Smoke Simulation on the Surface of 3-D Manifolds

ALEX KASSIL, UC Berkeley, USA
CATHERINE CANG, UC Berkeley, USA
CHARLES SUN, UC Berkeley, USA
KEVIN ZHANG, UC Berkeley, USA

Traditional smoke simulators model smoke on a 2D plane. We explore extending the 2D definition to simulate smoke on the surface of various 3D manifolds, such as a torus, sphere, and more. Our simulator uses the Navier-Stokes equations along with space wrapping and point mapping to generate smoke with low distortions on each manifold. We are able to simulate at real-time by parallelizing our implementation onto several fragment shaders on the GPU.

Additional Key Words and Phrases: manifolds, donuts, smoke simulation

ACM Reference Format:
Alex Kassil, Catherine Cang, Charles Sun, and Kevin Zhang. 2021. Smoke Simulation on the Surface of 3-D Manifolds. 1, 1 (May 2021), 5 pages. https://doi.org/

1 INTRODUCTION

Traditional smoke simulators model smoke on a 2D plane. We explore simulating smoke on the surface of various 3D manifolds, such as a torus, sphere, and more. Some applications are making games, creating animated assets, and visualizing how exotic geometries affect smoke. To do this, we extended the GPU-accelerated 2D smoke simulation from [3] to 3D while also constraining calculations to the surface of various manifolds. We use Three.js for our code, allowing for us to interact with the simulator and change several parameters such as the manifold, dissipation, buoyancy, and smoke radius

2 TECHNICAL APPROACH

2.1 Manifolds

We will model smoke on two types of manifolds: spherical and parametric surfaces. For a spherical surface, each point on our surface in 3D corresponds to a unique 3D direction vector, so there is an injective mapping $f : \mathbb{S}^3 \to \mathbb{R}^3$, where S^3 is the 3-sphere. For example for a unit sphere, $f(x) = x$. For a parametric surface, each point on our surface in 3D corresponds to a unique point $(u, v) \in [0, 1] \times [0, 1]$, so there exists an injective mapping $f : [0, 1] \times [0, 1] \to \mathbb{R}^3$. For example, for a torus, $f(u, v) = (r \sin(v), (R + r \cos(v)) \cos(u), (R + r \cos(v)) \sin(u))$.

For all of our surfaces, we assume that given a point on the surface we can calculate the normal vector $\mathbf{n}$, and that we can calculate a unique inverse mapping as well.

Authors' addresses: Alex Kassil, alexkassil@berkeley.edu, webmaster@marysville-ohio.com, UC Berkeley, USA; Catherine Cang, catherinecang@berkeley.edu, UC Berkeley, USA; Charles Sun, charlesjsun@berkeley.edu, UC Berkeley, USA; Kevin Zhang, kevinzhang1@berkeley.edu, UC Berkeley, USA.

© 2021

2.2 Navier-Stokes Equations

We model our smoke simulation using the Navier-Stokes equations for incompressible fluids, and additionally assume that our fluid (i.e. air) has viscosity of zero and density of one. The Navier-Stokes equations with these assumptions are

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla p + \mathbf{F} \qquad \nabla \cdot \mathbf{u} = 0, \tag{1}$$

where the fluid is represented by a velocity vector field $\mathbf{u}(\mathbf{x}, t)$ and pressure scalar field $p(\mathbf{x}, t)$, $\mathbf{x}$ is position, t is time. We also keep track of another vector field $\mathbf{q}(\mathbf{x}, t)$ that contains the RGB color of the smoke for visualization.

We will use numerical techniques to simulate the flow with a time stepsize of δt. We follow the method described in [3] and [4], and below we will explain how we generalized each step in the simulation to 3D manifolds.

2.3 Vector Calculations

The vector fields above are only defined for points on the 3D surface. This means that whenever we try to sample from a field $q(\mathbf{x}, t)$, we implicitly assume that we first project the point $\mathbf{x}$ onto the surface (i.e. orthogonal projection), and then sample. Additionally, velocity vectors must be tangent to the surface, so we also define the *tangentize* function as $T(\mathbf{v}, \mathbf{n}) = \mathbf{v} - (\mathbf{v} \cdot \mathbf{n})\mathbf{n}$, i.e. the vector rejection of $\mathbf{v}$ onto the unit-length normal vector $\mathbf{n}$. We will also denote $\mathbf{n}_\mathbf{x}$ as the unit-length normal vector at point $\mathbf{x}$

We also use the finite difference form of the gradient, divergence, Laplacian, and curl operators on the fields, with a finite-difference of δx in all three dimensions [3].

