Rhythm & Flow

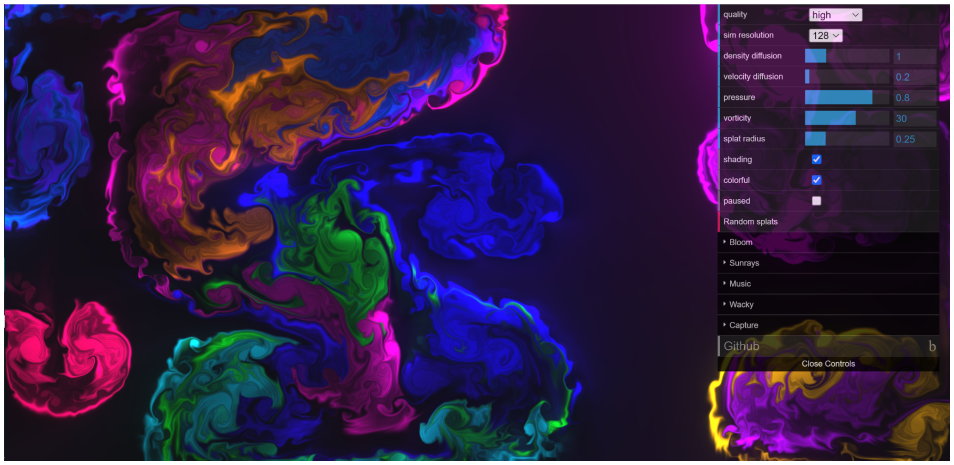
Clayton Ferguson '23, Toussaint Webb '22, Enmandra Wright '22

Abstract

Rhythm & Flow is a fluid dynamics simulator that changes depending on musical attributes (i.e., BPM) and renders "wacky" fluid movements through carefully altering the physics of fluid simulations. The development process started with attempting a CPU-implementation of a fluid simulator, by creating an introductory flow and color field simulator. However, as we will discuss later, a CPU implementation is not optimal, therefore we switched to a GPU-implementation. The fluid simulations are generated from the Navier-Stokes equations and produced using a combination of advection, diffusion, divergence, density, and pressure which act on the vector-fields of the simulator. The musical analysis comes from relying on JavaScript packages and built-ins that allow playing, stopping, and analyzing songs.

Introduction

Our team developed this project to create a fluid dynamics simulator that changed depending upon musical attributes of a song (i.e., BPM). The motivation behind this project was our collective love for music and our desire to create a project that afforded us the opportunity to use graphics to visualize music. We broke the project down into two parts: fluid dynamics simulation and musical analysis. For our MVP it was important to focus on getting the basic fluid dynamics simulation working because that is the crux of our project. Additionally, simulating fluid dynamics requires a deep conceptualization of the complex math associated with it. Below is a picture of our simulator. The project is widely applicable with applications including but not limited to: allowing musical artists to visualize their music, film makers to visualize sounds, teachers to explain introduction to fluid dynamics, and someone who just wants to see their favorite songs visualized.



Previous Work

The most helpful previous work included literature, git repositories, and example projects.

Starting with example projects, the fluid and sound simulator published by YouTube creator MKGames Art & Visuals served as motivation for our project and a goal to strive towards.[1]

Another example project that helped our understanding of fluid dynamics was a Fluid Simulation project created on WebGL by Pavel Dobryakov. This project helped us understand Texture objects used in WebGL as an optimization over using arrays on a CPU. In this simulator, the user can manually enter

[1] https://www.youtube.com/watch?v=5vaOWL2Pj_g&ab_channel=MKGamesArt%26Visuals

random splats using the gui, or drag their mouse across the screen and these splats will disperse throughout the screen in a fluid like motion. The user is able to manipulate aspects of the simulation such as, density, diffusion, pressure, voticity, splat radius, and Sunrays.[2]

Moving onto literature the two most helpful pieces of literature were Chapter 38 of *GPU Gems*, "Fast Fluid Dynamics Simulation on the GPU," written by Mark J. Harris and "Fluid Simulation," a tutorial written by Jamie Wong. The work of Mark J. Harris was instrumental in understanding the key concepts of fluid dynamics; the chapter walks through Navier-Stokes Equations and breaks it down into its key components advection, diffusion, external forces, and divergence. Finally, the chapter ends with generic pseudo code to give the reader a suggestion of how to implement the project. The work of Jamie Wong, relied on Chapter 38 of GPU Gems, but provided a more tangible introduction into the topic, particularly in its synopsis of key points and equations required for implementation of a basic fluid dynamics simulator.

