Constraints

Fluid simulation requires that we maintain full-dimensional constraints exactly in the reduced-dimensional simulation. In particular, we must maintain a hard incompressibility constraint in the reduced simulation. As in [Treuille et al. 2006], all basis vectors of $\mathbf{B}_f$ are divergence-free by construction. They remain divergence-free under geometric deformation, since the units of flux are *volume per unit time*, which are independent of the geometry. However, unlike earlier work, we have multiple bases whose relationships may change with the mesh geometry. In particular, the velocity and momentum bases are not necessarily divergence-free. A reduced projection step is therefore necessary. Reducing the full-dimensional projection (Eq. 4.8), we obtain:

$$\frac{1}{2}||\hat{\mathbf{u}} - (\hat{\mathbf{V}}^{-1}(\hat{\mathbf{g}})(\hat{\mathbf{P}} \otimes_2 \hat{\mathbf{g}})\hat{\mathbf{f}}||_{\hat{\mathbf{V}}(\hat{\mathbf{g}})}. \tag{4.13}$$

Since the flux basis guarantees $\nabla \cdot \mathbf{B}_f \hat{\mathbf{f}} = \mathbf{0}^T$, we can omit the constraint in the reduced projection.

4.4.5 Fluid-Geometry Coupling

To make the flow independent of the changes in the geometry, we proceed as in the full space. We subtract the induced fluxes from the fluxes when computing advection, which, since $\mathbf{B}_f$ and $\mathbf{B}_h$ are different, requires us to generate two reduced advection tensors $\hat{\mathbf{A}}$ and $\hat{\mathbf{A}}_h$ (§4.4.2); boundary constraints $\hat{\mathbf{D}}\hat{\mathbf{f}} + \hat{\mathbf{D}}_h\hat{\mathbf{h}} = 0$ are added to the projection. While the constraints are in the reduced space, we apply constraint reduction (§3.4.1) to ensure that fulfilling the reduced constraints entails fulfilling the corresponding full space constraints exactly; this process also gives us our boundary flux basis $\mathbf{B}_d$.

4.4.6 Runtime Visualization

To visualize the flow field, we advect massless marker particles with the flow. Each particle can be advected separately. Because the velocity state is not necessarily divergence free, we reconstruct advection

velocities from the flux field. We assume that the velocity is constant in each cell. Whenever we need to evaluate a velocity in a cell i, we locally compute

$$\mathbf{u}_i = \left(\mathbf{V}^{-1}(\mathbf{B}_g\hat{\mathbf{g}})(\mathbf{P} \otimes_2 \mathbf{B}_g\hat{\mathbf{g}})\mathbf{B}_f\hat{\mathbf{f}}\right)_i. \tag{4.14}$$

Using this technique, we never need to expand the full representation of the geometry or flux. Instead, for each particle we remember the cell i that currently contains it. We then evaluate only those parts of the geometry that are necessary to compute $\mathbf{u}_i$. We only have to compute the positions of vertices incident to the current element and its neighbors, as well as the fluxes across the faces of the current element. We can then advect the particle with the computed velocity. A particle may leave its current cell, either because the particle is moved by advection, or because the mesh deforms. In that case, we walk across the mesh starting at the particle's last known position, and moving in the direction of the particle's new position. We evaluate the mesh geometry only locally, and continue the walk until we have found an element that contains the particle. For advection, we use explicit Euler integration, with ten substeps per frame for the examples shown in the accompanying video.

4.5 Fluid Evaluation

We evaluate our reduced fluid simulation in two ways: evaluating its numerical error in a 2D wind tunnel domain containing a simple obstacle, and demonstrating its qualitative behavior in two different 3D domains with complex boundary motion.

4.5.1 Numerical Error Analysis

To assess the accuracy of our method, we measure deviation from a full-dimensional ground truth simulation for a simple two-dimensional example. We ran full simulations in a domain consisting of a 2D periodic tunnel-shaped domain containing a single triangular obstacle, and used the simulation frames to compute bases of different sizes. We show some example frames and basis vectors in Fig. 4.4(a-c).

To give a clear picture of how accurately a reduced model built with velocity basis $\mathbf{B}_u$ captures the simulation dynamics over a range of states, we use a *one-step* error measurement. To find the one-step error, we begin with a sequence of velocity snapshots $\mathbf{u}_1, \ldots, \mathbf{u}_T$ from a full-dimensional simulation and project them into $\mathbf{B}_u$ using an energy-conserving projection[2] to find reduced coordinates $\hat{\mathbf{r}}_1, \ldots, \hat{\mathbf{r}}_T$. From each reduced coordinate, we take a full timestep to obtain $\mathbf{u}_{\text{ground}}$ and we take a reduced timestep to obtain $\mathbf{u}_{\text{test}}$. Let the energy $E(\mathbf{u})$ of a velocity field $\mathbf{u}$ be $E(\mathbf{u}) = \frac{1}{2}\mathbf{u}^T\mathbf{V}(\mathbf{g})\mathbf{u}$. The one-step error is then $E(\mathbf{u}_{\text{ground}} - \mathbf{u}_{\text{test}})/E(\mathbf{u}_{\text{ground}})$.

Fig. 4.4(d) plots the one-step error over the evolution of a simulation for a range of basis sizes. Fig. 4.4(e) plots this same error integrated over the entire simulation. In both Fig. 4.4(d) and Fig. 4.4(e), we see that the error tends to decrease as we add more basis vectors. This reassures us that our method converges to the ground truth as the number of basis vectors grows to the dimension of the full simulation. While the decline in error with basis size is monotonic when the error is averaged over time (e), it is not necessarily so instantaneously (d). Our one-step error measure ensures that each simulation begins each step from as close an approximation to the same full state as possible, however, simulations with different basis sizes

[2]The projection is the minimization $\min_{\hat{\mathbf{r}}_t} ||\mathbf{u}_t - \mathbf{B}_u\hat{\mathbf{r}}_t||_{\mathbf{V}(\mathbf{g}_t)}$.

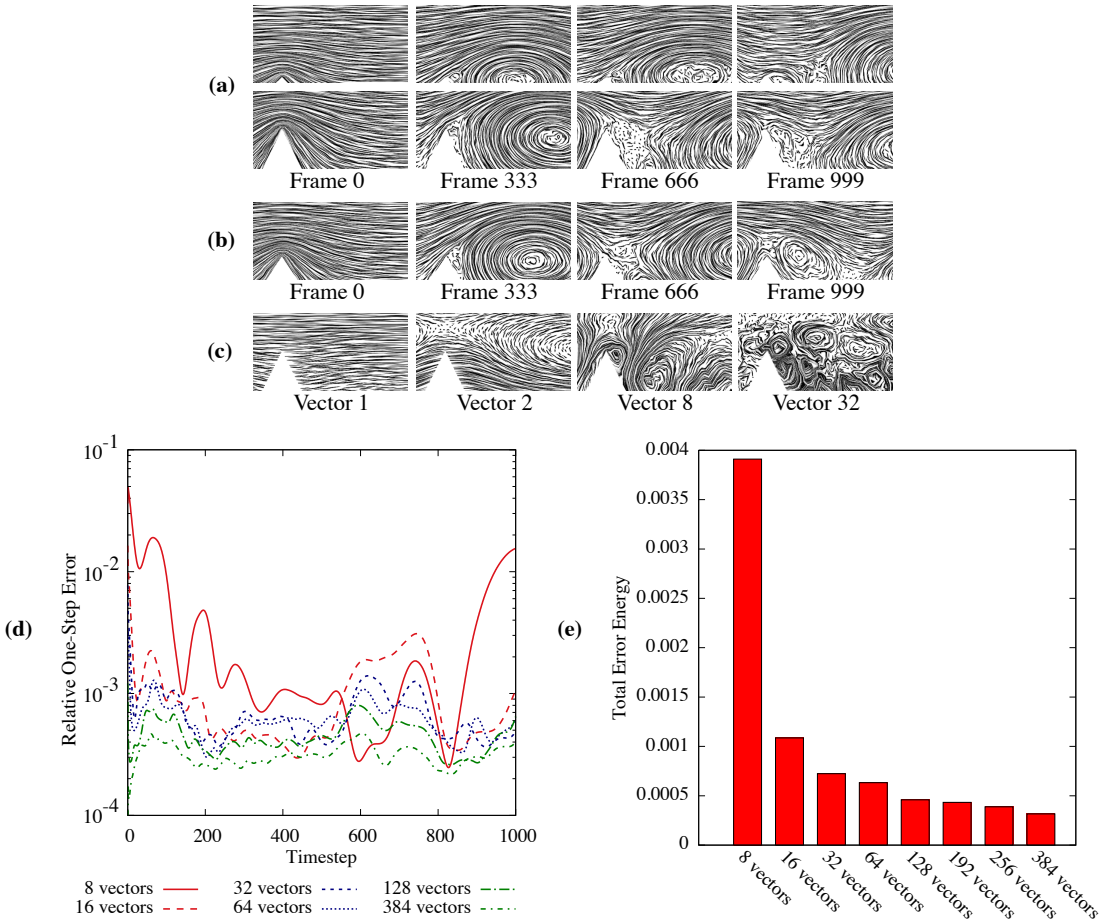


Figure 4.4: *The results of our fluid application error analysis.* ***(a)*** *Frames from the two training simulations: one with a large obstacle, and one with a small obstacle.* ***(b)*** *The corresponding frames from a reduced test simulation in a domain with a medium-sized obstacle.* ***(c)*** *Selected vectors for the velocity basis trained from part (a) and used to run part (b).* ***(d)*** *Relative one-step error over the course of 1000 frames for reduced models with different size bases.* ***(e)*** *Integrated one-step error for models with different size bases. The time-averaged integrated error monotonically decreases with basis size.*

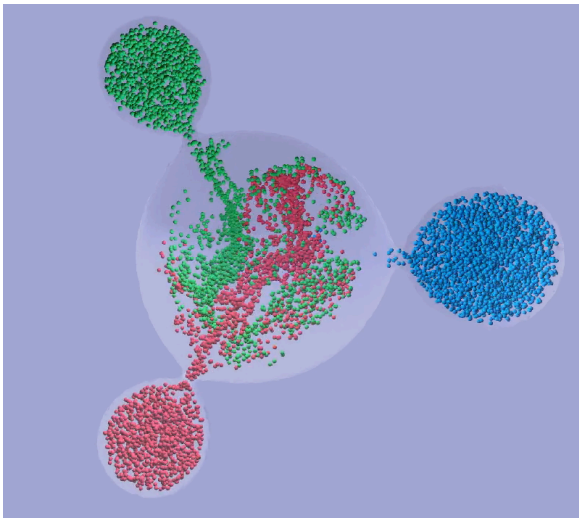


Figure 4.5: *Our method simulates this deformable mixing chamber at 70 frames per second (3 frames per second with rendering).*

	full simulation						reduced simulation							
	dimensions			runtime			basis dimensions			#particles	memory	runtime		speedup
	#cells	#faces	#vertices	t_d	t_a	t_p	m_f	m_u	m_g			t_s	t_{pa}	
Cavity	88671	13339	17823	2.17s	18.4s	220s	64	128	5	10000	29.7MB	0.0145s	0.308s	16565×
Eagle	300217	149903	28613	2.92s	18.9s	82.0s	70	193	7	~ 9500	131MB	0.0469s	0.0381s	2252×

Table 4.1: *Runtimes and statistics for our examples: The table shows timings for full dimensional deformation t_d, advection t_a and projection t_p, as well as reduced simulation t_s and particle advection t_{pa}. It also contains the number of cells, faces and edges determining the dimension of the full simulation, and the reduced basis dimensions of the flux, velocity, and geometry bases, m_f, m_u, and m_g, respectively.*

will start from slightly different initial states due to differences in basis expressivity. Therefore, some deviation from a monotonic decrease in error at some timesteps should be expected.

4.5.2 3D Results

In these results, we used a 110-node cluster with 2.2 GHz SMT quad-core AMD Opteron processors for precomputation, and a 2.6 GHz 8-core Intel Xeon processor with 24GB of memory for realtime simulation, rendering and timing comparison. Our precomputation wall-clock timing results include both PCA and tensor premultiplication time. We used a serial PCA implementation to compute each basis, so only the tensor premultiplication was fully parallelized across the cluster. Our runtime timing results (Table 4.1) include both simulation and particle advection runtime.

Deforming cavity. The *cavity* model (Fig. 4.5) consists of a large central cavity and three adjacent smaller ones, connected with thin tunnels. Each of the small cavities can be individually compressed, causing the fluid inside them to flow into the central cavity, which changes its volume accordingly. We designed the cavity using simple level set primitives, and then tetrahedralized the interior of the isosurface using CGAL [cga].[3] We generated a sequence of example deformations by analytically compressing the outer cavities

[3]Due to the regular shape of the domain, we were able to use a simpler method than the one described in §4.4.1 for generating

while renormalizing the volume and used these deformations to generate 14 full-space simulations, each capturing the compression and re-inflation of one or two small cavities. Not all deformations seen in the video are part of the original training set (for instance, the training set contains no examples where one cavity expands while another remains compressed). Precomputation for this example took 12 hours of wall-clock time, of which approximately the last half hour consisted of tensor premultiplication. At runtime, we can simulate at 70 frames per second, however, due to the dense coverage of particles in the simulation, advecting and rendering particles decreases the frame rate to 3 frames per second. Various forms of optimization which we have not pursued, such as parallelization, could certainly increase the frame rate further.

Eagle. Our eagle example (Fig. 4.3) shows a different use case. Here we started with 266 frames of a triangle mesh animation of a flying eagle flapping its wings and soaring left and right. We built a deformable tetrahedral mesh as described in §4.4.1, generating the base mesh by using Tetgen [Si 2007] to tetrahedralize the space around the eagle with fully extended wings. We used snapshots from a full space simulation run on these same 266 animation frames to build a reduced model. Precomputation for this example took approximately 4 hours of wall-clock time, of which again approximately the last half hour consisted of tensor premultiplication.

Note that the specific deformation sequence need not be preordained; in principle, this method could be extended to model the deformations in an entire motion graph, with the actual character motion determined at runtime. In order to compute a good geometry basis, we do not need more examples or keyframes than it takes to animate a movement sequence. Therefore, no additional modeling work is necessary to turn a rigged model into a reduced deformable fluid model — all steps to create the necessary simulation meshes and bases are fully automated.

4.6 Radiosity

We now apply our model reduction technique to real time computation of global illumination (radiosity) on diffuse deformable objects under varying lighting. As before, we first reformulate the radiosity equation in terms of tensor contractions and matrix operations, and then apply the projection rules in §4.2 to obtain a reduced model. We use this reduced model to interactively explore illumination design in architectural environments.

Following Goral et al. [1984], we divide the scene into discrete patches, and compute radiosity as:

$$\mathbf{b} = (\mathbf{I} - \rho\mathbf{F})^{-1}\,\mathbf{e} \tag{4.15}$$

where $\mathbf{b}$ is a vector of total face radiosity, ρ is a diagonal matrix of albedos, $\mathbf{e}$ is a vector of incident lighting intensities, and $\mathbf{F}$ is a matrix of *form factors*: $\mathbf{F}_{ij}$ describes the fraction of light incident on face j that is reflected to face i. Notice that, unlike fluid simulation, radiosity requires modeling dense interactions between scene components, which increases the computational complexity in a scene with n faces to $O(n^2)$.

the tetrahedral mesh.