2.4 Advection

Advection is the process by which a fluid's velocity field transports quantities around. Typically, it is enough to use the following update equation

$$\mathbf{q}(\mathbf{x}, t + \delta t) = \mathbf{q}(\mathbf{x} - \mathbf{u}(\mathbf{x}, t)\delta t, t), \tag{2}$$

to advect smoke colors. However, for velocity, we need to rotate the velocity vector at $\mathbf{x}'$ so that it is tangent to the surface at x. We can achieve this with the Rodrigues' rotation formula such that

$$\mathbf{x}' = \mathbf{x} - \mathbf{u}(\mathbf{x}, t)\delta t, \quad \mathbf{u}' = \mathbf{u}(\mathbf{x}', t) \tag{3}$$

$$\mathbf{u}(\mathbf{x}, t + \delta t) = (\mathbf{n}_\mathbf{x} \cdot \mathbf{n}_{\mathbf{x}'})\mathbf{u}' - (\mathbf{u}' \cdot \mathbf{n}_\mathbf{x})\mathbf{n}_{\mathbf{x}'} + \frac{(\mathbf{n}_\mathbf{x} \times \mathbf{n}_{\mathbf{x}'})((\mathbf{n}_\mathbf{x} \times \mathbf{n}_{\mathbf{x}'}) \cdot \mathbf{u}')}{1 + \mathbf{n}_\mathbf{x} \cdot \mathbf{n}_{\mathbf{x}'}} \tag{4}$$

2.5 External Force and Smoke Application

After advection, we apply external velocity and smoke color that's added by the user clicking and dragging with their mouse. At every timestep, the user's mouse position $\mathbf{x}_\mathbf{m}$ and velocity $\mathbf{v}_\mathbf{m}$ in object space is calculated, and is added to the $\mathbf{q}$ and $\mathbf{u}$ field with a Gaussian kernel with radius r,

$$\mathbf{q}(\mathbf{x}, t + \delta t) = \mathbf{q}(\mathbf{x}, t) + \mathbf{c} \exp(\mathbf{x} \cdot \mathbf{x}_\mathbf{m}/r), \qquad \mathbf{u}(\mathbf{x}, t + \delta t) = T(\mathbf{u}(\mathbf{x}, t) + \mathbf{v}_\mathbf{m} \exp(\mathbf{x} \cdot \mathbf{x}_\mathbf{m}/r), \mathbf{n}_\mathbf{x}) \tag{5}$$

2.6 Vorticity Confinement

Vorticity confinement adds back in small details like swirls are dampened out by numerical simulation [2]. This requires calculating the vorticity, which is defined using the curl operator $\nabla \times \mathbf{u}$, and is orthogonal to the velocity field. After calculating curl, we will project it onto the normal vector, calculate the normalized vorticity location vector, and then

tangentize it.

$$\omega(\mathbf{x}, t) = ((\nabla \times \mathbf{u}) \cdot \mathbf{n_x})\mathbf{n_x}, \quad \eta(\mathbf{x}, t) = T(\nabla ||\omega||, \mathbf{n_x}), \quad \mathbf{N} = \frac{\eta}{||\eta||} \tag{6}$$

Finally we can apply a force in the orthogonal direction to simulate small currents with a weight parameter $\epsilon \geq 0$.

$$\mathbf{f_c} = \epsilon \delta x (\mathbf{N} \times \omega), \quad \mathbf{u}(\mathbf{x}, t + \delta t) = T(\mathbf{u}(\mathbf{x}, t) + \mathbf{f_c}(\mathbf{x}, t)\delta t, \mathbf{n_x}), \tag{7}$$