Approach

Our team set out to develop a music fluid simulator. To implement the project we attempted two separate approaches: a CPU version and a GPU version.

The first approach relied upon Three.Js. To get started we utilized the Three.Js starter code provided by course staff. Chapter 38 of *GPU Gems*, "Fast Fluid Dynamics Simulation on the GPU," "Fast Fluid Dynamics Simulation on the GPU," helped us understand the basics of fluid dynamics and provided pseudo code to structure our code. Jamie Wong's "Fluid Simulation," provided a simple example of implementing a fluid simulator to help start our development process. Using these two sources as help we developed our schema, our implementation relied upon six main files: fluid.js, image.js, pressurefield.js, scene.js, sim.js, and renderer.js. The bulk of the code rests in fluid.js, which we broke up by the major functions: advection, diffusion, and divergence. Utilizing this approach allowed us to split up the major functions and test functions independently. Additionally, we made the decision to implement a CPU version rather than a GPU version. Although the majority of examples we observed utilized a GPU, we prioritized using a CPU for our MVP implementation because it was a better choice given time constraints. The challenge with the CPU implementation is the lack of parallelism, thus as the image size grows the computing time increases. However as we will discuss later the CPU implementation did not work.

Our second approach was to use a GPU implementation (WebGL). Given time constraints we, in discussion with our lead TA, decided to build upon Pavel Dobryakov's simulation and focus on implementing music analysis and "wacky" features that provided different ways to simulate fluid motion. This approach relies mainly upon script.js, this is where all the fluid calculations are occurring. Additionally, the parallelism that GPU's offer makes fluid simulations possible, however, our first attempt at a CPU version made it such that we fully understood all of the components of a fluid simulator and subsequently the GPU implementation.

Methodology

For our project we attempted two separate methodologies for implementation. The first methodology required us to start from scratch: first deeply conceptualizing the math behind fluid

[2] https://paveldogreat.github.io/WebGL-Fluid-Simulation/

dynamics, then using a CPU to implement it. Through this development, we encountered restrictions such as poor runtime and the limits on concurrency of tasks, which is a problem because fluid computations are highly iterative (i.e., Jacobi iteration). Aside from runtime restrictions, minor implementation errors may have affected the success of our first attempt. We attempted to use our knowledge of THREE.js and the Image structure from Assignment 1 to represent the velocity, color, pressure and fields. However, this becomes complicated when creating methods that need to handle multiple fields made up of different vector lengths. In response to these restrictions, our second methodology required us to utilize a GPU implementation so that we could quickly render high resolution images.

First Attempt (Fluid Simulation from Scratch)

To implement our approach we required six main files: fluid.js, image.js, pressurefield.js, scene.js, sim.js, and renderer.js. We adapted from a mix of assignment 1 and assignment 5 for inspiration on storing/manipulating images as well as animation. Below we will go into detail about the design choices of each file.

Fluid.js - The Math

Fluid.js contains a Fluid object that consists of the vector and color fields, q quantities, and the forces/constraints - advection (fluid and dye moving through field), diffusion (viscosity slowing movement), pressure (used in calculations to keep velocity field divergence-free/maintain incompressible nature of fluid)- that are used to manipulate the fields. There were two possibilities for the implementation of the velocity and color fields. We could have implemented them as images or as arrays. We decided to go with the former approach as it would allow us to utilize existing image properties from previous assignments to easily display changes in data.

Image.js - Modeling flow fields and pixels vs vectors

Image.js contains an Image object that consists of a width, height, a displayable data array, and a boolean for whether it should have 2d or 3d vectors stored at each position in the image. The file was adapted from the image.js file provided in assignment one. We could have used the original image class with pixel objects for representing our color field, but the math for applying fluid simulation operations to the velocity field (x and y components for magnitude in both directions) and the color field (r, g, and b components) were similar, so we wanted to represented them with similar data types (Three.js vectors). This ensured that we did not need distinct functions for each case.

Pressurefield.js - Modeling scalar fields

We further adapted the original image.js file to have a means of storing scalar quantities with similar indexing and visualization. The object consists of a width, height, and a data array of pressures. It has all the same functions as image.js but modified to hold scalars instead of vectors.

