Obtaining the Appearance of Fluid Dynamics in Real-Time Graphics

Jens Fursund

Technical University of Denmark
December 1, 2010

Figure 1: Smoke rendered with the application implemented in this thesis.

Abstract

This thesis presents a high performance method for animating and rendering passively advected fluids such as smoke, steam and fire, for real-time applications. The solution consists of; a fluid advection method which is not necessarily physically correct, but presents results which could be perceived as physically plausible, a method for creating real-time adaptive boundaries for the fluids, and a collection of different rendering techniques depending on the fluid type.
Through performance comparison with an existing technique and quality evaluations against offline renders and real video footage, this technique demonstrates that plausible animation and rendering of passively advected fluids can be created at acceptable performance rates.

1 Introduction

Fluid dynamics simulated in 3D is traditionally an area in real-time computer graphics which has not previously been simulated believably at performance rates which are acceptable for real-time graphics productions. Many impressive algorithms has been developed but most have proven to only perform at real-time rates in special scenarios. For example very few computer game productions have used 3D fluid dynamics in their games, and in these cases the effects have only been enabled for high-end hardware.
This thesis presents a method for simulating the appearance of passively advected fluids, such as smoke, steam and fire, with a performance acceptable for real-time graphics applications, while keeping the hardware requirements low and the hardware capabilities relatively few.
More specifically this thesis implements a method for animating and rendering passively advected fluids in real-time, not necessarily by using physically correct methods. Implementing this method is foremost carried out by utilizing the highly parallel hardware of a graphics processing unit (GPU), mainly on the precedence of recent fluid dynamics methods having gained performance from the same utilization.
Fluid interaction with the environment and methods for rendering the different types of fluids are also implemented, since a significant part of the perception of fluid dynamics not only depends on the correctness of the fluid computations by themselves, but also on how the fluid reacts with its environments and how it is perceived. Furthermore the techniques used for fluid interaction with the environment and rendering can also have a significant impact on the performance, which makes these methods important to consider with the scope of this thesis.
In short, the contribution presented in this thesis is:

> A solution for creating plausible animation of passively advected fluids with an acceptable performance/quality trade off for use in real-time applications on a wide range of hardware.

This covers:

- Curve-based control of fluid animation
- Real-time boundary creation for fluid-scene interaction
- Rendering different passively advected fluid types

To evaluate the performance results of the work, a comparison with an existing method and a *real* usage scenario performance evaluation, is carried out. For evaluating the visual quality a comparison with offline renderings or real video footage is performed.

2 Related Work

Obtaining the appearance of passively advected fluid dynamics can be done with different methods. Computational fluid dynamics (CFD) is a discipline including

many of these methods. It involves numerical methods and algorithms for solving and analysing fluid dynamics, amongst others, usually discretizing the Navier-Stokes partial differential equations. However, fluid flows can also be generated by utilizing procedural methods.
The following sections presents different methods for obtaining the appearance of fluid dynamics advection, and also determines the applicability in relation to the scope of this thesis. Namely making sure the performance of the method is acceptable while making passive fluid advection possible.

Grid-based Advection is an Eulerian method for simulating fluid dynamics. The method works by discretizing the Navier-Stokes partial differential equations into a grid with which a vector field is advected. The vector field is then used to push e.g. particles around.
Jos Stam, popularized an extension of this method[1] with a real-time divergence free fluid solver for games. In contrast to previous methods, which might diverge over time[2], this method is divergence free. This stability is obtained by tracing each point of the field backwards in time using the velocity at $-\Delta t$, which gives an interpolated value between the grid velocities. In principle this is not called eulerian, due to its nature of following a point in time, but rather *Semi-Lagrangian*.
One of the advantages of this method is the grid representation of the fluid which produces close to physical correct simulations even at low resolutions.
A disadvantage with this method is relatively high memory requirements, because of the different states (velocity, pressure and e.g. vorticity) which needs to be stored for each grid-position, in this case in three dimensions. Making the method run on the GPU means keeping at least 2 states across simulation steps, but usually more[3]. Using a grid also makes it suffer from *fluid-in-a-box* problem, which basically means that the fluid can not expand outside the grid. There have been examples of solving this problem, e.g. with *Translating Eulerian Grids*[4].
Though there are example implementations of this technique running at interactive frame-rates, with impressive results, they still require high-end hardware to have reasonable performance and appearance in interactive applications[5]. As the objectives of this thesis requires a much lower performance penalty and not necessarily complete physical correctness, the *Grid-based Advection* method is found to be inapplicable.