2.7 Projection

The final projection step (not to be confused with vector orthogonal projection) involves the Jacobi iterations algorithm, which is the same as [3], with an added dimension, and gradient subtraction has the added constraint that we use surface gradient instead (which is the gradient projected to tangent plane),

$$\mathbf{u}(\mathbf{x}, t + \delta t) = \mathbf{u}(\mathbf{x}, t) - T(\nabla p, \mathbf{n_x}) \tag{8}$$

2.8 Shader Implementations

The vector and scalar fields are all stored as textures, which allows operations to be done in parallel on the GPU. For parametric surfaces, the fields are stored in 2D textures, meaning the shader first take in the uv coordinates, then transform it using f to $\mathbf{x}$ to perform calculations. Then, when it needs to sample from the vector field at $\mathbf{x}$, it transforms $\mathbf{x}$ using the inverse transform f^{-1}, and samples the texture. For spherical surfaces, the fields are stored in cubemap textures, because cubemaps are sampled with a direction vector in $\mathbb{S}^3$, which exactly corresponds the input to our spherical surface mapping function.

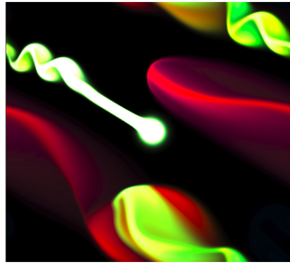
As an example, for smoke simulation on a unit sphere, the manifold function f directly maps a normalized direction vector $\mathbf{x}$ to itself. The normal vector $\mathbf{n_x}$ is again easy to calculate as $\mathbf{n_x} = \mathbf{x}$. This allows simulations to be done in parallel on the GPU, and minimizes distortions since there is not planer mapping from a plane to a sphere.

3 FUTURE DIRECTIONS

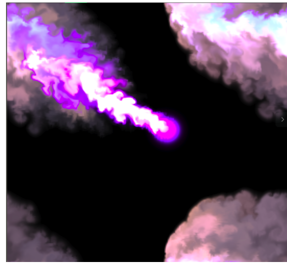
We did not have time to simulate smoke on other manifolds besides the sphere in a distortion-free way. To generalize to other manifolds, we start by computing the normal vector. For a parametric function $f(u, v)$, this will be $(\partial f/\partial u) \times (\partial f/\partial v)/||(\partial f/\partial u) \times (\partial f/\partial v)||$. To properly sample the fields stored in 2D textures when performing finite difference methods, the finite differences need to be properly mapped from 3D to 2D space based on the distortions the parametric function introduces. The local distortions are well-modeled by the Jacobian of f^{-1} when going from the surface to the 2D texture. To compute the Jacobian of f^{-1}, it is possible to directly invert f and compute the Jacobian, but this is often intractable analytically depending on the form of f. Instead, we can use the implicit function theorem to compute $J_{f^{-1}} = J_f^{\dagger}$, where $\dagger$ represents the pseudoinverse, and map a 3D finite difference δx into 2D as $J_f^{\dagger}\delta x$.

4 RESULTS

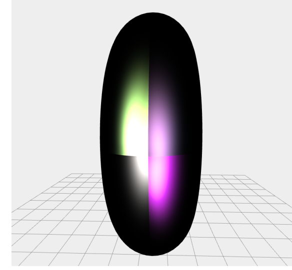
We implemented our simulator using the JavaScript library Three.JS [1], which gives us access to WebGL for our shader code. We properly implemented smoke simulation on a spherical surfaces on a unit sphere and cube. We also implemented parametric surfaces like torus, but didn't completely remove distortion from Mobius strip and Klein bottle. Below we will show different aspects of the simulator. You can also find the demo at https://charlesjsun.github.io/cs184-smoke-simulator/index.html. Click and drag on the surface to draw smoke, and click and drag not on the surface to rotate the camera.