Scene.js - Making fields available for display

Scene contains the reference to the html canvas and context. It uses these to initialize and store references to all of the needed fields (vector, color, and pressure).

Sim.js - Updating

Sim.js was adapted from the scaffolding of assignment 5. For our case, on initialization it creates a Fluid object to keep track of the state of the simulation and complete the necessary math. From then on, each time it is called, sim.js calls the advanceProgram() function in fluid.js to determine what the new vector and scalar fields should be and updates the fields in scene.js to match.

Renderer.js - Animating and displaying

Renderer.js was adapted from the scaffolding of assignment 5. It calls Sim.js with a recursive requestAnimationFrame function such that the Scene.js fields are continuously updated. After calling simulate, it displays the color field so the color's movement through the fluid can be seen.

Building on an External Fluid Sim

To implement the second approach, we utilized Pavel Dobryako's WebGL-Fluid-Simulation, which uses a GPU to simulate fluid dynamics. We focused on understanding and making substantial changes to script.js and created bpm.js. This required us to first understand Textures. After doing some research on textures, we analyzed the source code, realizing that they chose a similar approach to our original attempt. Script.js includes implementations of advection, diffusion, and divergence and pressure using the equations described in the Collaboration portion of this paper. Below we go into detail about our contributions and design choices for each file.

Script.js

Script.js contains the bulk of our code and functionality with functions that control canvas and gui configurations, audio and tempo analysis, play and pause music controls, music application, and wacky user customizations. The user customizations include the ability to toggle between multiple levels of dissipation rates, curl, mirrored advection, and types of motion all describing forces applied to the motion of the splats on the screen. Playing with different combinations of these forces alots the user the ability to simultaneously add their own unique moments, while our own scene plays.

Dissipation Rates

There are five dissipation choices: regular, fast, slow, none, strobe, and marker. All of these implementations relied upon altering the advection function, particularly velocity diffusion. Altering how quickly the velocity force dissipated allowed for the creation of cool effects.

Curl

There are three curl choices: regular, tight, and loose. All of these implementations relied upon altering the vorticity shader. The vorticity shader describes the spinning motion near a point, therefore in increasing vorticity you increase the amount of spinning or curl of each particle.

Mirrored Advection

The mirrored advection effect is done by having half of the screen default to taking its color from the other half. In the advectionShader code, we add a check to see which half of the image the coordinate being examined is on and, if it is in a location where x is less than y, it checks to see if its reflection has any color (i.e. rgb values above 0.05) which should be advected to it. If so, it takes that color and displays it. Otherwise, it displays the color which would naturally advect to this coordinate. The locations where y

is less than x do the opposite. They first check if there's any dye to be advected there, and, if not, look to see if the mirror side has dye to display.

Types of Motion

As of writing this there are four types of motion: regular, opposite, parallel, and collision. The opposite effect was done by manipulating the functions which take user 'mousemove' events and create splats, to also include splats which are reflected across the x and y axis and moving in the opposite direction. The collision effect manipulates the same functions, but creates a simultaneous splat that guarantees to collide/intersect with the original splat. Parallel manipulates the same functions, but creates a simultaneous splat that is directly parallel to the original splat.

Play and Pause Music

To successfully play and pause music an audio tag was added to index.html with the file path of the song Waterfalls by TLC. Additionally, a pause and play button were added to the gui, such that when the user hits play script.js grabs the audio element and plays the music. When the user hits pause, script.js grabs the audio element and pauses the music. The play music function also triggers applying the musical analysis.

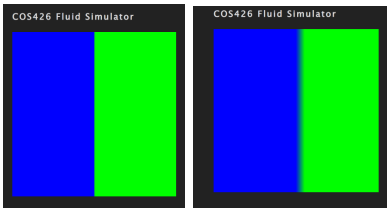
Tempo Detection and Rhythmic Splats:

Bpm.js stems from the web-audio-beat-detector npm package. Because our project did not use Node.js, we used the browserify package to convert this module into code which could be handled in the browser. With the module, we are able to load an audio sample (in our case, songs) and analyze its tempo information in beats per minute. To be clear, the tempo is not hard-coded, it is analyzed asynchronously on loading and could be made to work with more songs by calling the same asynchronous functions.