Real-time Practical Volumetric Effects Simulation is a method developed to take advantage of Eulerian based fluid dynamics methods without doing a full 3D-simulation of the fluids[6]. The method computes the fluid dynamics in 2D, and subsequently rotates the advection around an arbitrary axis in 3D, thus extending the 2D simulation into 3D. The 3D advection is modified by 3D noise to make the result more dynamic.
This method is very fast, because it only requires fluid computations in 2D. Furthermore it creates sufficiently realistic results for 3D fluid animations. These animations have to be somewhat symmetric around one axis to provide realistic results, making the method inflexible, which is a major drawback.
Though this method has a high performance, the inflexibility of the symmetry is not suitable for creating a proper dynamic method, hence it is deemed not suitable for the purposes of this thesis.

Smoothed Particle Hydrodynamics (SPH) is a particle-based Lagrangian method for simulating fluid dynamics. The method discretizes fluid dynamics by sampling interacting smoothed volumetric particles. The fluids are represented by particles and calculation of the fluid properties are done by using the particles as interpolation points[7].
For each particle a smoothing kernel is defined that, determined by a pre-defined distance numerically sums up the properties of the nearby particles. This way each particle is affected by the surrounding particles. Different smoothing kernels can be used to calculate different properties of fluids such as pressure and viscosity[8].
The main advantage of SPH is the inherent conservation of mass because each particle represents a fixed mass, which in Eulerian methods is a computational overhead.
A disadvantage of SPH is the significant amount of particles needed for obtaining the appearance of realistic fluids, which means it is quite taxing to get the same results as Eulerian implementations[9]. Therefore most implementations that run at interactive rates perform calculations on the GPU[10][11].
On low-end hardware SPH is not performant enough, while still achieving believable appearance[10], which makes SPH unfit for the scope of this thesis.

Curl-noise for Procedural Fluid Flow (Curl-noise) is a procedural method for generating a plausible fluid advection field. No discretization of the Navier-Stokes equations is needed, as it is decoupled from physically correct simulations[12]. However, the strength of the Curl-noise method is that it creates a plausible fluid vector field while being divergence free. Hence it can create the appearance of fluid advection. Furthermore the method also allows for modelling boundary conditions in a simple manner.
Curl-noise models a plausible vector field by evaluating the curl of 3D Perlin noise[13]. It is a completely parallel method, since no information is needed from the surrounding particles. Everything is based on the curl of the vector field, which in this case is generated by Perlin noise. Boundaries can be added by gradually exchanging the influence from the noise potential with the normal of the surface depending on the distance to the boundary surface.
The main advantage of this method is its potentially low memory usage and low cost computation, compared to the previously mentioned methods (SPH and Grid-based advection) the computational difference seems quite big. There is no published data on the performance of the Curl-noise method in a real-time context, but the relative few operations needed to calculate the Curl-noise indicates a high performance. The possibility of making this method highly parallel is also an advantage if the execution of the method is performed on the GPU. One example of this

has been demonstrated in [14].
A disadvantage of this method is its physical in-correctness which might lead to undesirable visual results. As most passively advected fluids can be considered chaotic this might not be a problem. However the procedural nature of the method does not make it readily extensible to actively advected fluids.

For this thesis, Curl-noise has been chosen as the base technique for obtaining the appearance of passively advected fluid dynamics, as this technique seems to provide a good trade-off between performance and visual likeness of passively advected fluid dynamics. Though the technique is not based on a discretization of the Navier-Stokes equations, it does provide a divergence free plausible advection, which can be used to create the appearance of advected fluids. Furthermore the Curl-noise method also includes a method to handle collision with objects.
These features combined fits the objectives of this thesis. The following section outlines how this method is used to create the appearance of fluid dynamics in real-time.