4.6.1 Tensor Formulation of Radiosity

Ordinarily, Eq. 4.15 describes a straightforward linear relationship between direct lighting and radiosity. However, if objects in the scene move or deform, this equation becomes highly nonlinear. The crucial term is the form factor matrix $\mathbf{F}$, which we sample at the centroid of each face to obtain

$$F_{ij} = \frac{(\mathbf{n}_i \cdot (\mathbf{c}_j - \mathbf{c}_i))\,(\mathbf{n}_j \cdot (\mathbf{c}_i - \mathbf{c}_j))\,\mathrm{Vis}\,(i,j)}{\pi\,(\mathbf{c}_i - \mathbf{c}_j)^4\,A_i}, \tag{4.16}$$

where $\mathrm{Vis}(i,j)$ is 1 if faces i and j are visible to each other and 0 otherwise, A_i is the area of face i, $\mathbf{c}_i$ is the centroid of triangle i, and $\mathbf{n}_i$ is computed as the vector cross product of two of i's edges.

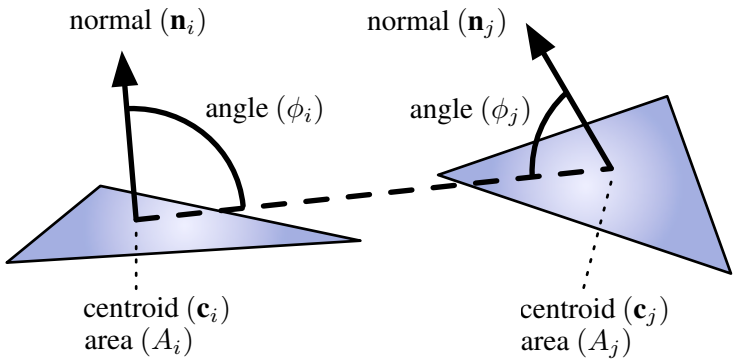


Figure 4.6: *Geometric layout of the illumination variables on two triangles.*

In order to model reduce this equation using our non-polynomial Galerkin projection method, we need to represent it as a composition of tensor products, matrix inverses, and matrix roots. This composition requires more tensors than our fluids application, however, the use of each tensor tends to be simpler. We arrive at this composition by breaking Eq. 4.15 down into progressively smaller parts. These parts are summarized in Table 4.2.

Before we begin, we need to define some notation. Our tensor representation requires that we be able to unroll $\mathbf{F} \in \mathbb{R}^{n\times n}$ into a vector $\mathbf{p} \in \mathbb{R}^{n^2}$ by re-indexing: $p_k = F_{ij}$, where $k = ni + j$ and n is the number of faces in the mesh. We also need the transpose index $k^T = nj + i$, so that $p_{k^T} = F_{ji}$.

The first step in decomposing Eq. 4.15 is to compute the matrix $\mathbf{I} - \rho\mathbf{F}$. We begin by unrolling $\mathbf{F} \in \mathbb{R}^{n\times n}$ into a vector $\mathbf{p} \in \mathbb{R}^{n^2}$ by re-indexing: $p_k = \mathbf{F}_{i,j}$, where $k = ni + j$ and n is the number of faces in the mesh. We also need the transpose index $k^T = nj + i$, so that $p_{k^T} = \mathbf{F}_{j,i}$. The tensor form of the radiosity equation (Eq. 4.15) is then:

$$\mathbf{b} = (\mathbf{I} - \mathbf{E} \otimes_2 \mathbf{p})^{-1}\mathbf{e}, \tag{4.17}$$

where $\mathbf{E}$ is the 3rd-order tensor that transforms the unrolled form factor vector $\mathbf{p}$ back into the form factor matrix $\mathbf{F}$ and left-multiplies $\mathbf{F}$ by the face albedos ρ.

Our task is now to compute $\mathbf{p}$, the vector of form factors. At this stage, we choose to separate the factor of area in the denominator of Eq. 4.16, making this stage a division of *area-free* form factors, denoted by $\mathbf{d}$, by the square roots of squared areas, where we denote squared areas by $\mathbf{a}$:

$$p_k = d_k/\sqrt{a_i}. \tag{4.18}$$

We implement this division using a tensor $\mathbf{P}$:

$$\mathbf{p} = (\mathbf{P} \otimes_2 \mathbf{a})^{-\frac{1}{2}} \mathbf{d}. \tag{4.19}$$

$\mathbf{a}$ is a simple function of the normals $\mathbf{n}$: $a_k = \mathbf{n}_i \cdot \mathbf{n}_i$. (Recall that $\mathbf{n}_i$ is the normal of the ith face, making it a 3-vector.) We define a tensor $\mathbf{N}$ to implement these dot products:

$$\mathbf{a} = \mathbf{N} \otimes_{1\ldots2} \mathbf{n}. \tag{4.20}$$

Returning to $\mathbf{d}$, the next factor we separate is visibility, which gives us the expression

$$d_k = c_k \mathrm{Vis}(i, j), \tag{4.21}$$

where $\mathbf{c}$ are *visibility-free* form factors. We define the visibility vector $\mathbf{v}$ such that $v_k = \mathrm{Vis}(i, j)$. We treat $\mathbf{v}(\mathbf{g})$ as a function of geometry directly, and reduce the computation of $\mathrm{Vis}(i, j)$ using a non-Galerkin method which we describe in more detail in §4.6.3.

Now we can write $\mathbf{c}$ as an element-wise product of two vectors:

$$c_k = \frac{1}{\pi} h_k h_{k^T}, \tag{4.22}$$

where

$$h_k = \frac{\mathbf{n}_i \cdot (\mathbf{c}_j - \mathbf{c}_i)}{||\mathbf{c}_i - \mathbf{c}_j||^2}. \tag{4.23}$$

We call $\mathbf{h}$ *half form-factors*, and define a tensor $\mathbf{C}$ implementing Eq. 4.22:

$$\mathbf{c} = \mathbf{C} \otimes_{1\ldots2} \mathbf{h}. \tag{4.24}$$

Next, we rewrite both parts of the quotient in Eq. 4.23:

$$h_k = \frac{s_k}{r_k}, \tag{4.25}$$

where

$$r_k = ||\mathbf{c}_i - \mathbf{c}_j||^2 \tag{4.26}$$

and

$$s_k = \mathbf{n}_i \cdot (\mathbf{c}_j - \mathbf{c}_i). \tag{4.27}$$

$\mathbf{r}$ is a vector of squared distances between face centroids. $\mathbf{s}$ is a vector of *scaled cosines*: $\mathbf{s}_k$ is the cosine of the angle between $\mathbf{n}_i$ and $\mathbf{c}_j - \mathbf{c}_i$, multiplied by face area and the distance between face centroids. We use a tensor $\mathbf{H}$ to construct a diagonal matrix, with $\mathbf{r}$ as its entries, which we invert to find $\mathbf{h}$:

$$\mathbf{h} = (\mathbf{H} \otimes_2 \mathbf{r})^{-1} \mathbf{s}. \tag{4.28}$$

While $\mathbf{s}$ is polynomial in $\mathbf{g}$, and so it would be possible to compute it directly as a from $\mathbf{g}$ by contracting a single tensor, we can reduce the polynomial degree of this system from 3 to 2 (and the maximum tensor order from 4 to 3) by decomposing $\mathbf{s}$ one step further. Note that Eq. 4.27 is bilinear in $\mathbf{n}$ and $\mathbf{c}$, the latter of which is linear in $\mathbf{g}$. Therefore, we can define a tensor $\mathbf{S}$ such that

$$\mathbf{s} = \mathbf{S} \otimes_1 \mathbf{n} \otimes_2 \mathbf{g}. \tag{4.29}$$

Intermediate Component	Full Space Element Notation	Full Space Tensor Notation	Result Basis	Galerkin Projection
Geometry (vertex positions)		$\mathbf{g}$	$\mathbf{B}_g$	
Normals	$\mathbf{n}_i = (\mathbf{g}_{i,1} - \mathbf{g}_{i,0}) \times (\mathbf{g}_{i,2} - \mathbf{g}_{i,0})$	$\mathbf{n} = \mathbf{N} \otimes_{1\ldots2} \mathbf{g}$	$\mathbf{B}_n$	$\hat{\mathbf{N}} = \mathbf{B}_n^T \mathbf{N} \otimes_{1\ldots2} \mathbf{B}_g$
Scaled Cosines	$s_k = \mathbf{n}_i \cdot (\mathbf{c}_j - \mathbf{c}_i)$	$\mathbf{s} = \mathbf{S} \otimes_1 \mathbf{n} \otimes_2 \mathbf{g}$	$\mathbf{B}_s$	$\hat{\mathbf{S}} = \mathbf{B}_s^T \mathbf{S} \otimes_1 \mathbf{B}_n \otimes_2 \mathbf{B}_g$
Squared Distances	$r_k = (\mathbf{c}_i - \mathbf{c}_j)^T (\mathbf{c}_i - \mathbf{c}_j)$	$\mathbf{r} = \mathbf{R} \otimes_{1\ldots2} \mathbf{g}$	$\mathbf{B}_r$	$\hat{\mathbf{R}} = \mathbf{B}_r^T \mathbf{R} \otimes_{1\ldots2} \mathbf{B}_g$
Half Form Factors	$h_k = s_k / r_k$	$\mathbf{h} = (\mathbf{H} \otimes_2 \mathbf{r})^{-1} \mathbf{s}$	$\mathbf{B}_h$	$\hat{\mathbf{H}} = \mathbf{B}_s^T \mathbf{H} \otimes_1 \mathbf{B}_h \otimes_2 \mathbf{B}_r$
Visibility-Free Form Factors	$c_k = h_k h_k{}^T$	$\mathbf{c} = \mathbf{C} \otimes_{1\ldots2} \mathbf{h}$	$\mathbf{B}_c$	$\hat{\mathbf{C}} = \mathbf{B}_c^T \mathbf{C} \otimes_{1\ldots2} \mathbf{B}_h$
Visibility	$v_k = \mathrm{Vis}(i, j)$	$\mathbf{v} = \mathbf{v}(\mathbf{g})$	$\mathbf{B}_v$	Learned model: see §4.6.3.
Area-Free Form Factors	$d_k = d_k v_k$	$\mathbf{d} = \mathbf{D} \otimes_1 \mathbf{c} \otimes_2 \mathbf{v}$	$\mathbf{B}_d$	$\hat{\mathbf{D}} = \mathbf{B}_d^T \mathbf{D} \otimes_1 \mathbf{B}_c \otimes_2 \mathbf{B}_v$
Squared Areas	$a_i = \mathbf{n}_i \cdot \mathbf{n}_i$	$\mathbf{a} = \mathbf{A} \otimes_{1\ldots2} \mathbf{n}$	$\mathbf{B}_a$	$\hat{\mathbf{A}} = \mathbf{B}_a^T \mathbf{A} \otimes_{1\ldots2} \mathbf{B}_n$
Form Factor Vector	$p_k = d_k / \sqrt{a_i}$	$\mathbf{p} = (\mathbf{P} \otimes_2 \mathbf{a})^{-\frac{1}{2}} \mathbf{d}$	$\mathbf{B}_p$	$\hat{\mathbf{P}} = \mathbf{B}_d^T \mathbf{P} \otimes_1 \mathbf{B}_p \otimes_2 \mathbf{B}_a$
Incident Illumination		$\mathbf{e}$	$\mathbf{B}_e$	
Radiosity	$b_i = \sum_{j=0}^{n} (\mathbf{I} - \rho \mathbf{F})^{-1}_{ij} e_j$	$\mathbf{b} = (\mathbf{I} - \mathbf{E} \otimes_2 \mathbf{p})^{-1} \mathbf{e}$	$\mathbf{B}_b$	$\hat{\mathbf{E}} = \mathbf{B}_e^T \mathbf{F} \otimes_1 \mathbf{B}_b \otimes_2 \mathbf{B}_p$

Table 4.2: *Bases, operators, and Galerkin projections used in our reduced form factor implementation. Radiosity rendering consists of computing this table from top to bottom, either in the full space (columns 2 and 3) or the reduced space (column 5). Note that while the fluids application required only 4 bases, computing the radiosity form factors requires 9 (there are 12 bases listed here, but we constrain* $\mathbf{B}_s = \mathbf{B}_h$, $\mathbf{B}_d = \mathbf{B}_p$, *and* $\mathbf{B}_b = \mathbf{B}_e$*).*

This leaves us with only $\mathbf{n}$ to compute. The normals are given by

$$\mathbf{n}_i = (\mathbf{g}_{i,2} - \mathbf{g}_{i,0}) \times (\mathbf{g}_{i,1} - \mathbf{g}_{i,0}), \tag{4.30}$$

where $\mathbf{g}_{i,\ell}$ is the ℓth vertex of face i and $\times$ is the vector cross product. This expression is a low-degree (quadratic, in fact) polynomial in geometry, and so we can complete the decomposition with a final tensor $\mathbf{N}$ such that

$$\mathbf{n} = \mathbf{N} \otimes_{1\ldots2} \mathbf{g} \tag{4.31}$$

With the tensors defined above, we can now compute radiosity as described in Table 4.2 given only the scene geometry $\mathbf{g}$ and the incident illumination $\mathbf{e}$. Computing $\mathbf{b}$ consists simply of evaluating each row of Table 4.2 in order. Note that the tensor form of the radiosity equation (Eq. 4.17) is the final row of Table 4.2.

This completes the decomposition of Eq. 4.15 into tensors. Now that it is in tensor form, we can reduce it using our non-polyomial Galerkin projection technique.

4.6.2 Reducing the Radiosity Equation

We begin with a set of mesh deformations $\mathbf{G} = \{\mathbf{g}_1, \ldots, \mathbf{g}_m\}$ and for each deformation compute the intermediate vector quantities described in columns 2 and 3 of Table 4.2. We run PCA on each of these vector quantities to obtain the corresponding 10 bases (column 4). Using these bases, we apply our non-polynomial Galerkin projection technique §4.2 to compute reduced form factors $\hat{\mathbf{p}}$. This reduction consists of replacing every tensor in column 3 of Table 4.2 with its reduced equivalent in column 5.

4.6.3 Reduced Visibility

It is not clear how to write a binary discontinuous function like visibility using our tensor formulation, so we handle it separately. Note that computing the visibility of a new deformation at runtime is too slow and

would defeat the purpose of model reduction. Instead, we use the following strategy: first, we compute the visibilities $\mathbf{V} = \{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$ corresponding to the deformation set $\mathbf{G}$, and run PCA on them to form the visibility basis $\mathbf{B}_v$. Then, we determine the reduced visibility $\hat{\mathbf{v}}_{\text{test}}$ of a new deformation $\hat{\mathbf{g}}_{\text{test}}$ as a convex combination of the visibilities in the training set. We find the convex combination $\mathbf{x}$ of training deformations that best predicts $\hat{\mathbf{g}}_{\text{test}}$:

$$\arg\min_{\mathbf{x}} ||\hat{\mathbf{g}}_{\text{test}} - \mathbf{B}_g^T \mathbf{G}\mathbf{x}|| \tag{4.32}$$

$\hat{\mathbf{v}}_{\text{test}} = \mathbf{B}_v^T \mathbf{V}\mathbf{x}$ is the convex combination of reduced visibility vectors with the same coefficients. Note that $\mathbf{B}_g^T \mathbf{G}$ and $\mathbf{B}_v^T \mathbf{V}$ can be precomputed and that their sizes do not depend on the number of vertices in the mesh. This is a reasonable strategy, since two deformations with similar reduced geometry will have similar full-space geometry, and therefore will also have similar visibilities and we are primarily concerned with low-frequency interreflections under area lighting.

