Learning to splash
CS 229 Course Project

Wen Zheng

1 Introduction

Physics-based methods have been successfully applied to the production of realistic liquids simulation. Nowadays, a giant tide of floods sweeping through the New York city is no longer a surprise in movies. However, expensive computation prohibits applying the same visual effects to games, medical training, and other real-time applications.

One way to achieve the real-time simulation is to simplify the model, so that computation cost can be reduced to an acceptable amount. However, among infinite number of possible simplification methods, how to choose the one that preserves the most visual credibility is a difficult task. Another way to reduce the computation cost while preserving visual quality is to use more complicated geometry primitives to represent liquids. For example, particle-based methods represent liquids as clusters of spherical particles, which are very simple primitives. This simplicity benefits human who can easily develop simulation methods based on those primitives, but it sacrifices efficiency of representation, and thus requires much more sampling and computation cost.

This project aims at a real-time but visually plausible simulation system by applying machine learning methods learned from [Ng 2009]. To achieve this goal, the planned work is divided into three parts: first, *data generation*, which is to implement a full fluids simulator to generate training data; second, *geometry primitives learning*, which is to use an unsupervised learning method to derive a set of representative primitives from the full simulation results; third, *dynamics rule learning*, which is to use a supervised learning method to develop the updating rules of the representative primitives attained previously from the full simulation results.

2 Simulation

The first part of the project implements a Navier-Stokes fluid simulator. The Navior-Stokes equation for incompressible viscous fluid is as follows:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nabla \cdot (\nu \nabla \mathbf{u}) - \frac{\nabla \mathbf{p}}{\rho} + \mathbf{f}$$

$$\nabla \cdot \mathbf{u} = \mathbf{0}$$

where $\mathbf{u}$ is the velocity, p is the pressure, ν is the viscosity, ρ is the density, and $\mathbf{f}$ is the external force, such as gravity, buoyancy, surface tension, and other user-defined forces. For the application of this project, the viscosity term is ignored for simplicity, because the viscosity smears out interesting dynamics which is needed for training.

The Navier-Stokes equations describe the updating rules for the velocity field $\mathbf{u}$ and the pressure field p. In addition to $\mathbf{u}$ and p, we also need to represent and evolve the interfaces of fluids. A typical fluid simulator usually uses the signed distance function ϕ to represent the distance to the nearest interface and the region: $\phi > 0$ if outside the fluid region, $\phi < 0$ if inside the fluid region, $\phi = 0$ if on the interface. Then we can evolve ϕ through the level set equation:

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = \mathbf{0}$$

where the velocity field $\mathbf{u}$ that drives the ϕ around is updated by the Navier-Stokes equations, whereas the updated ϕ is then used to distinguish the fluid region and the non-fluid region for solving the Navier-Stokes equations.

The numerical updating rule is divided into the following steps:

1. Evolve the interface ϕ by solving the level set equation.

   The BFECC method [Dupont and Liu 2003] is adopted to solve the advection, and the fast marching method [Sethian 1999] is then used to reinitialize ϕ.

2. Evolve the velocity field $\mathbf{u}$ by solving the Navier-Stokes equation.

   (a) Solve the advection term by the BFECC method [Dupont and Liu 2003] and the external force term by the forward Euler method to attain an intermediate velocity $\mathbf{u}^*$:

   $$\frac{\mathbf{u}^* - \mathbf{u}^{\mathbf{old}}}{\partial t} = -(\mathbf{u}\cdot)\mathbf{u} + \mathbf{f}$$

   (b) Project $\mathbf{u}^*$ onto an divergence-free field. First, use a Preconditioned Conjugate Gradient method to compute the pressure p by solving the follwing Poisson equation [Foster and Fedkiw 2001]:

   $$\nabla^2 p = \frac{\rho}{\partial t}\nabla \cdot \mathbf{u}^*$$

   Then use the pressure p to update the velocity:

   $$\frac{\mathbf{u}^{\mathbf{new}} - \mathbf{u}^*}{\partial t} = -\frac{\nabla p}{\rho}.$$