3 Algorithm Outline

To enable a real-time 3D application where fluids are advected, affected by obstacles and represented visually, several steps have to be performed. First a technique for simulating the fluid is set up, in this case using the Curl-noise method. Subsequently a representation of the objects the fluid will interact with is created, and if moving objects are desired the representation will have to be updated accordingly. Finally a rendering of the fluid, depending on the fluid type, is required to visualize the fluid.
Curl-noise, as used for fluid advection, can, by being a parallel method, benefit from evaluation on the GPU. Therefore this implementation uses the GPU for most of the calculations. As evident in the algorithm outline this has a significant impact on the design of the method. If not otherwise stated it can be expected that any computations are evaluated on the GPU. However, as the scope of this thesis is to target a broad range of hardware, high-end features such as the recent compute features in modern graphics cards (DirectCompute, CUDA, OpenCL) are avoided.
Beneath are the steps performed to advect the positions, make them interact with the environment and render them.

1. Set up ping-pong rendertextures for emission position and life. One pixel per particle.
2. Set up ping-pong rendertextures for position. One pixel particle.
3. Construct buffer used for boundary conditions.
4. Add forces to vector field, dependent on the fluid type.
5. Add smooth noise to vector field, dependent on the fluid type.
6. Add forces to respect boundaries of the world.
7. Evaluate curl by finite differences on the vector field.
8. Advect positions with the curl result
9. Move the vertices using vertex texture fetching.
10. Render dependent on the fluid type.

The following sections present the implementation details of the algorithm outline above.

4 Advecting Particles

As popularized by other GPU-based particle system implementations[15], advection of the particles is performed on the GPU using a ping-pong between two floating-point rendertextures with the particles's positions in them. Floating point rendertextures are used for higher precision, because regular 8-bit textures were deemed inadequate for smooth animation. The advection is solely based on the position of the particle, as the vector field is constructed using smooth noise and external forces, such as boundaries.
For each particle the previous position is read, advected by the Curl-noise evaluation and written back to the new position buffer. Another pair of ping-pong rendertextures contains the emission position and life of the particles. The life of a particle is decreased each frame. When it reaches zero both position and life is reset. The initial life of a particle is randomized to make the particles more dynamic, and make the emission of particles reach a continuous flow over time.
Later in the pipeline the particle positions are read back in the vertex shader with vertex texture fetching (VTF) to position the particles in the scene. This sets the hardware requirements at Shader Model 3.0, which is hardware from 2004 and on. This is an acceptable limitation, considering the scope of the thesis, because the majority of the hardware in use would be compatible[16].
Figure 2 on the following page illustrates the pipeline of advecting the particles.

5 Curl-Noise

As the Curl-noise method is used for the advection of the particles, the curl ($\nabla\times$) of a given vector field is the basis of all advection done in this solution. As the curl of a vector field is automatically divergence free ($\nabla \cdot \nabla\times = 0$), making the advection devoid of gutters or sinks, it fits most fluids property of being seemingly incompressible[12]. Curl can intuitively be understood as a value or a vector in two dimensions versus three dimensions, describing the rotation about a point in a vector field. For a given 3D vector field $\vec{\psi} = (\psi_1, \psi_2, \psi_3)$ the curl vector is calculated by:

$$\vec{v}(x,y,z) = \left(\frac{\delta\psi_3}{\delta y} - \frac{\delta\psi_2}{\delta z}, \frac{\delta\psi_1}{\delta z} - \frac{\delta\psi_3}{\delta x}, \frac{\delta\psi_2}{\delta x} - \frac{\delta\psi_1}{\delta y}\right)$$

Curl can be explained by the following example; imagine a small wheel stuck at a certain point in a flow, e.g. a stream. If the flow around the wheel has the exact same velocity on both sides, the wheel will not rotate. However, if the flow is slightly different on one side the wheel will rotate. The rotation this creates is the curl of that point in the vector field.