4.7 Radiosity Evaluation

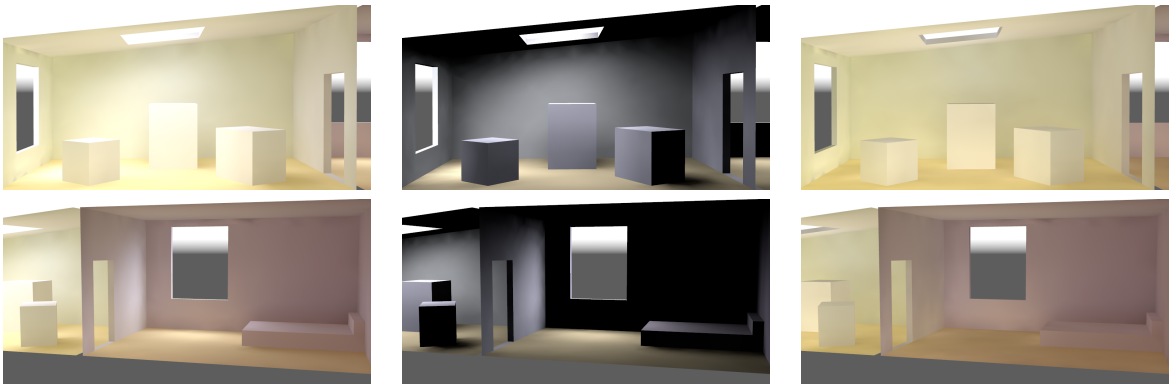
We demonstrate our reduced radiosity method by demonstrating an interactive method for exploring a space of architectural designs in order to achieve certain desired lighting conditions, similar to [Dorsey et al. 1991]. Creating a pleasingly-illuminated space requires careful selection of room arrangements, room sizes, window placements, orientation of the space relative to natural lighting sources, and so forth.

We modeled a scene with a living room and bedroom, consisting of 5012 faces. The living room is brightly-lit with light colored walls, and the bedroom is more dimly-lit with darker walls. In the scene, the sizes of the skylight, windows, and doors can be changed interactively, and the living room ceiling can be tilted.

Our training set for this scene consists of 540 samples, including extremal positions along the axes of the configuration space and samples drawn from the space's interior. We ran PCA on the examples to select basis vectors capturing 99.9% of the variance of each intermediate quantity (Table 4.2, column 4), up to a maximum of 60 vectors. The visibility basis $\mathbf{B}_v$, the area-free form factor basis $\mathbf{B}_d$, and the form factor basis $\mathbf{B}_p$ reached the 60 vector cap; all of the others were able to capture 99.9% of the variance using fewer than 60 vectors. As with fluid simulation, we merged and orthogonalized pairs of bases to comply with our matrix inverse and matrix root basis requirements. The basis pairs we merged are $\mathbf{B}_s$ and $\mathbf{B}_h$, and $\mathbf{B}_d$ and $\mathbf{B}_p$. We used identical 120-vector bases for $\mathbf{B}_e$ and $\mathbf{B}_b$. We ran the basis selection and precomputation on a 8-node Amazon EC2 cluster composed of 8-core 2.67GHz Intel processors with 67.5GB RAM[4], however, both stages were primarily limited by cluster I/O bandwidth. Precomputation took 5 hours 1 minute wall-clock time for PCA, and 1 hour 36 minutes wall-clock time for tensor premultiplication. We rendered two simulation sequences, of 420 and 500 frames, using our reduced model, and compared them with full-space renderings (Fig. 4.7). We rendered each sequence twice, using two different methods for computing the direct illumination. The first method was to compute the direct illumination outside of our reduced radiosity at runtime using physically-based sky model; the second was to compute direct illumination inside of our model by placing area lights in the windows. Fig. 4.8 shows the relative error of these results, which varies from 3% to 11% over the course of the test animations for both methods.

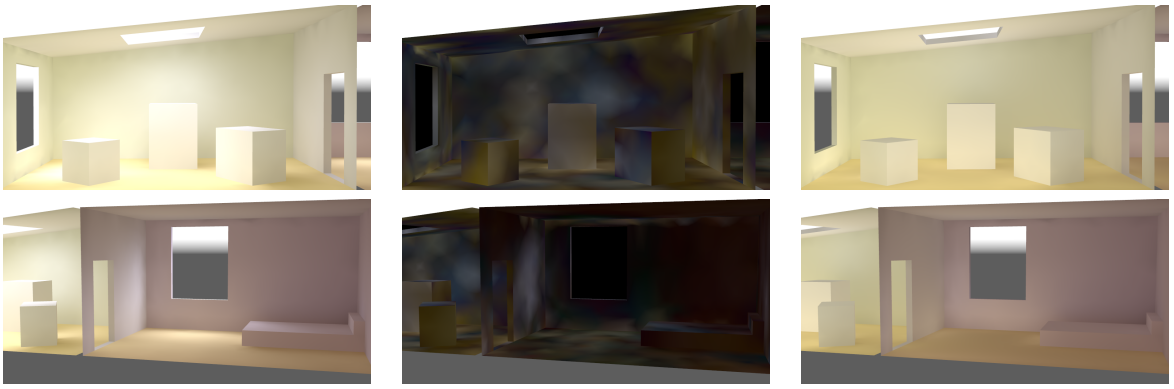
A comprehensive analysis of the space and time requirements for full-space and reduced methods for radiosity depends on the particular algorithm and implementation used. We only discuss relative benefits

[4]Instance type `m2.4xlarge`.



Radiosity | Direct Illumination | Indirect Illumination

(a) Model Reduction



Radiosity | Difference Image | Indirect Illumination

(b) Ground Truth

Figure 4.7: *Results from our architectural rendering example. The scene consists of two rooms, a brightly-lit living room with light colored walls linked by a door to a dimmer bedroom with darker-colored walls. The scene is illuminated by an overcast sky through two windows and a skylight. The scene mesh consists of 5012 faces, and the sizes of the skylights, windows, and doors can be changed interactively. In addition, the living room ceiling can be tilted. (a) Results generated by our model reduced radiosity implementation. (b) Results generated by a full space radiosity implementation. Notice the qualitative similarity between the ground truth and model reduced renderings. Notice the bright spots near the cubes and the foot of the bed, as well as below the windows. There are artifacts in both the reduced and full space renderings due to the coarse meshing near the edges.*

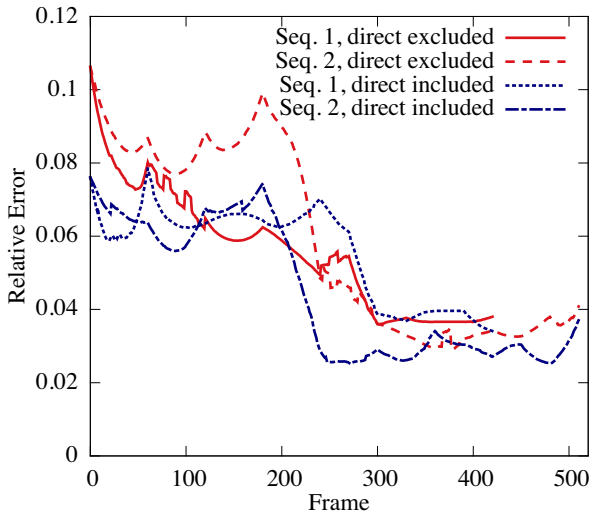


Figure 4.8: *Relative error over the course of the two radiosity animation sequences for direct illumination included and excluded from the model reduction process. Our radiosity basis computation sampled brighter scenes more heavily than dimly lit scenes, so brighter scenes are represented more accurately in the final results.*

with respect to classical radiosity for an interactive application. Our full-space radiosity implementation requires no precomputation time, runs at 0.2 frames per second and requires 200 MB of memory. Our reduced radiosity implementation requires 6 hours and 37 minutes of precomputation time, runs at 22.7 frames per second, and requires only 50 MB of memory. In this setting, our method achieves a runtime speedup of 113 times. If we precompute the form factors, we can load them from disk at 0.4 frames per second, reducing the speedup to 57 times, but at the cost of substantial precomputation time and required storage which, unlike for our reduced method, will grow with the number of frames viewed. Much of the execution time (around 80%) of our method is devoted to computing visibility, which we do not perform using our non-polynomial Galerkin projection technique. We expect that improved methods for computing visibility could dramatically improve our reduced radiosity implementation's performance.

4.8 Limitations

As a data-driven technique, the success of non-polynomial Galerkin projection is limited by the representational power of the precomputed subspaces. If these subspaces do not capture the underlying dynamics well, then the reduced results will likely diverge from the full space results. Both of our applications are particularly susceptible to this form of error, since evaluating our model reduced system at each timestep involves multiple matrix inverses (Fig. 4.1, Table 4.2), each of which minimizes error using a scaled norm that can vary over time and may or may not be well suited to the application at hand (§4.2.2). On the other hand, non-polynomial Galerkin projection is more widely applicable than previous methods, and our results demonstrate that our technique captures complex flow structures and lighting effects in real time. Moreover, error can always be decreased by increasing the size of the basis, at a corresponding polynomial runtime cost.

Our fluid simulation method can compensate for changes in the geometry of the underlying discretization,

and therefore can compute flow through deforming meshes. However, the topology of the discretization must remain fixed. This requires finding a single mesh topology which can accommodate all deformations experienced by the contained geometry. In our experience, using mesh improvement [Klingner and Shewchuk 2007] to create a good base mesh, and applying optimization-based smoothing to obtain good meshes after deformation, can work well — but for extreme deformations, constructing a usable mesh becomes difficult. In particular, we have noticed that at particular points in the eagle animation sequence, large numbers of tetrahedra will suddenly "pop" into new configurations. While our mesh construction methods usually avoid inverting any tetrahedra during this popping, the existence of this behavior suggests that it may not be always be possible to represent every desired deformation using a single mesh topology. An exciting avenue for future work would be to break up the deformation into shorter deformation sequences and use meshes with different topology for each sequence. This might allow us to capture even more drastic deformation, but would require reduced resampling operators to convert the fluid from one geometry basis to another at runtime. Meshing is much easier in the radiosity case, since that application only requires triangle meshes.

We select bases using PCA because it provides good representational fidelity and reasonable results in practice. Unfortunately, PCA bases do not allow us to provide any explicit guarantees about long-term simulation error generated by Galerkin projection, nor error arising from polynomial composition. In the fluids case, using modal bases derived from the simulation operators, as in [de Witt et al. 2012], which uses the modes of the Laplacian as a simulation basis, could allow us to more specifically describe the characteristics of the error.

4.9 Summary

This chapter has shown that Galerkin projection can be extended from polynomials to the much broader class of compositions of *elementary algebraic operations*, enabling the analytic approximation of functions containing addition, subtraction, multiplication, division, and roots, all in closed form. Unlike standard Galerkin projection, our *non-polynomial Galerkin projection* is guaranteed to preserve polynomial degree, essentially bounding the computational complexity of the approximation. Because of the widespread use of model reduction in graphics, we present non-polynomial Galerkin projection in general mathematical terms without specific reference to physical simulation. We believe that our approach can be broadly applied to Galerkin-project functions which previously could not be efficiently approximated.

We showed two different examples of such non-polynomial systems: global illumination of deformable objects, and fluid flow on deforming meshes. Standard Galerkin projection cannot be applied to these phenomena. We demonstrated that non-polynomial Galerkin projection can be applied to both of these phenomena. We also showed that non-polynomial Galerkin projection enables, for the first time, interactive simulations of high-resolution fluid flow around deforming geometry, such as a flying bird, as well as interactive radiosity for lighting design. We believe that this technique could also find use in design and engineering to give interactive feedback about the dynamic effects of geometric changes.

While non-polynomial Galerkin projection can produce marked speed improvements, in practical application it requires careful basis construction to deliver high-quality results. If the system inputs cause the simulation to deviate too far from the regions of the state space explored in the training data, then the resulting simulation can be of poor quality. In the next chapter, we show how to automatically construct high-resolution, interactive simulations that learn to reduce the error that users experience. Specifically,

we demonstrate the construction of a high-resolution, interactive liquid simulation which improves over time to optimize quality for the simulation behaviors that users are most interested in.

Chapter 5

Self-Refining Games

One of the most difficult tasks in constructing any data-driven simulation is selecting which training data will be used in simulation construction.

This chapter describes our work to construct self-refining games, which address the problem of how to select training data for data-driven simulations, and which can produce real-time interactive simulations of almost any phenomenon. Precomputing interactive simulations raises the challenge of anticipating the user: we must precompute training data for all of the simulation states that the user will want to see. However, even with vast computational resources, dynamical spaces are so large that we cannot precompute everything. Fortunately, exhaustive precomputation is unnecessary: user interactions are typically structured and thus explore only a vanishingly small subset of the configuration space. The main challenge is to automatically discover structure from crowdsourced interaction data and exploit it to efficiently sample the dynamical state space.

To address this challenge, we have developed a model *self-refining game* whose dynamics improve as more people play. The gameplay, controls, and objective are simple: the player tilts their mobile device and tries to cause a simulated liquid to splash through a target area of the domain (Fig. 5.1). Points are awarded according to the volume of the fluid passing through the target. Although the game is simple, the dynamics are not: free-surface fluids exhibit rolling waves, droplet sprays, and separating sheets which cannot be simulated and rendered in real time on today's mobile devices.

This data-driven solution is general, applicable to any dynamical system whose controls can be represented as a selection of discrete choices at discrete time intervals. The game is modeled as a *state graph* whose vertices are states and whose edges are short transitions between states. At runtime, the control (in this case, phone tilt) determines which transition to follow. Typically, each transition is simulated, but because we can only precompute a finite number of transitions, some edges *blend* simulations, returning to a previously computed state. Following Kim et al. [2013], the precomputation process interactively grows the state space by successively replacing blend edges with real simulation edges and new states.

The question then becomes: which states should we explore? We show that naïve growth strategies construct vast state graphs that only barely overlap with states explored by real players; these graphs also contain significant visual errors. Using player data, however, enables a novel form of crowd-based sampling which concentrates on those states players actually visit, building significantly better state graphs with far fewer visual artifacts.

Figure 5.1: *An illustration of the gameplay in our liquid game. The player tilts their device left and right to slosh the water back and forth in the box. When the water passes through a goal region in the top center of the box (highlighted in yellow in the middle image), the player scores points.*

Figure 5.2: *Our data-driven approach enables the high quality interactive simulation of free-surface fluids on a mobile device.*

5.1 Related Work

Self-refining games use crowdsourced gameplay data to improve the accuracy and fidelity of the game dynamics. Crowdsourcing has become a major research topic with applications including text recognition [von Ahn et al. 2008], drawing classification [Eitz et al. 2012], and performing user studies [Kittur et al. 2008]. An important subgenre of this research studies *games* which intrinsically motivate players to perform tasks from labeling images [von Ahn and Dabbish 2004; von Ahn et al. 2006] to designing biomolecules [Cooper et al. 2010; Lee et al. 2014]. Crowdsourcing has also been used improve gameplay experience. Zook et al. [2014] tune game parameters based on gameplay traces, the DrawAFriend game [Limpaecher et al. 2013] uses data from previous players to build a drawing improvement engine for later players, and Microsoft Research's Drivatar uses traces from a racing game to improve in-game driving controllers [Microsoft 2013]. Smith et al. [2011] describe a spectrum of different player models; in their taxonomy, our bootstrap model (§5.3.1) is a Universal Synthetic Generative Action model, and our learned model (§5.3.2) is a Universal Induced Generative Action model. Learned player-specific models

have been used to build player-adaptive AI [Houlette 2003], and to generate customized levels [Zook et al. 2012] and stories [El-Nasr 2007; Thue et al. 2007]. Crowdsourced data has also seen application to animation beyond games. McCann and Pollard [2007] used gameplay traces to select transition from a fixed motion graph, and Cooper et al. [2007] focused on computer-in-the-loop sampling of human motion capture data for a fixed set of known objectives. In contrast, we learn models of human player behavior in order to refine game dynamics through adding new transitions to a continually-expanding state graph.

Our playback mechanism is similar to video textures [Schödl et al. 2000], although we are not limited to a single fixed input video. It is also reminiscent of the video-playback mechanics of *Dragon's Lair* [Cinematronics 1983], although since we do not depend on human animators we are capable of generating vastly larger data sets.