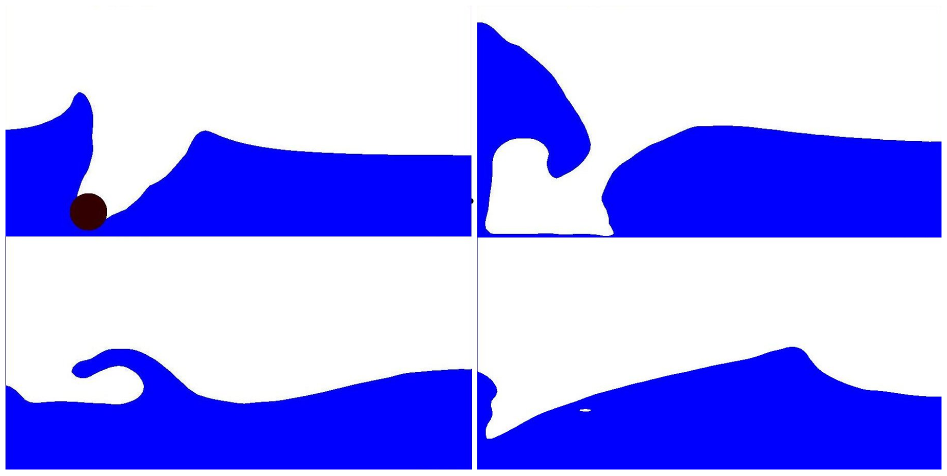

Figure 1: Snapshots of 2D full simulation of splashing water for training.

3 Data Generation

Since fluids are arbitrarily morphable, theoretically they can have any geometric shapes. However, the deformation of fluids obays particular dynamics rules, and thus the probability of shapes that fluids will form is not uniform. Furthermore, given a specific class of scenarios that we concern the most, certain kinds of shapes will be more possible to appear than others. For instance, in a game where the most fequent interaction with water is that characters step into water or objects smash into a pool, the most possible shapes of water are splashes and breaking waves. This implies that we can extract some representative geometric features from simulation data. It also implies that the training data are most useful when they are generated in similar situations.

Thus, in this project, the following scenario is considered for the source of training data: a rigid kinematic object (sphere for simpicify) smashing through a pool of water. To cover a sufficiently large range of possible shapes of splashes, a number of different simulations are run with randomly generated parameters, including the moving direction, the velocity, the size of the object, etc. (See Fig. 1). The result data were stored as matrices of the signed distance function $\mathbf{\Phi}_{i,j}$, where i is the index of simulations, and j is the index of time steps.

4 Learning the Geometry primitives

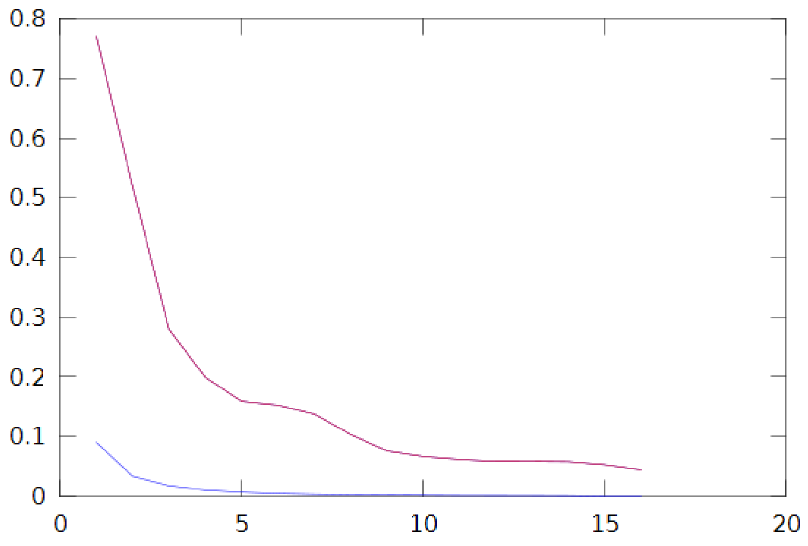

Figure 2: Errors (Y axis) of representation using different numbers (X axis) of principal components. The red curve is the max errors, and the blue curve is the mean errors.