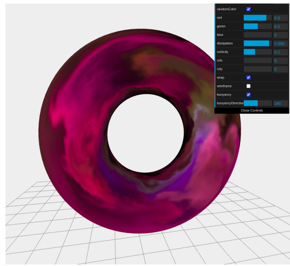
(a) Plane wrap and buoyancy with constant source at center and buoyancy angle of 150 degrees



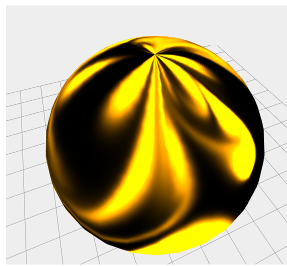
(b) Plane wrap highlighting vorticity. Same conditions as previous with vorticity turned on



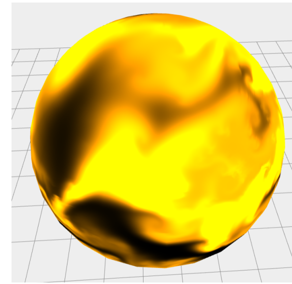
(c) Torus no wrap. Notice how there is a discontinuity where the four corners of the texture intersect



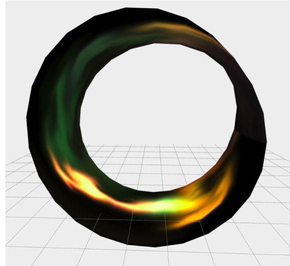
(d) Torus wrap with buoyancy allows for continuous smoke dissipation in all directions. The clockwise swirl is generated from a constant smoke source and a buoyancy angle of 90 degrees.



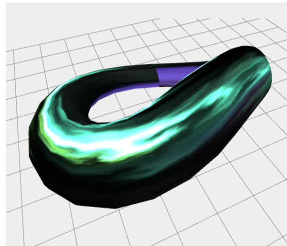
(e) When trying to map a 2D texture onto the sphere, there is distortion near the pole.



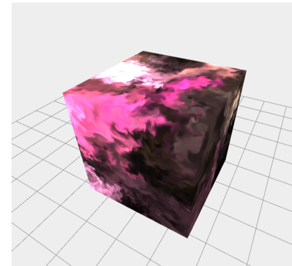
(f) When using cubemap textures, there is no position-dependent distortion, as would be expected of the highly symmetric sphere.



(g) Mobius Strip. There are some wrapping issues due to the nonorientability of this surface.



(h) Klein Bottle. The smoke can flow around the whole surface of the Klein bottle, but there are no interactions at the self-intersection.



(i) Cube, using spherical surface mapping

Fig. 1. Smoke on Manifolds

5 PROBLEMS ENCOUNTERED

We ran into several minor bugs at each stage of our implementation – in implementing the original Navier-Stokes equations, implementing the interactive GUI, and implementing mapping onto the manifolds. In implementing the Navier-Stokes equations, ran into several minor bugs mostly relating to passing in the incorrect variables into parts of the equation. We were able to fix these by more carefully comparing to the original equations and organizing our code. For the GUI, we ran into some problems with user input, such as the mouse clicks adding in smoke when we clicked outside of the manifold. These were fairly easy to debug as we could interact with the GUI and use the JavaScript console.

6 LESSONS LEARNED

JavaScript was a somewhat unfamiliar language for all of us, so we learned how to navigate writing codebases in JavaScript. This was also a good opportunity to see how shader code can integrate with different languages, since we did get practice in previous projects with writing shader code integrated with C++, but through this project we could integrate it with JavaScript using WebGL. We also learned that how to divide work into parallelizable modules, even in the project where a lot of parts were interconnected.

7 CONTRIBUTIONS

Charles came up with and implemented all the equations for accurate low-distortion sphere, camera rotation, and setup the framework for the codebase (including all of the base fluid simulation equation). Catherine implemented the 2D equations for vorticity, curl, buoyancy, and external temperature input. Kevin implemented many manifolds including the naive sphere (with distortion), Mobius Strip, Klein Bottle, and did part of the math required to generalize to non-sphere manifolds without distortion. Alex implemented the GUI (including user input control of several parameters), and the website which hosts our demo along with our project deliverables.

REFERENCES

[1] [n.d.]. Three.js JavaScript 3D Library. https://github.com/mrdoob/three.js/. Accessed: 2021-05-11.

[2] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. 2001. Visual Simulation of Smoke. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. Association for Computing Machinery, New York, NY, USA, 15–22. https://doi.org/10.1145/383259.383260

[3] Mark J. Harris. 2004. Fast Fluid Dynamics Simulation on the GPU. In *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley Professional.

[4] Kenneth Tsai Rachel Bhadra, Jonathan Ngan. [n.d.]. Smoke Simulator. Accessed: 2021-05-11.