The simulation technique underlying self-refining games, the state graph technique described in §2.1, works by tabulating arbitrary dynamics and rendering in an offline process. This approach was pioneered by James and Fatahalian [2003b] who tabulated the dynamics of deformable models driven by a small palette of impulse forces. Kim et al. [2013] extended this approach to cloth dynamics and used far greater computational resources to form a near exhaustive portrait of clothing motion on a moving character. The graph structure and growth process are similar to those of Kim et al., although we adapt these ideas to liquids using a new similarity measure and blend function.

To the best of our knowledge, the only work in graphics on attempting to minimize error in a data-driven simulation in a principled way is by Kim and James [2009], who solves this problem by incrementally building a reduced model for a specific simulation trajectory. By contrast, self-refining games attempt to capture an entire space of trajectories through a continuous state sampling process which uses game analytics to focus on that subset of the dynamics that players really explore. Self-refining games achieve this goal by estimating state visit probabilities using an algorithm, StateRank, which computes the stationary distribution of a Markov chain [Feller 1968] with transition probabilities derived from a player model, similar to the PageRank algorithm of Page et al. [1999].

5.2 State Graphs

I now describe a method for creating self-refining games based on learned player behavior. The foundation for these games is the *state graph*, whose vertices are game states, and whose edges are transitions induced by player actions. These graphs are similar to the secondary motion graphs of Kim et al. [2013]. A game could be represented by a single graph, or could use multiple graphs to represent different areas, levels, or challenges. To initialize a new state graph we use a heuristic player model to bootstrap synthesis of a minimally playable experience. Then, we make the game available to players and collect traces of the paths they take through this graph during play. We use these paths to learn a model of player behavior that is used to prioritize graph growth, adding states and increasing fidelity in those regions players are most likely to visit. By repeatedly collecting player data, updating our player model, and using the updated model to grow the graph, we create a game that continually improves over time. This process is general, and can be applied to any precomputed game with discrete control.

In this section, we describe our game model and the mechanics of graph growth. In the next section (§5.3), we describe how we learn player behavior models from collected player data. The following section (§5.4) applies this general framework to free-surface fluid simulation.

In a state graph, each edge is associated with an animation connecting its source and destination states.

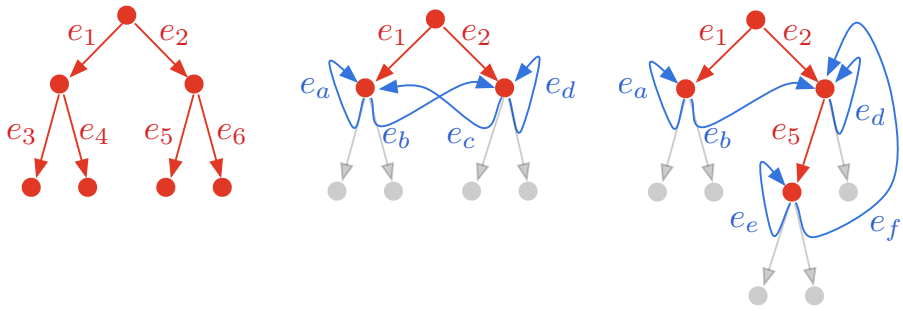


Figure 5.3: *State graph initialization and growth. Solid edges correspond to simulations and dashed edges correspond to blends. Left: We initialize the graph by sampling a tree of simulation data. Center: We transform the tree into a complete state graph by replacing dead-end edges with blends. For example, e_c is formed by blending dead end e_5 with e_1. Right: We expand the graph to remove error along blend e_c by removing e_c, re-inserting e_5, and simulating new animations from e_5 for each control. The new simulations form new dead ends that are blended to create e_e and e_f. A self-refining game repeats this last step continuously.*

The format of these animations is application-dependent, but they could be video clips, sequences of triangle meshes, or any other encoding of the dynamics that we wish to display. We consider games that sample player input from a set of N discrete controls, so that each vertex has N outgoing edges. At runtime, the system replays edge animations, determining which branch to take based on player control. If the transition edges are sufficiently short (1/3 of a second in our application) the game feels interactive. We discuss the choice of transition edge durations further in §5.7.

To initialize a new state graph, we begin at a start state and simulate every possible outgoing transition. We continue this process, generating N new simulations from each state until we create a small N-ary tree (Fig. 5.3, left). Leaf edges represent dead ends in the state graph which we eliminate by blending with interior edge transitions leading back to an internal node. This blending procedure turns the tree into a complete state graph (Fig. 5.3, center). Since we begin with only a small tree, it is likely that many of these blends were between dissimilar edges, and therefore of low quality.

We improve the quality of the graph by growing it using new simulation data, similar to [Kim et al. 2013]. We grow the graph by replacing blend edges with simulation edges. To add a simulation edge e to the graph, we simulate the N outgoing transitions which follow e and blend these simulations into interior simulation edges (Fig. 5.3, right) selected to minimize an estimate of the perceptual error of the blend. Growing the graph can be a continuous process, going on as long as we have space to store the results of new simulations. The key challenge is determining which blend edges to replace.

We can view this question as one of graph quality evaluation. If we can quantify each edge's contribution to the quality of the graph, a simple strategy to reduce error is to greedily replace the blend edge most detrimental to the quality of the graph. The behavior of this strategy depends strongly on which measure of graph quality is selected. Kim et al. [2013] use a worst-case quality measure: $\max_{e \in B}(\mathrm{err}(e))$, where B is the set of blend edges in the graph and err is an application-defined estimate of a blend edge's perceptual error. This metric, which we call BASELINE, suggests that we should always replace the blend edge $e_{\max}$ with the highest error.

In a simulation-based game, however, the game occupies a huge state space, and the game objective encourages players to pursue strategies that lead to rewards. Therefore, it is likely that players will never visit

the vast majority of the state space, rendering most of BASELINE's additions to the graph wasteful. Instead, we propose a different metric, STATERANK, which measures the *expected* error: $\sum_{e \in B} P(e)\mathrm{err}(e)$, where $P(e)$ is the probability of traversing the edge e. This metric suggests we replace the blend edge e_{exp} with maximum expected error $P(e_{\mathrm{exp}})\mathrm{err}(e_{\mathrm{exp}})$. We infer $P(e)$ from a *player model* $P(c|v)$ giving conditional probabilities of controls c at each vertex v. Similar to the PAGERANK [Page et al. 1999] procedure for ranking webpages, STATERANK computes edge probabilities $P(e)$ as the normalized first eigenvector of the transition matrix implied by $P(c|v)$.

If STATERANK correctly predicts edge probabilities, then this procedure will improve graph quality around precisely those states players are most likely to visit. To accurately estimate $P(e)$, however, we must have accurate estimates of the player control probabilities $P(c|v)$. In the next section we discuss how we learn this player model $P(c|v)$ from analytics data.

5.3 Player Model

The previous section explains how we can use a player model $P(c|v)$ to improve our sampling of huge state spaces. In this section we describe both how we can learn player models from data and how we use these models to create self-refining games. We use two different player models: a heuristic model to bootstrap the simulation, and a learned model to guide our exploration.

5.3.1 Bootstrap Model

When the game is first created, no player data exists. To bootstrap state graph growth, we use a heuristic player model $P_h(c|v)$ which essentially guesses what players will do. Many heuristics are possible, and the best heuristic will vary by application. In this study, we maintain the current control with probability α and otherwise choose an alternate control uniformly at random:

$$P_h(c|v) = \begin{cases} \alpha & \text{if } c = c_v \\ (1-\alpha)/N & \text{otherwise.} \end{cases} \tag{5.1}$$

Combining this heuristic player model with STATERANK produces a growth strategy we call SR-HEURISTIC. This simple model can initialize a state graph, but performs poorly when used exclusively to generate a full game (§5.6.2). We therefore propose using the heuristic model only for bootstrapping, then growing the graph using a player model learned from gameplay traces.

5.3.2 Player Analytics Model

Once we have a bootstrap model, we can begin to collect gameplay traces to learn a more accurate player model. We learn our player model from traces of player traversals of the state graph, each trace consisting of a list of vertices visited and the control selected at each vertex. Let $P_{\mathrm{obs}}(c|v)$ be the observed conditional control probabilities computed by normalizing control counts at v, and $P_{\mathrm{obs}}(v)$ be the observed probability of visiting v, obtained by normalizing the number of visits to v by the total number of vertex visits. To generalize our model to unvisited states, we assume that players will take similar actions in states that

resemble each other closely. This observation leads us to implement our player model using a kernel density estimator combined with a Markov prior with weight ϵ:

$$P(c|v) \propto \sum_{\substack{u \in V \\ c_u = c_v}} w_u P_{\text{obs}}(c|u) P_{\text{obs}}(u) \tag{5.2}$$

$$w_u = k_{\text{tri}}\left(r, \text{pdist}\left(u, v\right)\right) + \epsilon,$$

where $k_{\text{tri}}(r, x)$ is a triangular kernel with radius r, c_u and c_v are the controls of the simulation clips generating u and v, V is the set of vertices in the graph, and pdist is an inexpensive distance function (§5.4.2). Note that the condition $c_u = c_v$ in the summation effectively creates a different player model for each control. Also observe that this model can be evaluated even when v has not been visited by any player. As a result, this model can be used to guide sampling deep into the graph without having to wait for new player data at every step. The model can even be used to transfer predicted player behaviors gathered on one graph to explorations of other similar graphs.

Combining this player model with our StateRank technique described in the previous section yields a crowdsourced graph quality measure, SR-Crowd, which we can use to select blend edges to replace as we grow the graph. We evaluate state graphs generated using this player model in detail in §5.6.

5.4 Application to Liquids

In this section, we describe our construction of a liquid simulation game using the generic game precomputation framework that we described in §5.2. Our liquid simulations are generated using PCISPH [Solenthaler and Pajarola 2009]. We represent the liquid state at graph vertices v as lists of liquid particle positions and velocities, and k-frame animations along graph edges as sequences of signed distance functions $e = \left[\phi^1, \ldots, \phi^k\right]$, generated from particle data using the method of Zhu and Bridson [2005]. Each edge also has an associated video rendered using Mitsuba [Jakob 2010]. In our application $k = 10$, which yields transitions of 1/3 of a second. This latency in response to changes in player control is acceptable for our liquids game, however, different game mechanics entail different latency requirements. [Claypool and Claypool 2010]

We use two different metrics. For blending, we use a function $\text{dist}(e_i, e_j, c)$ based on energy and detailed liquid shape information (§5.4.1). Our player model, however, requires more frequent distance computations, so we use a more efficient metric $\text{pdist}(e_i, e_j)$ which compares only coarse shape descriptors (§5.4.2). Finally, we describe our clip blending function $\text{blend}(e_i, e_j)$ in §5.4.3.

5.4.1 Edge Distance

We define $\text{dist}(e_i, e_j, c)$ to be a perceptually-motivated error function incorporating information both about the liquid's shape and its energy:

$$\text{dist}(e_i, e_j, c) = \text{norm}_e(e_i, e_j)\left(\text{dist}_s\left(e_i, e_j\right) + w_e \text{dist}_e\left(e_i, e_j, c\right)\right). \tag{5.3}$$

Here, dist_s and dist_e denote the parts of distance attributable to the shapes and the energies of the two states, respectively; norm_e is a normalization term that increases distance at low energies, reflecting that fact that errors are easier to perceive when the liquid is moving more slowly. The weight w_e controls the

relative priority of the shape and energy terms of dist. We set w_e so that for edges r_i and r_j where the fluid is nearly at rest,

$$\mathrm{dist}_s\left(r_i, r_j\right) \approx w_e \mathrm{dist}_e\left(r_i, r_j, c\right).$$

Shape distance. The dist_s metric penalizes the blending of animations which contain liquid in very different shapes. It is the sum of the volumes of the symmetric difference ($X \triangle Y = X \cup Y \setminus X \cap Y$) between each animation's liquid volumes at each frame:

$$\mathrm{dist}_s(e_i, e_j) = \sum_{f=1}^{k} \mathrm{vol}\Big(\phi_i^f \triangle \phi_j^f\Big). \tag{5.4}$$

Energy distance. The dist_e metric penalizes the blending of animations that have very different energies, and it strongly penalizes blending an animation with low energy into an animation with high energy, thus enforcing conservation of energy. Omitting dist_e can result in the formation of small loops in the state graph far away from energy minima, which look extremely unnatural.

We define energy at a vertex v as $E(v, c) = T(v) + V(v, c)$, where T is kinetic energy, V is potential energy and c is the incoming control. Notice that energy depends on the current control since selecting a gravity vector will change the potential energy. Let v_i and v_j be the destination vertices (final frames) of e_i and e_j. The energy error between edge e_i and e_j is given by

$$\mathrm{dist}_e(e_i, e_j, c) = \gamma |E(v_i, c) - E(v_j, c)| \tag{5.5}$$
$$\gamma = \begin{cases} c_{\text{gain}} & \text{if } |E(v_i, c) - E(v_j, c)| < T_0 \\ c_{\text{loss}} & \text{if } |E(v_i, c) - E(v_j, c)| \geq T_0 \end{cases}$$

where $c_{\text{gain}} \gg c_{\text{loss}}$, and T_0 is approximately the residual kinetic energy of the fluid when it is visually at rest. We attempt to match the energies as closely as possible, rather than anticipating an energy loss, since we are comparing energies between two clips at identical points in time. We place the threshold between the minor energy loss penalty and the major energy gain penalty at T_0 to avoid penalizing blends between visually indistinguishable animations of static fluid.

Energy normalization. We normalize the previous two terms by multiplying them by norm_e. Let v_i and v_j again be the destination vertices of e_i and e_j, c_i and c_j be their controls, and $T_{\text{avg}} = \frac{1}{2}\left(T(v_i) + T(v_j)\right)$. Then

$$\mathrm{norm}_e(e_i, e_j) = \begin{cases} 0 & \text{if } T_{\text{avg}} < T_0 \text{ and } c_i = c_j \\ \frac{1}{\sqrt{T_{\text{avg}} + T_0}} & \text{otherwise.} \end{cases} \tag{5.6}$$

Note that this implies that the distance between two edges with the same control and kinetic energy below T_0 is 0. Again, this threshold prevents the unnecessary exploration of liquid states that are visually at rest. This unnecessary exploration would otherwise consume the vast majority of our exploration effort since norm_e goes to infinity as the fluid energy goes to zero.

5.4.2 Player Model Distance

StateRank requires us to perform neighbor searches using a brute-force scan of all vertices in the graph, so the function we use to compute vertex distances must be fast. We therefore use a more efficient coarse

shape similarity function pdist in our player model. We compute a shape descriptor d_i for each edge e_i by dividing the fluid domain into a $6 \times 6 \times 6$ grid and computing the average fraction of each cell that is occupied by liquid, and define

$$\mathrm{pdist}(e_i, e_j) = ||d_i - d_j||_2. \tag{5.7}$$

5.4.3 Blending

We construct animations for blend edges by blending signed distance functions. A simple linear interpolation works well in cases where the fluid surfaces do not contain many fine features. However, in the presence of droplets, splashes, and thin sheets, linear interpolation can cause popping artifacts at the beginning and end of the blend. We remedy this problem by blending, using convex combinations of three signed distance functions: the source, ϕ_s, the destination, ϕ_d, and the union of their shapes, $\min(\phi_s, \phi_d)$. We use the following blend function, where $0 \leq t \leq 1$ denotes the position in the blend, clip clips its argument to lie between 0 and 1, and ℓ is a parameter that limits the blending coefficient applied to the union:[1]

$$\begin{aligned} \mathrm{blend}(\phi_s, \phi_d, t) &= w_s \phi_s + w_d \phi_d + w_{s \cup d} \min(\phi_s, \phi_d) \\ w_s &= \mathrm{clip}\left((1 + \ell - 2t)/(1 + \ell)\right) \\ w_d &= \mathrm{clip}\left((2t + \ell - 1)/(1 + \ell)\right) \\ w_{s \cup d} &= 1 - w_s - w_d \end{aligned} \tag{5.8}$$

In our implementation, we use $\ell = 0.1$ to avoid perceptible increases in liquid volume during blends.