The next step is to attain a compact representation of the water geometry, so that we can reduce the number of simulated elements and speed up the simulation. To achieve this goal, the training data $\mathbf{\Phi}_{i,j}$ is first decomposed into column vectors $\phi_{i,j,k}$, where $\mathbf{\Phi}_{i,j} = [\phi_{i,j,k}]_k$.

Note that other ways of decomposition have also been tested, such as decomposing the domain into rectangular blocks. However, they suffers from noises and discontinuities near decomposition boundaries. Among all the decomposition methods tested, decomposing the domain into columns is the only one whose compressed geometry has acceptable visual effects, and thus it is selected as the final method of decomposition.

The Principal Component Analysis method is used to compress $\phi_{i,j,k}$ into a limited set of representative column vectors $\hat{\mathbf{\Phi}}$. Then each column vector $\phi_{i,j,k}$ can be expressed as a small number of real values attained from the following equation

$$\alpha_{i,j,k} = \hat{\mathbf{\Phi}}^T(\phi_{i,j,k} - \mu),$$

where $\alpha_{i,j,k}$ is a lower dimensional vector, and μ is the mean value of $\phi_{i,j,k}$. Also the column vectors can be

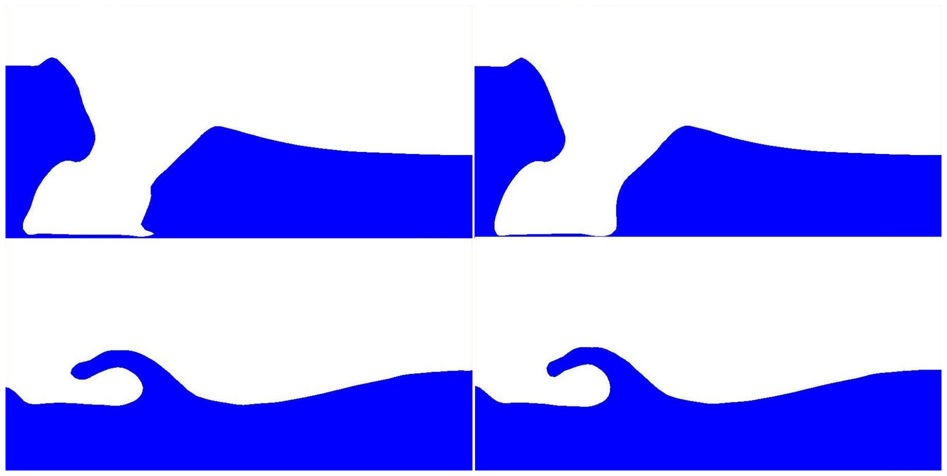
Figure 3: Comparison between original geometry (left) and compressed geometry (right) with 8 principal components.

recovered as follows

$$\phi_{i,j,k} = \hat{\mathbf{\Phi}}\alpha_{i,j,k} + \mu.$$

As we can see in Fig. 2, the errors of the compressed geometry representation decrease fast as the number of principal components increases. geometry. The first 8 principal components are selected as the subspace basis for compression. Fig. 3 shows the comparison of two snapshots from the original data and the compressed data. We can see that the compression preserves most of geometric features except high-curvature details.

5 Learning the Dynamics rules

The next step is to learn the dynamics updating rule of the geometry primitives attained previously. Based on the intuition from the Navier-Stokes equation, the updating rule should be spatially and temporally local, spatially symmetric. And since the final system solved in the full simulation model is a linear system, we can also assume that the updating rule is in the form of a linear system.