Once the animation loop sees this bpm has been analyzed and set, we are ready to draw rhythmic splats. When the user presses play on the music, we draw a random splat on the screen and record the time this splat was made. Once an amount of time equivalent to a musical measure has passed, we draw another splat. While this does not always draw splats at the beginning of the measure in the music, it maintains the same rhythm as the song such that whatever beat the song starts on receives the splat (i.e. if the song starts halfway through a measure, splats happen at the halfway point of each measure). If the user pauses, the splat now happens at whatever point in the measure they stopped the music. This allows users to pick exactly when they want splats to appear (i.e. pausing and playing on the downbeat so the splats come at the beginning of the measure) and allows for an engaging element to see how close to perfect timing with the song the user can get from clicking!

Results

Combining our focus on conceptualizing in the first approach with our focus on optimizing our engine and implementation in the second approach, we were able to develop our MVP, a 2D fluid dynamics simulator that changes depending upon musical attributes. Though we made substantial contributions to Pavel Dobryako's fluid dynamics simulation, we truly measure success from our ability to reach stretch goals - maximizing user interactions and integrating music analysis in order to manipulate flow based on music. Overall, we were able to create a 2D fluid dynamics based music visualizer with high user interaction. Below are pictures from both implementations.



Above are images of our first attempt. As mentioned previously, the CPU implementation caused lagging. The image on the left is the image at the start of the simulation (t = 0) and the picture on the right is at time after t=0. As you can see, by looking at the center, there is movement, but not enough.

Above are the images of our second attempt. The image on the right is the gui for the implementation by Pavel Dobryakov and the image on the right is the gui from our implementation, with the added folders for music and wacky.

Discussion

Due to the iterative nature of CPU's, it was not a promising approach for implementing a realistic fluid dynamic simulation. Implementing a fluid simulator requires understanding and calculating multiple complex equations, across space and time, and even multiple iterations of each equation such as in the Jacobi iterations. However, it was promising for the sake of conceptualizing the math, and confirming that we can prove this understanding through practice. Having had this deep conceptualization as a basis for approaching the project, we were able to make substantial changes to an existing simulator and integrate it with multiple different components of music analysis, creating a fluid based musical visualizer. This project could serve as a stepping stone for other forms of visualizers such as smoke, bubbles, or mud based visualizers. It could also be developed further to have thorough implementations for handling unique attributes for multiple audio types such as voice and music recordings.

Conclusion

Our project was successful, despite the first implementation not working as we would have liked, our second implementation, building upon Pavel Dobryakov's simulator proved fruitful. As a group we were able to develop a fluid dynamics simulator that changes depending on musical attributes (i.e., BPM) and renders "wacky" fluid movements through carefully altering the physics of fluid simulations. If we were to continue, next steps would include improving our musical analysis capabilities and possibly having the ability to change the music based on the flow (e.g., a backwards swipe reverses the song). Additionally, we could explore adding additional textures that are related to this implementation such as bubbles.

Contributions

All team members spent time reading Chapter 38 of *GPU Gems*, "Fast Fluid Dynamics Simulation on the GPU" and Jamie Wong's "Fluid Simulation," after completing the readings we all individually created a mockup for how we thought we should progress with the project. As a team we merged the ideas into a cohesive schema that included the major functions and how we would structure storing key values such as velocity, color, pressure. How we split the work is described below.

In the simulator from scratch, Emmandra focused on developing the advection function that transports objects along a flow. This consisted of tracing the trajectory of each particle back to its prior position, and using an interpolation of 4 values nearest to this prior position to update the attributes at the current position. We use the current particle and its previous position in the last time step as opposed to its position in the next time step as GPUs cannot change the positions of fragments that they are writing. Additionally an interpolation of surrounding values in the previous timestep are used as opposed to the value itself because using the negative. For the visualizer built with Dobryakov's simulation, Emmandra focused on developing the functionality for the Wacky Curl, where the user can toggle between the amount of curl force applied to the splats.