5.5 Implementation

We constructed a distributed simulation system to carry out the large-scale state graph explorations required for our work. This system consists of a pool of worker nodes, which perform simulation and render animations, and a master node, which orchestrates computation by maintaining the graph structure, computing edge priorities and distances, and assigning work to the workers. We deploy this system on Amazon EC2 in configurations featuring up to 40 worker nodes. In total for our high-viscosity and low-viscosity experiments, this system performed over 8600 CPU-hours of computation and generated over 1.6 TB of data, at a cost of approximately $500 in compute time.

The system initializes graph exploration with a minimal state graph containing one vertex and N edges. Graph expansion then proceeds as described in §5.2. During exploration, the master maintains a work queue enumerating blend edges to explore. When workers become available, the master extracts the highest priority blend from the queue and assigns the corresponding simulation tasks to a worker. When workers return simulation results to the master, the master computes blends for the newly simulated edges, then updates the graph. In StateRank-based explorations, the master periodically discards blend-edge priorities, recomputes StateRank on the current graph, then uses the results to re-initialize blend edge priorities.

[1] The results in our supplementary video show an incorrect version of the blend function that shortens the blend time by one frame at each end of the transition.

5.5.1 Optimizations

A number of key optimizations were necessary to achieve high system performance, as well as to ensure high-quality graphs.

Lazy relinking. Each time a new edge is added to the graph, nearest neighbor relationships among edges in the graph may change. Since it would be costly to recompute optimal graph blends after each new inserted edge, our implementation does not attempt "relinking" of existing blend edges when new edges are created. Instead, prior to exploring any blend edge, the system first attempts to relink this edge with existing edges in the graph. If a superior blend can be found in the graph at this time, the new blend is immediately created and used to replace the existing blend. Lazy relinking ensures that simulation is only performed on edges for which there are no good blend candidates in the graph, but it does not incur the overhead of unnecessarily reevaluating edge nearest neighbors after new edges are simulated.

Pre-publish relinking. In our experiments we make graph data available for play ("publishing") at select checkpoints during exploration. In order to provide the best possible play experience, before playing a graph we attempt to relink every blend edge. This process ensures that all graph blends are the best possible blends given available data at the time of play.

Animation caching. The primary scaling bottleneck of our system is the high cost of distance computations during edge nearest-neighbor search. These computations must be performed by the master each time a worker returns new simulation data (to create blends), and they are expensive because they require fetching fluid-volume occupancy data from network storage. Rather than incur substantial system complexity by parallelizing the master node's execution, we were able to accelerate distance computations needed for nearest neighbor search by caching voxel occupancy information in memory (our implementation uses `memcached`).

Energy pre-filtering. Even with the caching optimizations described above, nearest-neighbor search remains expensive. (For example, linear-time nearest neighbor search makes graph relinking a quadratic-time operation, which is unacceptable even for graphs of moderate size.). To accelerate nearest-neighbor search we only perform full distance evaluation on the k-closest graph edges according to our energy distance metric described in §5.4.1. It is important to set k appropriately; setting k too low can result in a failure to find good blends even if high-quality blend targets do exist in the graph. We have found $k = 100$ to work well in practice.

5.5.2 Mobile Client

We make games available to players using an Android client application. The key feature of this client is its ability to continuously play back short ($1/3$ second) videos without lag between them. When a player selects a game, the client downloads the most recent version of the game's state graph, then downloads and caches any videos for edges in the current graph that it has not already cached. We use device accelerometer data to select game controls. After each play session, the client uploads a list of visited graph vertices and the control selected on each visit to our server.

5.6 Evaluation

We use the self-refining liquid control game described at the beginning of this chapter to evaluate the utility of StateRank and player models learned through crowdsourced data to explore a large dynamic space. Recall that the player's objective in this game is to interactively tilt their device so as to splash fluid through a target region of the domain. The game admits three possible controls (N=3) corresponding to holding the device level and tilting it to the left and right. A sequence of screen shots from the game is shown in Fig. 5.1, with the target region in the upper-middle part of the space highlighted in the middle frame.

We grew state graphs for our fluid game using three different graph error measures. The SR-Heuristic and SR-Crowd measures described in §5.3 prioritize growth using StateRank and either a heuristic or crowdsourced player model, respectively. The simplified Baseline measure does not use StateRank to globally prioritize exploration. Instead, it only prioritizes growth using local conditional control probabilities, $P(c|v)$. To provide a better comparison with StateRank, we scale the maximum error in our Baseline metric by the conditional probabilities in Equation 5.1 with $\alpha = 0.8$, but we do not calculate edge probabilities as in SR-Heuristic or SR-Crowd.

All explorations use the energy-based fluid distance metric and fluid animation blending techniques described in §5.4. Our fluid simulations are 42K-particle PCISPH simulations. In addition to the results analyzed here, we performed experiments on a high viscosity fluid configuration in order to observe system behavior under a second set of fluid parameters; we show results in the video.

We grew each of our graphs until graph size reached 200K frames. (Approximately 4,300 CPU-hours were used, per graph, to compute each graph's 1.8 hours of animation). We paused graph expansion at 10K, 20K, 50K, 100K, and 200K frames so that the graphs could be played by a group of six test players, yielding gameplay traces for all graphs at these checkpoints. After collection, the checkpoint traces for the SR-Crowd graph were used to calculate per-vertex conditional probabilities that informed the exploration of SR-Crowd until the next checkpoint.

5.6.1 Predicting Player Behavior

To assess the predictive power of our models, we used the 200K-frame Baseline graph to evaluate SR-Heuristic's and SR-Crowd's predictions of player behavior and overall graph quality against the observed data from gameplay. Fig. 5.7 (left, center) illustrates predicted play behavior by coloring graph edges according to predicted visit densities. At right, we show the observed probabilities from player data (ground truth). It is clear that SR-Crowd predicts behavior far more accurately than SR-Heuristic. This entails a better estimate of the expected blend error that players encounter: SR-Crowd's prediction is within 10% of the observed value, while SR-Heuristic underestimates it by nearly 50%. Note also that very little of the graph is actually visited by players since Baseline has explored many "unimportant" regions of the state space. Guiding exploration with player data stands to create a graph that better samples the played regions.

Fig. 5.4 shows the 200K-frame state graphs produced by the Baseline, SR-Heuristic, and SR-Crowd; the graphs indeed exhibit noticeably different structure. Specifically, the Baseline graph is approximately twice as deep as the StateRank-produced graphs. While the Baseline graph samples many long sequences of constant-control play (as per the heuristic model), our game's design does not

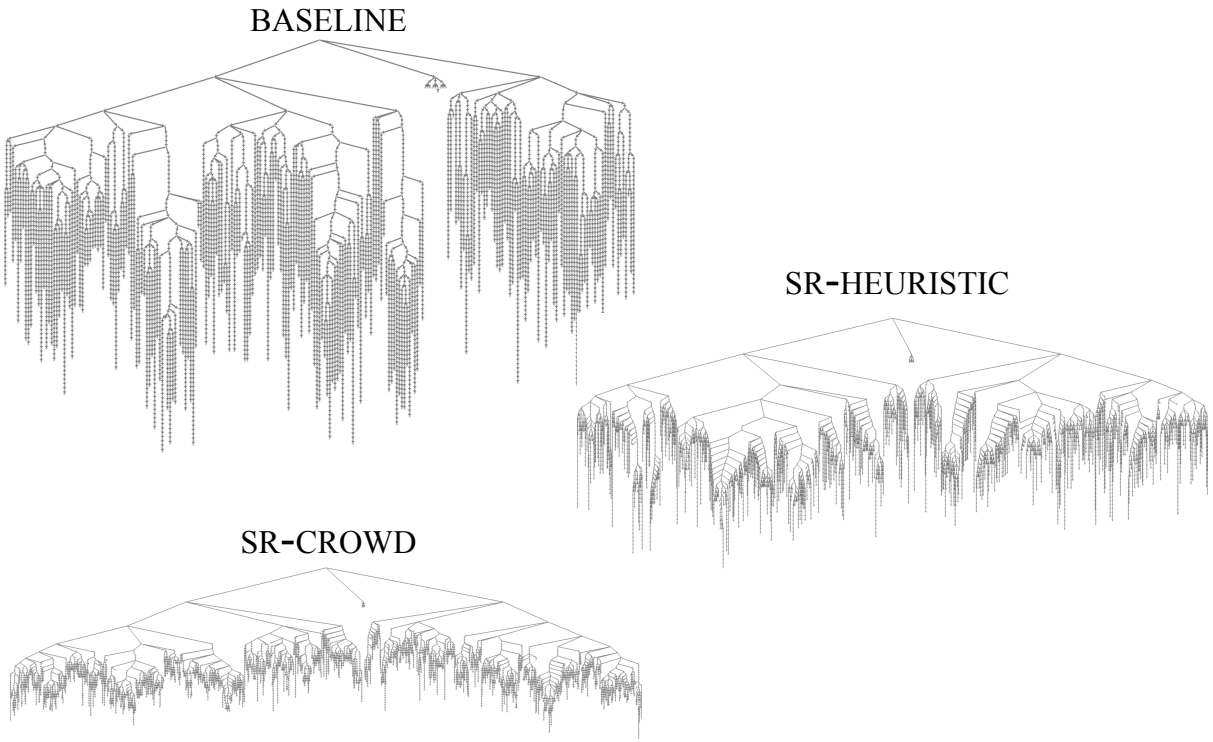


Figure 5.4: *200K-frame state graphs.* BASELINE *explores long chains of low energy states. In contrast, both* SR-HEURISTIC *and* SR-CROWD *prioritize more likely high energy states, yielding a shallower graph structure.*

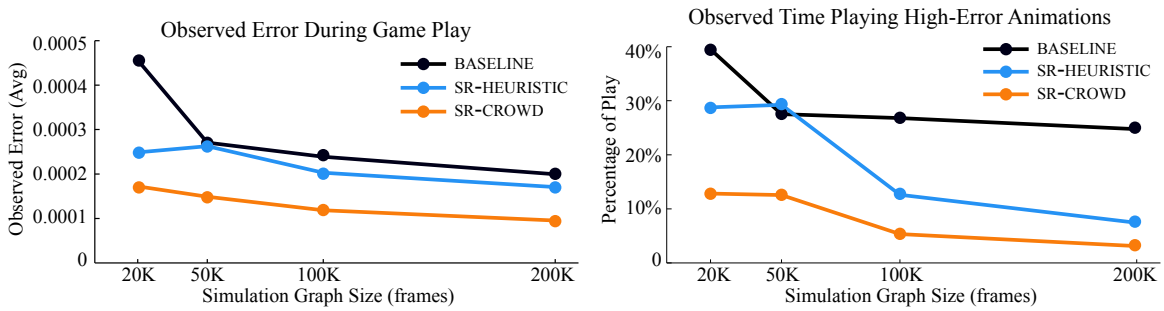


Figure 5.5: *Comparison of errors observed by players on graphs generated using different methods. Left: On average, test players observed the lowest error while playing the sequence of graphs generated by* SR-CROWD *Right: Animations observed by players of the 200K-frame* SR-CROWD *graph exceeded our empirical high-error threshold during only 3% of play time. Gameplay for the similarly-sized* BASELINE *graph presented high-error animations ten times as often.*

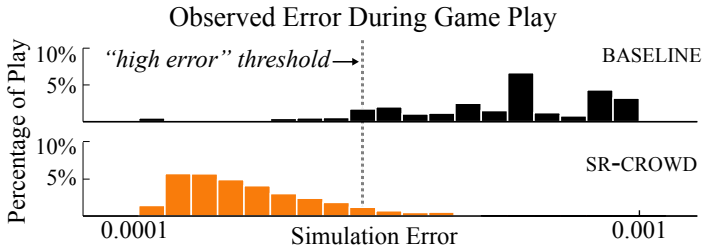


Figure 5.6: *Histogram of blend errors observed by players on two of the 200K-frame graphs. In contrast to* BASELINE*, the error distribution for* SR-CROWD *is largely concentrated below our high-error threshold.*

encourage this form of play, so these paths do not appear in the SR-CROWD graph.

The unique structure of the SR-CROWD state graph results in a higher quality game experience. Fig. 5.5, left, plots the average error observed by players at all graph sizes. (Error is defined using the perceptually-motivated metric described in §5.4.) While SR-HEURISTIC alone provides a modest benefit over the BASELINE method, average observed error is nearly a factor of two lower for SR-CROWD games. The gameplay data acquired at each checkpoint during the graph growth process helps focus simulation effort on the state space regions that players are most likely to encounter.

5.6.2 Error Analysis

While state explorations prioritized by STATERANK successfully reduce observed error on average, they risk leaving severe blend edges in the graph if they are deemed unlikely to be visited. While small animation errors are difficult for a player to notice, large errors, even if infrequent, can significantly reduce the perceived quality of the game. Through gameplay testing, we empirically determined that blend animations with error scores exceeding 0.0005 corresponded to visually objectionable animations. We plot the fraction of time that players spent viewing these high-error animations in Fig. 5.5, right.

In the large 200K-frame graph produced by SR-CROWD, players view high-error animations significantly

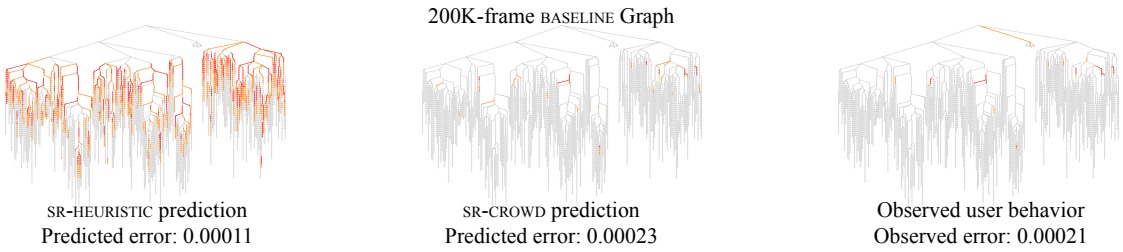


Figure 5.7: *Visualization of edge-visit densities predicted by* SR-HEURISTIC *(left), by* SR-CROWD *(center), and the densities from recorded gameplay (right) for a 200K-frame* BASELINE *graph. Edges visited at least 10% as frequently as the most visited edge are red, edges visited 2-10% as frequently are orange, and all others are gray. Blend edges are hidden to reduce clutter.* SR-CROWD *accurately predicts player visit densities while* SR-HEURISTIC *does not, indicating the value of player analytics in informing* STATERANK*'s probability estimates.*

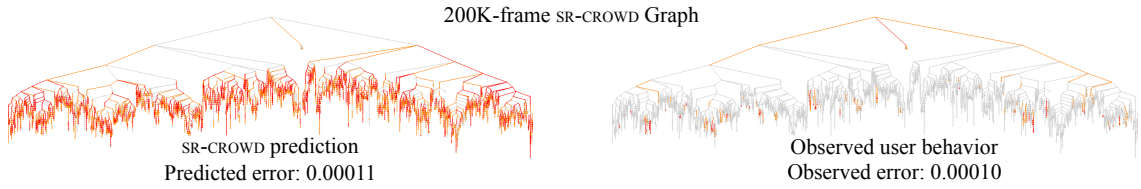


Figure 5.8: *Edge-visit densities predicted by* SR-CROWD *(left) and the recorded densities from actual gameplay (right) for a 200K-frame* SR-CROWD *state graph. While its prediction of the expected edge blend error remains accurate,* SR-CROWD *no longer predicts player behavior on this graph well, suggesting an opportunity for better prediction models based on the acquired player data.*

less often than in the other methods (only 3% of the time). In fact, gameplay for the small 20K-frame SR-CROWD graph showed high-error animations less often than play through a BASELINE graph ten times as large. The fraction of observed high-error frames does not significantly diminish in the BASELINE graphs as they grow beyond 50K frames. A more detailed view of the distribution of errors encountered when playing 200K-frame BASELINE and SR-CROWD graphs is provided in Fig. 5.6.