Based on the above assumption, a hypothesis model is attained as follows

$$\alpha_i^n = \sum_{j=1}^{j_{max}} \sum_{k=0}^{k_{max}} \left(\mathbf{A}_k^j (\alpha_{i+k}^{n-j} + \alpha_{i-k}^{n-j}) \right),$$

where α_i^n is the compressed geometry representation vector at time n at column i, and $\mathbf{A}$s are parameter matrices to be trained for. The above equation can be expressed in the form of $\vec{\alpha} = \hat{\mathbf{A}}\hat{\vec{\alpha}}$, and thus can be solved by the normal equation

$$\hat{\mathbf{A}} = (\vec{\alpha}\vec{\alpha}^T)^+ \hat{\vec{\alpha}}\vec{\alpha}^T$$

where "+" is the pseudo-inverse operator.

Boundary conditions are also an important aspect for simulation. In lack of physical meaning of α, two commonly used boundary conditions are assumed to be applicable: The reflective boundary condition simply copy the mirror image of neighbors of the boundary node to the exterior nodes, and the inverse boundary condition is the negative of the reflective boundary condition.

To select parameters j_{max} and k_{max} that work best, a hold-out cross validation has been executed. The training set is formed by 70% of randomly chosen data, and the test set is the other 30%. Since the training is on a large data set and thus expensive, only 3 different j_{max} and 3 different k_{max} are tested. The training

Table 1: Mean errors for the reflective boundary condition.

	$k_{max} = 2$	$k_{max} = 3$	$k_{max} = 4$
$j_{max} = 1$	0.043071	0.00744	0.00706
$j_{max} = 2$	0.042996	0.00753	0.00714
$j_{max} = 3$	0.042939	0.00760	0.00719

Table 2: Mean errors for the inverse boundary condition.

	$k_{max} = 2$	$k_{max} = 3$	$k_{max} = 4$
$j_{max} = 1$	0.043068	0.00723	0.00695
$j_{max} = 2$	0.043002	0.00749	0.00715
$j_{max} = 3$	0.042942	0.00762	0.00721

errors and test errors in Table 1 and 2 show that larger j_{max} and k_{max} have both smaller training errors and test errors, which implies that a global model will be a better choice. However, a global model requires to train a huge parameter matrix which is infeasible. A possible solution is to update α in an *implicit* way, which means the global information is incorporated into the updating by spreading local information through a sparse linear system. We can also observe that errors of both boundary conditions have little difference with each other, which may indicates that both of them are incorrect. The search for the correct boundary condtion will be desirable in the future work.

6 Conclusion

Two steps of machine learning are designed and tested to speed up the water simulation. The first step is to use unsupervised learning techniques to attain a compressed geometry representation of water. The simulation data are first decomposed into columns and then compressed by the PCA method. The compressed result is visually pleasing and has a considerable compression ratio. The second step tried to attain a dynamics updating rule based on the compressed representation attained from the previous step. The resulting system fails to generate long-term simulation results with acceptable accuracy. However, the result of model selection implies that a global model and a new boundary condition can help to reduce the error. Solutions to these problems need to be explored in the future work.

References

[Dupont and Liu 2003] DUPONT, T., AND LIU, Y. 2003. Back and forth error compensation and correction methods for removing errors induced by uneven gradients of the level set function. *J. Comput. Phys. 190/1*, 311–324.

[Foster and Fedkiw 2001] FOSTER, N., AND FEDKIW, R. 2001. Practical animation of liquids. In *Proc. of ACM SIGGRAPH 2001*, 23–30.

[Ng 2009] NG, A. 2009. *CS229 Course Notes.*

[Sethian 1999] SETHIAN, J. 1999. Fast marching methods. *SIAM Review 41*, 199–235.