Equation 1: $$\vec{u}^a_{i,j} = \vec{u}^a(x := i\epsilon, y := j\epsilon, t + \Delta t) := \vec{u}(x - u_x(x,y)\Delta t, y - u_y(x,y)\Delta t,$$

Equation 2: $$\vec{c^a} = \vec{c^a}(x := i\epsilon, y := j\epsilon, t + \Delta t) := \vec{c}(x - u_x(x,y)\Delta t, y - u_y(x,y)\Delta t)$$ [3]

In our initial implementation, Toussaint worked on diffusion. Diffusion required implementing the Jacobi iteration to solve the poisson equation, which was used to find the velocity's diffusion and pressure at a particular x,y. Below is the Jacobi iteration this equation is run at least 20x during each iteration that updates the vector fields : Equation 16 $$x_{i,j}^{(k+1)} = \frac{x_{i-1,j}^{(k)} + x_{i+1,j}^{(k)} + x_{i,j-1}^{(k)} + x_{i,j+1}^{(k)} + \alpha b_{i,j}}{\beta},$$ [4]

In the final visualizer, Toussaint worked on adding additional folders to the gui, getting the music to start and stop, analyzing the frequency of the music, the dissipation features, parallel motion, and the collision feature.

In our simulator, Clayton implemented scaffolding, the flow field classes, and the divergence equation (Equation 3). The scaffolding required a system which could initialize fields storing the velocity, pressure, and color of the fluid at all points, continuously call to fluid.js and update the fields as needed, display the up-to-date color field, and simplify the process of implementing the vector and color operations necessary on the fields. The divergence equation is used to compute pressure and make the velocity field divergence-free.

For the visualizer, Clayton implemented tempo detection and rhythmic splatting, the mirror advection effect, and the opposite splat effect.

Equation 3: $$d_{i,j} = -\frac{2\epsilon\rho}{\Delta t}(u^a_{x_{i+1,j}} - u^a_{x_{i-1,j}} + u^a_{y_{i,j+1}} - u^a_{y_{i,j-1}})$$

[3] Equations 1,2,3 from: http://jamie-wong.com/2016/08/05/webgl-fluid-simulation/

[4] Equation from: https://developer.nvidia.com/gpugems/gpugems/part-vi-beyond-triangles/chapter-38-fast-fluid-dynamics-simulation-gpu

Appendix

Link to github repo: https://github.com/rtwebb/Rhythm-Flow

Link to website: https://rtwebb.github.io/Rhythm-Flow/

Honor Code

This work represents my own work in accordance with University regulations.

- Clayton Ferguson, Toussaint Webb, Emmandra Wright

Works Cited

"Advanced Techniques: Creating and Sequencing Audio - Web Apis: MDN." *Web APIs | MDN*, https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Advanced_techniques.

Ahmed ElyamaniAhmed Elyamani 48622 silver badges1111 bronze badges, et al. "Web Audio Analyze Entire Buffer." *Stack Overflow*, 1 Feb. 1965, https://stackoverflow.com/questions/43867902/web-audio-analyze-entire-buffer.

Beautypi, Wyatt. "Shadertoy." *Shadertoy* , https://www.shadertoy.com/view/XtGcDK.

Happy MachineHappy Machine 76955 silver badges2020 bronze badges. "Why Does Audio Not Play in Javascript." *Stack Overflow*, 1 Sept. 1966, https://stackoverflow.com/questions/53987982/why-does-audio-not-play-in-javascript.

Harris, Mark J. "Chapter 38. Fast Fluid Dynamics Simulation on the GPU." *NVIDIA Developer*, https://developer.nvidia.com/gpugems/gpugems/part-vi-beyond-triangles/chapter-38-fast-fluid-dynamics-simulation-gpu.

Dobryakov, Pavel. "PavelDoGreat/WebGL-Fluid-Simulation: Play with Fluids in Your Browser (Works Even on Mobile)." *GitHub*, 29 May 2020, https://github.com/PavelDoGreat/WebGL-Fluid-Simulation.

Michaud, Austin. "Building an Audio Visualizer with JavaScript." *Medium*, The Startup, 27 Jan. 2022, https://medium.com/swlh/building-a-audio-visualizer-with-javascript-324b8d420e7.

MKGames Art & Visuals. "Fluids & Sounds Simulation." *YouTube*, YouTube, 5 Feb. 2021, https://www.youtube.com/watch?v=5vaOWL2Pj_g&ab_channel=MKGamesArt%26VisualsM.

Wong, Jamie. "Fluid Simulation (with WebGL Demo)." *Zero Wind*, 5 Aug. 2016, http://jamie-wong.com/2016/08/05/webgl-fluid-simulation/.