Our experiences playing the games corroborate these numerical results. In all cases, playing the SR-CROWD games showed the highest quality animation. Play through the 200K-frame SR-CROWD graph for high viscosity simulations reveals virtually no artifacts at all. In low viscosity simulations, the fluid exhibits more geometric variety, and artifacts are occasionally visible (as indicated by Fig. 5.5, right), but their frequency is greatly reduced in SR-CROWD compared to other methods. As desired, animation is of the highest quality when the player plays with intention, attempting to score maximum points, and thus conforming closely to the actions of previous players. We refer the reader to the accompanying video to inspect the quality of the generated animations.

As a final experiment, we also measured SR-CROWD's ability to predict player behavior on the 200K-frame graph that it produced. This prediction is compared to ground truth observed player data in Fig. 5.8. Interestingly, although SR-CROWD still produces an expected edge error within 10% of the observed value, its predictions of player visit behavior on this final graph are relatively poor as compared to its predictions on the baseline graph (Fig. 5.7). We hypothesize that although our player model is accurate, self-referentially growing a graph based on this model amplifies small inaccuracies, even while signifi-

cantly lowering error. Thus, Fig. 5.8 suggests that creating more sophisticated models of player behavior from the acquired play data could aid in sampling the game state space even more efficiently.

5.7 Limitations

Our game was designed as a simple research vehicle both to explore the potential of leveraging crowd-sourced player data to efficiently sample large state spaces, and to demonstrate a new form of self-refining game using complex 3D liquid dynamics as a primary element. Our results demonstrate that models built from player data can concentrate precomputation in an important (but tiny) subset of the full state space, a result that overcomes a major hurdle in scaling data-driven techniques. Nevertheless, our current approach has several limitations which we hope further research can address.

Total dynamic complexity. To our knowledge, our fluid example represents one of the most complex systems ever precomputed at a large scale. However, even simple generalizations of the dynamics would overwhelm our system. For example, inserting floating objects would explode the state space. We emphasize that our completely monolithic technique of precomputing everything about a simulation state represents just one (extreme) end of a spectrum of approaches. Precomputed and live elements can be decomposed, composited together, and even coupled. Precomputed systems could also be generalized (e.g., using multi-way blends), potentially turning our discrete state graph representation into a continuous space of precomputed dynamics.

Limits of control. Our system offers only a small number of discrete controls and samples control to 1/3 of a second. Increasing the temporal or spatial control resolution not only explodes the state space, but also causes specific technical problems. Because our state graphs have complete N-way branching at every vertex, the blend edge fraction is $(N-1)/N$. In the limit of increasing control resolution, nearly every clip will be a blend. We believe the solution is to sparsely sample control, inserting control branches only when player data indicates they are necessary. At run-time, the system would trade off simulation and control error. It may be possible to even scale this approach to (multi-dimensional) continuous controls. Similarly, clip length has implications for blending. As the clip length approaches one frame, blends become jump discontinuities. We believe this issue could be addressed by globally optimizing state graphs to ensure smooth transitions.

Range of applicable phenomena. Our approach assumes it is possible to meaningfully blend two simulations without obvious visual artifacts. Fluids work well because the eye often ignores errors in turbulence. Some phenomena might be less forgiving. We observe, however, that blending has been successfully applied to a wide range of phenomena, from human motion graphs to image morphing. Blending should work for any sufficiently well-sampled continuous phenomenon, making the ability to densely sample important subsets of the state space – the very goal of this paper – even more important.

Single-viewpoint rendering. We chose to include rendering in our precomputation to achieve rich visual effects. This decision constrains our game to single-viewpoint rendering. However, rendering could be decoupled from the precomputation and performed either on the server or the client. A server-based rendering system would stream rendered images from any viewpoint. Client-based rendering is also possible, although efficient compression of 3D data would be required, a particular challenge for fluid data with temporally changing surface topology.

Storage requirements. Our 200K-frame state graphs correspond to about 5 GB of data. Unlike our prototype, practical implementations of complex data-driven games would likely rely on cloud-based an-

imation storage and streaming. We used off-the-shelf video compression for simplicity, but our video corpus contains enormous redundancy across clips that standard video compression methods do not exploit. Deduplicating similar video sequences (perhaps by linear dimension reduction, or by applying frame prediction between clips) could potentially yield vast savings.

Applicability to existing games. Some types of open-world games encourage exploration and the discovery of new experience and content. However, many categories of games do encourage stereotyped player behavior, and even games that prioritize novelty will likely feature substantial overlap in player behaviors that our method can exploit.

5.8 Summary

This chapter presents a first step towards *self-refining games* whose dynamics continuously improve based on player analytics. We observe that game objectives cause players to explore only a small fraction of the entire state space, making data-driven simulation feasible even for complex dynamical systems. We adapt the data-driven simulation method of Kim et al. [2013] to liquids, and replace the precomputation phase with a continuous process that concentrates state sampling in the subset of the dynamics that players really explore.

We compare three strategies to sample the game dynamics and show that using real player data (SR-CROWD) significantly outperforms both a more simplistic player model (SR-HEURISTIC) and a baseline model without player data (BASELINE). Interestingly, even our best player model significantly mispredicts player actions (Fig. 5.8), suggesting that further improvements are possible. Nevertheless, our results strongly indicate that player data can be successfully exploited to capture very complex dynamical systems.

Our method is well suited to mobile platforms with limited control precision and computational capacity. Player-driven state sampling enables us to deliver high quality rendered content in realtime with bounded simulation error. In addition to improving existing games, these ideas could enable a new class of cloud-based games where designers no longer have to worry about simulating and rendering the world in fractions of a second.

The ideas presented in this chapter suggest several interesting questions and generalizations. How much player data is required to sufficiently sample a space? How does adding states affect the difficulty and the strategy of the game? How can we adapt our sampling approach to applications beyond games? Further research could yield more powerful techniques to composite precomputed dynamics models like ours with other virtual elements, create more flexible models through decomposition, decouple rendering from simulation, and address other limitations (§5.7).

Chapter 6

Conclusion

Data-driven techniques are one of the most powerful tools computer graphics has for creating detailed, interactive simulations. They allow massive amounts of offline computational power to be brought to bear on interactive applications, enabling even very highly detailed simulations to run in real-time. In this document, we have provided three solutions for problems that limit the use of data-driven simulation in graphics.

Chapter 3 described a method for generating scalable data-driven simulations with domains that can be reconfigured at runtime. By creating a collection of data-driven simulation tiles that act as building blocks for our desired domains, we can construct domains that would be too large for conventional simulations, or build a huge assortment of different domains without re-training. The key technical challenge in this chapter was the enforcement of constraints between neighboring simulation tiles. To solve this challenge, we introduced the *constraint reduction* technique, which modifies reduced bases to enable the enforcement of linear inequality constraints in Galerkin-projected simulations. We demonstrated an application of our method to fluid simulation, but its general nature paves the way for coupling of all kinds of reduced simulations. In fact, Chapter 4 uses constraint reduction to enforce boundary non-penetration constraints in fluid simulations on deforming meshes. In the future, we hope to see this approach extended to phenomena such as explosions, elastic dynamics, and free-surface fluids – or even coupling different types of reduced models in the same simulation. This technique could also potentially be adapted to work with the non-polynomial Galerkin projection technique presented in Chapter 4, and a similar domain decomposition approach could perhaps be applied to self-refining games (Chapter 5) to enable them to capture much larger scenes.

Chapter 4 addressed one of the central limitations of analytic Galerkin projection: its restriction to purely polynomial dynamics. By extending Galerkin projection to also handle divisions and roots, we extended the scope of Galerkin projection to a huge swath of new phenomena. These phenomena include not just the two that we demonstrated, fluids and radiosity in continuously deformable domains, but also n-body gravitational systems and atomic-scale Lennard-Jones interactions. The existence of non-polynomial Galerkin projection also encourages us to search for further methods to analytically capture, simulate and render the many complex, high-dimensional phenomena in the world around us.

Finally, Chapter 5 presented *self-refining games*, which provide a principled approach to collecting training data for interactive data-driven simulations. By observing players interacting with a simulation game, we can determine which parts of the state space they want to visit at runtime, and concentrate our training

data sampling in those areas. Future work could extend these precomputed physics models by making them more flexible using domain decomposition (like we did with Galerkin projection in Chapter 3), or by allowing precomputed state graph models to interact with traditionally-simulated virtual elements at runtime. Another exciting avenue of future work would be to combine self-refinement with a different underlying data-driven simulation technique, such as Galerkin projection.

Our ultimate goal is to create immersive, interactive simulations of *anything*. Data-driven simulation is a vital tool for achieving this goal. It allows for incredible detail while maintaining interactive response times, and it is ideally suited to an increasingly heterogeneous computational landscape. Mobile devices are rapidly becoming faster, and large-scale cloud compute clusters are available cheaply and on-demand, while the console, the PC, and other devices intermediate in computational capacity are becoming less important. Data-driven simulation makes it possible to utilize cheap offline computation to enhance realtime interactive experiences, even on devices with limited computational capability.

The work that we have presented in this document greatly enhances the reach of data-driven simulation, but it is just a beginning. Data-driven simulation still has many limitations – in scale, in authoring, and in the types of phenomena that can be simulated – that must be overcome. We believe that two of the directions that we have begun to explore in this document will be particularly important in the future development of simulation in graphics. One of these is increasing modularity in data-driven simulation, like we did with our our modular bases technique, to scale simulations to huge domains, incorporate more detail, and allow for greater runtime flexibility while maintaining interactive response times. The other is approaching simulation design and construction from a human factors perspective, as we did with self-refining games, in order to create efficient simulations that prioritize the user experience over simple physical definitions of accuracy and adapt to how they are actually used over time.

6.1 Modular Data-Driven Simulation

Data-driven simulation's strengths include its abilities to represent huge amounts of detail in real-time simulations, and to permit efficient use of heterogeneous computation resources in an increasingly diverse computational landscape. There are still many limits to its scalability and versatility, however. Very large scenes will often offer too many possibilities and too large a state space to be sampled adequately during training. Large scenes also require large state representations and reduced models, making it necessary to process large volumes of data at runtime, especially when it is necessary to capture very localized or sparse user interactions. Furthermore, there is still a large gap between the range of phenomena that we can simulate using traditional methods and using data-driven methods, a problem that is exacerbated by the fact that it can be difficult to couple data-driven and traditional simulations in the same scene.

The root of the problem, in some sense, is that data-driven simulation is usually a monolithic proposition – either capture the entire scene in a single simulation, or avoid a data-driven techniques. Monolithic data-driven simulations must have very large state spaces to produce good results, and, by definition, do not couple to other simulations. Their large state spaces make them difficult to train, since it is becomes harder and harder to predict what users will do as the range of possible user actions increases. Even learning the important parts of the state space from users, like we did with self-refining games, may not be of much help if the state space becomes too complex. Large scenes may also incorporate some scene elements that could be more efficiently simulated using traditional techniques.

We suggest three possible avenues for addressing these problems: domain decomposition, coupling tradi-

tional and data-driven simulation, and local perturbations.

Domain decomposition. Domain decomposition can help keep state space sizes manageable by dividing a simulation in space and limiting the size of each piece. Keeping each piece small reduces the quantity of data that each piece requires for training. Being able to assemble the pieces into larger simulations, however, allows the combined simulation to cover a very large state space. Chapter 3 demonstrated the power of this technique for Galerkin-projected fluid simulations. Allowing self-refining games to be similarly constructed from parts would enable them to handle much larger state spaces. Other extensions to domain decomposition, such as being able to continuously move or blend simulation tiles at runtime, could allow for much more efficient simulations of complex scenes.

Coupling traditional and data-driven simulations. Data-driven interactive simulations and traditional interactive simulations excel in different areas. Data-driven simulations are well-suited to large compute clusters, large detailed scenes, and scenes with restricted user interaction. Traditional simulations are well-suited to small scenes without too much detail, are best run on local devices, but do not place many limits on the types of user interaction they can support. Coupling traditional and data-driven simulation techniques would also allow complex scenes to be simulated using the most appropriate simulation technique for each type of dynamics. Consider a simulation of multiple rigid objects floating in water. Rigid bodies are easy to simulate using traditional methods, and we demonstrated precomputed water in Chapter 5, but including floating rigid bodies in our water game would make the state space too large to sample effectively. If we could couple a traditional rigid body simulation to our precomputed liquid simulation, we could avoid storing the exact state of each rigid body in the liquid simulation, greatly reducing its state space and hopefully rendering it feasible for precomputation.

Local perturbations. One particular weakness of data-driven simulation is their handling of sparse interactions, where the user can introduce large changes to a small part of the simulation domain. To support this type of interaction, data-driven simulations need to be able to represent these changes at every single point where the user could introduce them, which can make the reduced representation impracticably large. Returning to the coupled liquid and rigid body simulations from the previous paragraph, it seems unavoidable that the introduction of rigid bodies into the water should increase the amount of precomputation required for the water simulation, even if the rigid bodies are computed separately at runtime. If a brick is thrown into the water, for example, a splash should appear at precisely the spot where the brick contacted the surface. Even for a single brick, sampling all possible splash locations becomes an enormous burden, since we must also sample the time evolution of each splash, at least to the point where one splash becomes indistinguishable from the next. Ideally, we would not include the splashes in our training data at all, restricting our reduced representation to low-frequency fluid states only, and introduce splashes at runtime as local perturbations that would be incorporated into the precomputed simulation over a short period of time. Harmon and Zorin [2013] proposed a method for temporarily incorporating localized bases into Galerkin-projected simulations of deformable body dynamics. However, no analog exists for fluid simulation, or for any simulation using our state graph representation.

The suggested future directions in this section would make data-driven simulation a more useful tool for creating interactive experiences by increasing its power. There is another way to increase its utility, however: by creating simulations optimized specifically for human perception and interactive use.

6.2 Human-Centered Simulation Design

The inspiration for using data-driven simulation techniques in graphics derives substantially from the success of these techniques in disciplines outside of graphics, especially engineering. The goals of interactive simulation in graphics and of typical engineering simulations are substantially different, however. An engineering simulation is generally intended to answer a question about physics: how much drag a particular airplane wing design will generate, or how much load a bridge can support without breaking. Interactive simulations in graphics, on the other hand, do not exist primarily in order to generate physically accurate outcomes, but to create perceptually plausible animation to be consumed by humans. We can describe these two different types of simulations as *physics-centered* and *human-centered*, respectively.

The distinction between human and physics-centered simulations is not a bright line. Rather, it is intended to illuminate two different sets of priorities that can be taken into account when creating a simulation. These priorities are in tension, but they are not mutually exclusive. Simulations built while primarily holding one or the other set of priorities do tend to share some common traits.

Data-driven simulations, as approximations to traditional simulations, all seek to minimize the error that their approximation incurs. One difference between physics-centered and human-centered data-driven simulations is reflected in the errors that these simulations seek to avoid. Physics-centered applications seek to measure some specific numerical property as accurately as possible. This goal naturally induces a definition of error, relative to which the performance of the simulation can be optimized. Simple error functions, such as $\mathcal{L}^2$ error, are often reasonable choices, and tend to correspond to useful physical quantities such as energy. Human-centered applications are more directly concerned with perceptual notions of error, which do not necessarily map nicely to such simple formulations.

Human-centered simulations are also typically more error-tolerant. If errors are neither too frequent nor too large, they will not interfere substantially with the user experience. It is therefore not necessary for a simulation to have hard error bounds for it to be useful; it is necessary only that the error typically be low. In physics-centered simulations, however, error bounds are frequently of paramount importance, particularly if a simulation is being used to evaluate the efficiency or safety of a real-world object.

Finally, and perhaps most obviously, human-centered simulations are intended to be used by people – typically many different people. Whether they appear in games, art, education, or some other venue, their entire purpose is to respond convincingly to user inputs. Physics-centered simulations, on the other hand, may not be used by anyone at all: they may appear in control systems or other automated processes. The smaller user population for physics-centered simulations also means that their inputs are typically more predictable.

Most simulations in graphics are constructed using physics-centered techniques, including our Galerkin-based simulations from Chapters 3 through 4. (For example, these simulations use $\mathcal{L}^2$ error both to select their bases and to construct their reduced dynamics tensors.) Using physics-centered simulation techniques in human-centered applications can work well, because physical error is often a good proxy for perceptual error. However, optimizing for simple physical error measures can be inefficient, since in general such measures are not perceptually valid, and simulation creators may invest substantial effort eliminating errors that users will never notice but are weighted highly using the physical measure.

Adopting human-centered simulation design has, we believe, the potential to produce simulation-based interactive experiences far more immersive, convincing, and enjoyable than would be possible with physics-centered simulation. Self-refining games are strong evidence for this statement: their design is strongly

human-centric, and, despite being a poor fit for engineering or other traditional simulation applications, they produced beautiful interactive 3D liquid simulations that other types of data-driven simulation cannot emulate. Here, we suggest three ways in which future data-driven simulations could be improved by prioritizing human factors in their designs.

Better models of perceptual error in physical systems. Defining useful perceptual error measures for dynamical systems can be a difficult and complex process, as we demonstrated in §5.4. Even our perceptual error metric there was defined in an ad hoc manner. Experimentally validated measures of perceptual error in physical simulations would make data-driven simulations more efficient and help avoid the process of trial and error that is currently required to develop application-specific perceptual error estimates. Better perceptual understanding could also help guide the choice of other simulation design decisions, like representation selection.

More trajectory-based simulation representations. One major difference between the Galerkin-based data driven techniques in this paper and the state graph technique of self-refining games is the nature of the parts that comprise their reduced models. In Galerkin-based techniques, the fundamental unit of representation is the simulation snapshot. The reduced model consists simply of a collection of snapshots that can be combined to yield new states. In the state graph model, the fundamental unit of representation is the *trajectory*, that is, a short segment of simulation behavior in time. This difference is similar to that between dynamic and kinematic methods for character animation. As with kinematic animation methods, using trajectories in reduced representations allows the simulation to avoid explicitly handling physics calculations at runtime, which allows both the physics equations *and* the error metrics to take any form. Also, if only perceptually convincing trajectories are incorporated into the reduced model, and good blending techniques are chosen, then we can guarantee that the user will never see an implausible animation. (A complementary line of work would be more advanced blending methods that can generate plausible animation by combining even very different trajectories, such as the liquid blending technique of Raveenendran et al. [2014]). However, reliance on recorded trajectories hurts generalizability, especially the ability to generalize to small perturbations from the training data, and as such may make them less appropriate for engineering or other physics-centered tasks.

Adapting to user behavior. Perhaps the most powerful way to prioritize human factors in simulation design is to let simulations actually adapt to user behaviors in order to achieve their design goals. This was the key insight of self-refining games. The creation of more advanced and more accurate player models would of course allow self-refining games to even more efficiently explore simulation state spaces. Simulations could lower the error that users perceive not just by providing high-quality results, but by actually introducing limited amounts of error into the simulation to guide users towards better-explored regions of the state space. Customized simulations could also provide different dynamics to each user, either to conform to user-specific models of simulation error, or to achieve application-specific goals unrelated to error (such as maximizing playing enjoyment in a game).

6.2.1 Final Thoughts

We are excited about the future of data-driven simulation. Data-driven techniques, including Galerkin projection and cubature, have already proven to be useful tools for crafting interactive experiences, and we hope that our self-refining games will join their ranks. Self-refining games can create easy-to-analyze simulations of even very complex phenomena, and they are currently the only method we are aware of that can interactively simulate high-resolution 3D liquids on mobile devices. Data-driven simulations can

create increased simulations of unparalleled scale and detail, and are uniquely well-positioned to exploit large-scale cloud computation. In the future, we expect modular, human-centered data-driven simulations to allow game creators, designers, educators, and artists to create almost any interactive experience they acn dream of.

Bibliography

ADAMS, B., PAULY, M., KEISER, R., AND GUIBAS, L. J. 2007. Adaptively sampled particle fluids. In *Proc. SIGGRAPH '07*.

AN, S. S., KIM, T., AND JAMES, D. L. 2008. Optimizing cubature for efficient integration of subspace deformations. *ACM Transactions on Graphics 27*, 5 (Dec.), 165:1–165:10.

ANDO, R., THUEREY, N., AND WOJTAN, C. 2013. Highly adaptive liquid simulations on tetrahedral meshes. *ACM Transactions on Graphics (SIGGRAPH) 32*, 4.

ANGELIDIS, A., AND NEYRET, F. 2005. Simulation of smoke based on vortex filament primitives. In *Proc. SCA '05*.

ANGELIDIS, A., NEYRET, F., SINGH, K., AND NOWROUZEZAHRAI, D. 2006. A controllable, fast and stable basis for vortex based smoke simulation. In *Proc. SCA '06*.

AUSSEUR, J., PINIER, J., GLAUSER, M., AND HIGUCHI, H. 2004. Predicting the dynamics of the flow over a NACA 4412 using POD. *APS Meeting Abstracts*, D8.

BABUŠKA, I. 1973. The finite element method with Lagrangian multipliers. *Numer. Math. 20*, 3.

BANKS, H., DEL ROSARIO, R. C., AND SMITH, R. C. 2000. Reduced-order model feedback control design: numerical implementation in a thin shell model. *IEEE Transactions on Automatic Control 45*, 7, 1312–1324.

BARAFF, D., AND WITKIN, A. 1992. Dynamic simulation of non-penetrating flexible bodies. *Computer Graphics (SIGGRAPH 92) 26*, 2 (July), 303–308. ISBN 0-201-51585-7. Held in Chicago, Illinois.

BARBIČ, J., AND JAMES, D. 2005. Real-time subspace integration for St. Venant-Kirchhoff deformable models. In *Proc. SIGGRAPH '05*.

BARBIČ, J., AND POPOVIĆ, J. 2008. Real-time control of physically based simulations using gentle forces. *ACM Transactions on Graphics 27*, 5.

BARBIČ, J., AND ZHAO, Y. 2011. Real-time large-deformation substructuring. *ACM Trans. on Graphics (SIGGRAPH 2011) 30*, 4, 91:1–91:7.

BENZI, M., GOLUB, G. H., AND LIESEN, J. 2005. Numerical solution of saddle point problems. *Acta Numerica 14*, 1–137.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *Proc. SIGGRAPH '03*.

BORGGAARD, J., GUGERCIN, S., AND ILIESCU, T. 2006. A domain decomposition approach to POD. *IEEE Conference on Decision and Control*.

CARLBERG, K., BOU-MOSLEH, C., AND FARHAT, C. 2011. Efficient non-linear model reduction via a least-squares Petrov-Galerkin projection and compressive tensor approximations. *International Journal for Numerical Methods in Engineering 86*, 2, 155–181.

CARLBERG, K. 2011. *Model Reduction of Nonlinear Mechanical Systems via Optimal Projection and Tensor Approximation*. PhD thesis, Stanford University.

CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

CHADWICK, J., AN, S. S., AND JAMES, D. L. 2009. Harmonic shells: A practical nonlinear sound model for near-rigid thin shells. *ACM Transactions on Graphics 28*, 5 (Dec.), 119:1–119:10.

CHAHLAOUI, Y., AND VAN DOOREN, P. 2005. Model reduction of time-varying systems. In *Dimension Reduction of Large-Scale Systems*. Springer, 131–148.

CHAI, M., ZHENG, C., AND ZHOU, K. 2014. A reduced model for interactive hairs. *ACM Trans. Graph. 33*, 4 (July).

CHENNEY, S. 2004. Flow tiles. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, SCA '04, 233–242.

CHENTANEZ, N., AND MÜLLER, M. 2010. Real-time simulation of large bodies of water with small scale details. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, 197–206.

CHENTANEZ, N., AND MÜLLER, M. 2011. Real-time eulerian water simulation using a restricted tall cell grid. *ACM Trans. Graph. 30* (August), 82:1–82:10.

CINEMATRONICS, 1983. Dragon's Lair. [Arcade].

CLAYPOOL, M., AND CLAYPOOL, K. 2010. Latency can kill: Precision and deadline in online games. In *Proceedings of the First ACM Multimedia Systems Conference*.

COHEN, M. F., SHADE, J., HILLER, S., AND DEUSSEN, O. 2003. Wang tiles for image and texture generation. In *Proc. SIGGRAPH '03*.

COOPER, S., HERTZMANN, A., AND POPOVIĆ, Z. 2007. Active learning for real-time motion controllers. *ACM Trans. Graph. 26*, 3 (July).

COOPER, S., KHATIB, F., TREUILLE, A., BARBERO, J., LEE, J., BEENEN, M., LEAVER-FAY, A., BAKER, D., AND POPOVIĆ, Z. 2010. Predicting protein structures with a multiplayer online game. *Nature 466* (August).

COUPLET, M., BASDEVANT, C., AND SAGAUT, P. 2005. Calibrated reduced-order POD-Galerkin system for fluid flow modeling. *J. Comput. Phys. 207*, 1.

CRANE, K., LLAMAS, I., AND TARIQ, S. 2007. *Real Time Simulation and Rendering of 3D Fluids*. Addison-Wesley, ch. 30.

DE WITT, T., LESSIG, C., AND FIUME, E. 2012. Fluid simulation using Laplacian eigenfunctions. *ACM Transactions on Graphics 31*, 1 (Jan.).

DEBUSSCHERE, B. J., NAJM, H. N., PEBAY, P. P., KNIO, O. M., GHANEM, R. G., AND MAITRE, O. P. L. 2004. Numerical challenges in the use of polynomial chaos representations for stochastic processes. *SIAM J. Sci. Comp. 26*, 2, 698–719.

DECONINCK, H., AND RICCHIUTO, M. 2007. Residual distribution schemes: foundation and analysis. In *Encyclopedia of Computational Mechanics*, E. Stein, E. de Borst, and T. Hughes, Eds., vol. 3. John Wiley and Sons, Ltd.

DOBES, J., AND DECONINCK, H. 2006. An ALE formulation of the multidimensional residual distribution scheme for computations on moving meshes. In *Proc. Int. Conf. CFD*.

DORSEY, J., SILLION, F., AND GREENBERG, D. 1991. Design and simulation of opera lighting and projection effects. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, 41–50.

DRETTAKIS, G., AND SILLION, F. 1997. Interactive update of global illumination using a line-space hierarchy. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, 57–64.

DURAND, F., DRETTAKIS, G., AND PUECH, C. 1999. Fast and accurate hierarchical radiosity using global visibility. *ACM Trans. Graph. 18*, 2 (Apr.), 128–170.

EBERT, F., AND STYKEL, T. 2007. Rational interpolation, minimal realization and model reduction. Tech. rep., DFG Research Center Matheon.

EDWARDS, E., AND BRIDSON, R. 2014. Detailed water with coarse grids: Combining surface meshes and adaptive continuous Galerkin. *ACM Trans. Graph. 33*, 4 (July).

EITZ, M., HAYS, J., AND ALEXA, M. 2012. How do humans sketch objects? *ACM Trans. Graph. 31*, 4 (July), 44:1–44:10.

EL-NASR, M. S. 2007. Interaction, narrative, and drama: Creating an adaptive interactive narrative using performance arts theories. *Interaction Studies 8*, 2 (June), 209–240.

ELCOTT, S., TONG, Y., KANSO, E., SCHRÖDER, P., AND DESBRUN, M. 2005. Stable, circulation-preserving, simplicial fluids. In *Discrete Differential Geometry*, Chapter 9 of Course Notes. ACM SIGGRAPH.

ELCOTT, S., TONG, Y., KANSO, E., SCHRÖDER, P., AND DESBRUN, M. 2007. Stable, circulation-preserving, simplicial fluids. *ACM Transactions on Graphics 26*, 1 (Jan.).

FARHAT, C., TEZAUR, R., AND TOIVANEN, J. 2000. A domain decomposition method for discontinuous Galerkin discretizations of Helmholtz problems with plane waves and Lagrange multipliers. *Int. J. Numer. Meth. Engng*.

FARHAT, C., HARARI, I., AND FRANCA, L. P. 2001. The discontinuous enrichment method. *Comput. Methods Appl. Mech. Engrg. 190*.

FARHAT, C., HARARI, I., AND HETMANIUK, U. 2003. A discontinuous Galerkin method with Lagrange multipliers for the solution of Helmholtz problems in the mid-frequency regime. *Applied Mechanics and Engineering 192*, 1389–1419.

FARHOOD, M., AND DULLERUD, G. E. 2007. Model reduction of nonstationary LPV systems. *IEEE Transactions on Automatic Control 52*, 2, 181–196.

FELDMAN, B. E., O'BRIEN, J. F., AND KLINGNER, B. M. 2005. Animating gases with hybrid meshes. In *Proc. SIGGRAPH '05*.

FELDMAN, B. E., O'BRIEN, J. F., KLINGNER, B. M., AND GOKTEKIN, T. G. 2005. Fluids in deforming meshes. In *2005 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 255–260.

FELLER, W. 1968. *An Introduction to Probability Theory and Its Applications*. Wiley.

FOGLEMAN, M., LUMLEY, J., REMPFER, D., AND HAWORTH, D. 2004. Application of the proper orthogonal decomposition to datasets of internal combustion engine flows. *Journal of Turbulence 5*, 23 (June).

FOSTER, N., AND METAXAS, D. 1996. Realistic animation of liquids. *Graphical Models and Image Processing 58*, 5.

GALLIVAN, K., GRIMME, E., AND DOOREN, P. V. 1996. A rational Lanczos algorithm for model reduction. *Numerical Algorithms 12*, 33–63.

GOLAS, A., NARAIN, R., SEWALL, J., KRAJCEVSKI, P., DUBEY, P., AND LIN, M. 2012. Large-scale fluid simulation using velocity-vorticity domain decomposition. *ACM Trans. Graph. 31*, 6 (Nov.), 148:1–148:9.

GOODNIGHT, N., WOOLLEY, C., LUEBKE, D., AND HUMPHREYS, G. A. 2003. Multigrid solver for boundary value problems using programmable graphics hardware. In *Proceeding of Graphics Hardware*.

GORAL, C. M., TORRANCE, K. E., GREENBERG, D. P., AND BATTAILE, B. 1984. Modeling the interaction of light between diffuse surfaces. *Computer Graphics 18*, 3 (July), 213–222.

GOSWAMI, P., SCHLEGEL, P., SOLENTHALER, B., AND PAJAROLA, R. 2010. Interactive sph simulation and rendering on the gpu. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, SCA '10, 55–64.

GRIMME, E. J. 1997. *Krylov Projection Methods For Model Reduction*. PhD thesis, Ohio State University.

GUGERCIN, S., AND ANTOULAS, A. 2004. A survey of model reduction by balanced truncation and some new results. *International Journal of Control 77*, 8, 748–766.

GUGERCIN, S., ANTOULAS, A., AND BEATTIE, C. A. 2006. A rational Krylov iteration for optimal H2 model reduction. In *Intl. Symposium on Mathematical Theory of Networks and Systems*.

GUPTA, M., AND NARASIMHAN, S. G. 2007. Legendre fluids: A unified framework for analytic reduced space modeling and rendering of participating media. In *2007 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 17–26.

HANRAHAN, P., SALZMAN, D., AND AUPPERLE, L. 1991. A rapid hierarchical radiosity algorithm. In *Proc of SIGGRAPH*.

HARMON, D., AND ZORIN, D. 2013. Subspace integration with local deformations. *ACM Trans. Graph. 32*, 4 (July).

HAUSER, K. K., SHEN, C., AND O'BRIEN, J. F. 2003. Interactive deformation using modal analysis with constraints. In *Graphics Interface*, CIPS, Canadian Human-Computer Commnication Society, 247–256.

HOLMES, P., LUMLEY, J. L., AND BERKOOZ, G. 1996. *Turbulence, Coherent Structures, Dynamical Systems and Symmetry*. Cambridge University Press.

HOSSAIN, M.-S., AND BENNER, P. 2008. Projection-based model reduction for time-varying descriptor systems using recycled Krylov subspaces. *Proceedings in Applied Mathematics and Mechanics 8*, 1, 10081–10084.

HOULETTE, R. 2003. Player modeling for adaptive games. In *AI Game Programming Wisdom 2*, S. Rabin, Ed. Charles River Media.

ID SOFTWARE, 1997. Quake 2.

JAKOB, W., 2010. Mitsuba renderer. http://www.mitsuba-renderer.org.

JAMES, D. L., AND FATAHALIAN, K. 2003. Precomputing interactive dynamic deformable scenes. In *Proc. SIGGRAPH '03*.

JAMES, D. L., AND FATAHALIAN, K. 2003. Precomputing interactive dynamic deformable scenes. Tech. Rep. CMU-RI-TR-03-33, Carnegie Mellon University Robotics Institute.

JAMES, D. L., AND PAI, D. K. 2002. DyRT: Dynamic response textures for real time deformation simulation with graphics hardware. *ACM Transactions on Graphics 21*, 3 (July), 582–585.

JAMES, D. L., BARBIČ, J., AND PAI, D. K. 2006. Precomputed acoustic transfer: output-sensitive, accurate sound generation for geometrically complex vibration sources. *ACM Transactions on Graphics 25*, 3 (July), 987–995.

KEISER, R., ADAMS, B., GASSER, D., BAZZI, P., DUTRE, P., AND GROSS, M. 2005. A unified Lagrangian approach to solid-fluid animation. In *Proceedings Symposium Point-Based Graphics*.

KELLER, A. 1997. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '97, 49–56.

KIM, T., AND DELANEY, J. 2013. Subspace fluid re-simulation. *ACM Transactions on Graphics 32*, 4.

KIM, T., AND JAMES, D. L. 2009. Skipping steps in deformable simulation with online model reduction. *ACM Transactions on Graphics 28*, 5 (Dec.), 123:1–123:9.

KIM, T., AND JAMES, D. L. 2011. Physics-based character skinning using multi-domain subspace deformations. In *Proc. SCA '11*.

KIM, D., KOH, W., NARAIN, R., FATAHALIAN, K., TREUILLE, A., AND O'BRIEN, J. F. 2013. Near-exhaustive precomputation of secondary cloth effects. *ACM Transactions on Graphics 32*, 4 (July), 87:1–7. Proceedings of ACM SIGGRAPH 2013, Anaheim.

KITTUR, A., CHI, E. H., AND SUH, B. 2008. Crowdsourcing user studies with Mechanical Turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, 453–456.

KLINGNER, B. M., AND SHEWCHUK, J. R. 2007. Agressive tetrahedral mesh improvement. In *Proceedings of the 16th International Meshing Roundtable*, 3–23.

KLINGNER, B. M., FELDMAN, B. E., CHENTANEZ, N., AND O'BRIEN, J. F. 2006. Fluid animation with dynamic meshes. *ACM Transactions on Graphics 25*, 3 (July), 820–825.

KRÜGER, J., AND WESTERMANN, R. 2003. Linear algebra operators for GPU implementation of numerical algorithms. In *Proc. SIGGRAPH '03*.

LEE, J., KLADWANG, W., LEE, M., CANTU, D., AZIZYAN, M., KIM, H., LIMPAECHER, A., YOON, S., TREUILLE, A., DAS, R., AND ETERNA PARTICIPANTS. 2014. RNA design rules from a massive open laboratory. *Proceedings of the National Academy of Sciences*. 2014. (Preprint).

LEGRESLEY, P. A., AND ALONSO, J. J. 2003. Dynamic domain decomposition and error correction for reduced order models. *41st AIAA Aerospace Sciences Meeting and Exhibit*.

LI, W., WEI, X., AND KAUFMAN, A. 2003. Implementing lattice Boltzmann computation on graphics hardware. *The Visual Computer 19*, 7-8.

LI, J.-R. 2000. *Model Reduction of Large Linear Systems*. PhD thesis, Massachusetts Institute of Techology.

LIMPAECHER, A., FELTMAN, N., TREUILLE, A., AND COHEN, M. 2013. Real-time drawing assistance through crowdsourcing. *ACM Trans. Graph. 32*, 4 (July), 54:1–54:8.

LIUA, C., YUAN, X., MULLAGURU, A., AND FAN, J. 2008. Model order reduction via rational transfer function fitting and eigenmode analysis. In *International Conference on Modeling, Identification and Control*.

LOOS, B. J., ANTANI, L., MITCHELL, K., NOWROUZEZAHRAI, D., JAROSZ, W., AND SLOAN, P.-P. 2011. Modular radiance transfer. *ACM Transactions on Graphics 30*, 6 (Dec.).

LOOS, B. J., NOWROUZEZAHRAI, D., JAROSZ, W., AND SLOAN, P.-P. 2012. Delta radiance transfer. In *Proceedings of the 2012 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D 2012.

LOSASSO, F., GIBOU, F., AND FEDKIW, R. 2004. Simulating water and smoke with an octree data structure. In *Proc. SIGGRAPH '04*.

LUCIA, D. J., AND KING, P. I. 2002. Domain decomposition for reduced-order modeling of a flow with moving shocks. *AIAA Journal 40*, 11, 2360–2362.

LUMLEY, J. L. 1970. *Stochastic Tools in Turbulence*, vol. 12 of *Applied Mathematics and Mechanics*. Academic Press.

MACKLIN, M., AND MÜLLER, M. 2013. Position based fluids. *ACM Trans. Graph. 32*, 4 (July), 104:1–104:12.

MARION, M., AND TEMAM, R. 1989. Nonlinear Galerkin methods. *SIAM J. Numer. Anal. 26*, 5, 1139–1157.

MCCANN, J., AND POLLARD, N. 2007. Responsive characters from motion fragments. *ACM Trans. Graph. 26*, 3 (July).

MICROSOFT, 2013. Drivatar website. http://research.microsoft.com/en-us/projects/drivatar.

MULLEN, P., CRANE, K., PAVLOV, D., TONG, Y., AND DESBRUN, M. 2009. Energy-preserving integrators for fluid animation. *ACM Transactions on Graphics 28*, 3 (July), 38:1–38:8.

MÜLLER, M., CHARYPAR, D., AND GROSS, M. 2003. Particle-based fluid simulation for interactive applications. In *Proc. SCA '03*.

OLSSEN, K. H. A. 2005. *Model Order Reduction with Rational Krylov Methods*. PhD thesis, KTH Stockholm.

PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. 1999. The PageRank citation ranking: bringing order to the Web. Technical Report 1999-66, Stanford InfoLab, November.

PARK, S. I., AND KIM, M. J. 2005. Vortex fluid for gaseous phenomena. In *Proc. SCA '05*.

PAVLOV, D., MULLEN, P., TONG, Y., KANSO, E., MARSDEN, J., AND DESBRUN, M. 2011. Structure-preserving discretization of incompressible fluids. *Physica D: Nonlinear Phenomena 240*, 6, 443 – 458.

PENTLAND, A., AND WILLIAMS, J. 1989. Good vibrations: Modal dynamics for graphics and animation. *Computer Graphics (SIGGRAPH 89) 23*, 3 (July), 215–222. Held in Boston, Massachusetts.

PFAFF, T., THUEREY, N., AND GROSS, M. 2012. Lagrangian vortex sheets for animating fluids. *ACM Trans. Graph. 31*, 4 (July), 112:1–112:8.

PHILLIPS, J. R. 1998. Model reduction of time-varying linear systems using approximate multipoint Krylov-subspace projectors. *Computer Aided Design*, 96–102.

PIXAR, 1997. Geri's Game.

RAVEENENDRAN, K., WOJTAN, C., THUEREY, N., AND TURK, G. 2014. Blending liquids. *ACM Trans. Graph. 33*, 4 (July).

REN, Z., WANG, R., SNYDER, J., ZHOU, K., LIU, X., SUN, B., SLOAN, P.-P., BAO, H., PENG, Q., AND GUO, B. 2006. Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation. *ACM Transactions on Graphics 26*, 4.

ROWLEY, C. W., AND MARSDEN, J. E. 2000. Reconstruction equations and the Karhunen-Loéve expansion for systems with symmetry. *Phys. D 142*, 1-2, 1–19.

ROWLEY, C. W., KEVREKIDIS, I. G., MARSDEN, J. E., AND LUST, K. 2003. Reduction and reconstruction for self-similar dynamical systems. *Nonlinearity 16* (July), 1257–1275.

ROWLEY, C., WILLIAMS, D., COLONIUS, T., MURRAY, R., AND MACMARTIN, D. 2006. Linear models for control of cavity flow oscillations. *J. Fluid Mech. 547*, 317–330.

SANDBERG, H., AND RANTZER, A. 2004. Balanced truncation of linear time-varying systems. *IEEE Transactions on Automatic Control 49*, 2, 217–229.

SAVAS, B., AND ELDÉN, L. 2009. Krylov subspace methods for tensor computations. Tech. Rep. LITH-MAT-R-2009-02-SE, Department of Mathematics, Linköpings Universitet.

SCHMIT, R., AND GLASUER, M. 2002. Low dimensional tools for flow-structure interaction problems: Application to micro air vehicles. *APS Meeting Abstracts* (Nov.), D1+.

SCHÖDL, A., SZELISKI, R., SALESIN, D. H., AND ESSA, I. 2000. Video textures. In *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, 489–498.

SELLE, A., RASMUSSEN, N., AND FEDKIW, R. 2005. A vortex particle method for smoke, water and explosions. In *Proc. SIGGRAPH '05*.

SEWALL, J., MECKLENBURG, P., MITRAN, S., AND LIN, M. 2007. Fast fluid simulation using residual distribution schemes. In *Proc. Eurographics Workshop on Natural Phenomena*.

SHEWCHUK, J. R. 1994. An introduction to the conjugate gradient method without the agonizing pain. Tech. Rep. CS-94-125, Carnegie Mellon University, Pittsburgh, PA, USA.

SI, H. 2007. A quality tetrahedral mesh generator and 3-dimensional Delaunay triangulator. *http://tetgen.berlios.de/*.

SIRISUP, S., KARNIADAKIS, G. E., XIU, D., AND KEVREKIDIS, I. G. 2005. Equation-free/Galerkin-free POD-assisted computation of incompressible flows. *J. Comput. Phys. 207*, 2, 568–587.

SIROVICH, L. 1987. Turbulence and the dynamics of coherent structures. *Quarterly of Applied Mathematics 45*, 561–590.

SLOAN, P.-P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *Proc. SIGGRAPH '02*.

SLOAN, P.-P., LUNA, B., AND SNYDER, J. 2005. Local, deformable precomputed radiance transfer. *ACM Transactions on Graphics 24*, 3 (Aug.), 1216–1224.

SMITH, A. M., LEWIS, C., HULLETT, K., SMITH, G., AND SULLIVAN, A. 2011. An inclusive taxonomy of player modeling. Tech. Rep. UCSC-SOE-11-13, University of California, Santa Cruz.

SOLENTHALER, B., AND PAJAROLA, R. 2009. Predictive-corrective incompressible SPH. *ACM Trans. Graph. 28*, 3 (July), 40:1–40:6.

SORKINE, O. 2006. Differential representations for mesh processing. *Computer Graphics Forum 25*, 4, 789–807.

STAM, J. 1999. Stable fluids. In *Computer Graphics (SIGGRAPH 99)*.

ŠT'AVA, O., BENEŠ, B., BRISBIN, M., AND KŘIVÁNEK, J. 2008. Interactive terrain modeling using hydraulic erosion. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08, 201–210.

TEZAUR, R., AND FARHAT, C. 2006. Three-dimensional discontinuous Galerkin elements with plane waves and Lagrange multipliers for the solution of mid-frequency Helmholtz problems. *Int. J. Numer. Meth. Engng 66*.

TEZAUR, R., ZHANG, L., AND FARHAT, C. 2008. A discontinuous enrichment method for capturing evanescent waves in multiscale fluid and fluid/solid problems. *Comput. Methods Appl. Mech. Engrg. 197*.

THUE, D., BULITKO, V., SPETCH, M., AND WASYLISHEN, E. 2007. Interactive storytelling: A player modelling approach. In *The Third Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE '07.

THUREY, N., MÜLLER-FISCHER, M., SCHIRM, S., AND GROSS, M. 2007. Real-time breaking waves for shallow water simulations. In *Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, PG '07, 39–46.

TOSELLI, A., AND WIDLUND, O. 2005. *Domain Decomposition Methods - Algorithms and Theory*. Springer.

TREUILLE, A., LEWIS, A., AND POPOVIĆ, Z. 2006. Model reduction for real-time fluids. In *Proc. SIGGRAPH '06*.

VON AHN, L., AND DABBISH, L. 2004. Labeling images with a computer game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, 319–326.

VON AHN, L., LIU, R., AND BLUM, M. 2006. Peekaboom: a game for locating objects in images. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, 55–64.

VON AHN, L., MAURER, B., MCMILLEN, C., ABRAHAM, D., AND BLUM, M. 2008. reCAPTCHA: Human-based character recognition via web security measures. *Science 321*, 5895 (August), 1465–1468.

WU, E., LIU, Y., AND LIU, X. 2005. An improved study of real-time fluid simulation on the GPU. *Computer Animation and Virtual Worlds 15*, 3-4.

ZHANG, L., TEZAUR, R., AND FARHAT, C. 2006. The discontinuous enrichment method for elastic wave propagation in the medium-frequency regime. *Internat. J. Numer. Methods Engrg. 66*.

ZHAO, Y., AND BARBIČ, J. 2013. Interactive authoring of simulation-ready plants. *ACM Trans. on Graphics (SIGGRAPH 2013) 32*, 4, 84:1–84:12.

ZHOU, Y. 2002. *Numerical Methods for Large Scale Matrix Equations with Applications in LTI System Model Reduction*. PhD thesis, Rice University.

ZHU, Y., AND BRIDSON, R. 2005. Animating sand as a fluid. *ACM Trans. Graph. 24*, 3 (July), 965–972.

ZHUKOV, S., IONES, A., AND KRONIN, G. 1998. An ambient light illumination model. In *Rendering Techniques*, 45–56.

ZOOK, A., LEE-URBAN, S., DRINKWATER, M. R., AND RIEDL, M. O. 2012. Skill-based mission generation: A data-driven temporal player modeling approach. In *Proceedings of the 7th International Conference on the Foundations of Digital Games*, FDG '12.

ZOOK, A., FRUCHTER, E., AND RIEDL, M. O. 2014. Automatic playtesting for game parameter tuning via active learning. In *Proceedings of the 9th International Conference on the Foundations of Digital Games*, FDG '14.