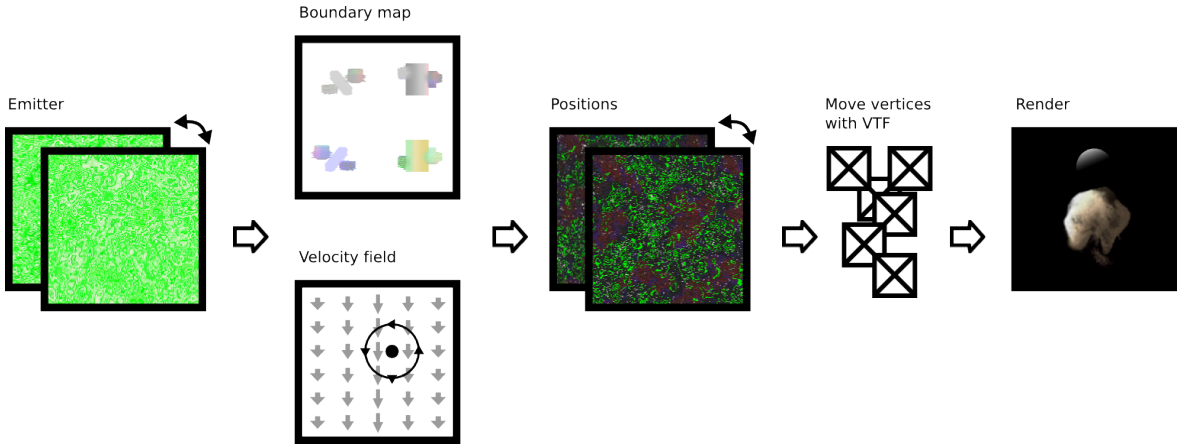


Figure 2: Pipeline from emitter map, advecting on to rendering.

In implementation, the discretization of the curl differential operator is done by finite difference approximation, requiring six evaluations of the vector field for each particle. Which essentially implies six evaluations of whatever force is suppose to make impact on the advection.
Using the curl of a vector field for advecting particles has one drawback. As the evaluation of curl on a vector field is what drives the simulation, all vector field modifications that are trying to direct the flow of the simulation have to be indirect. Thus the advection has to be directed through how the curl is expected to handle the modification. The next section provides one way of controlling the advection.

5.1 Controlling Curl

While the Curl-noise method in itself provides advection by the means of smooth noise, it might also be desirable to be able to control the advection by other means. Controlling the noise advection alone is not enough to produce advection for different kinds of fluid flows. I.e. trying to make the particles move upwards, would require the vector field to change to make the curl of the vector field return a vector pointing upwards. Intuitively, this could be achieved by evaluating the curl of the cross product of the vector field with the upwards vector. More generally this can be expressed as:

$$\nabla \times (\tfrac{\vec{u}\times\vec{x}}{2}) = \vec{u}$$

Where $\vec{u}$ is the vector describing the direction of the advection.

In application this can be used to define animation curves for the vector field, enabling designers to control the Curl-noise more easily over time. To enable this functionality, here a method has been employed where designers can draw animation curves, that are sampled offline and put into a texture. During the curl evaluation the animation curve texture is sampled with the lifetime or some other value of the particle, making particles move along the curve individually. Figure 3 demonstrates the pipeline.

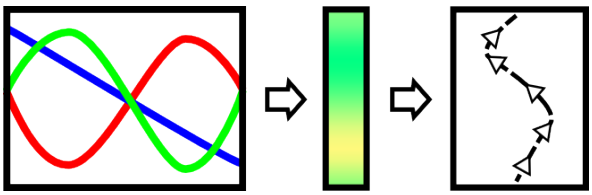


Figure 3: Pipeline from animation curve, to curve texture and on to animating particles.

5.2 Noise

To create turbulent velocity fields, the Curl-noise method uses 3D Perlin smooth noise to generate a smoothly varying velocity field. In reality any smooth noise technique can be used to generate the turbulent velocity field.
In [12] it is furthermore suggested that a noise method which changes with time could be desirable to add more realism. For evolving low turbulent fluid advection, such as cigaret smoke, this can however lead to swaying motions throughout the whole advection. This occur because the whole vector field is offset by the same value, which breaks the illusion of evolving advection.
Not having any change in the fourth dimension of the noise, does however lead to no flowing change in the advection, i.e. particles seem to follow the same path. This static advection appears because the deterministic nature of the smooth noise creates vector fields that always evolve in the same manner. For now a mechanism for randomly switching the initial emission map has been employed to get around this issue. However, it is not a solution which can be used if a static initial emission map is required.
As the objective of this thesis is to obtain the appearance of fluid dynamics in real-time, different approaches for producing the proper smooth noise on the GPU has been investigated.
Most smooth noise methods require quite a few instructions to evaluate in their raw form. Therefore the first technique explored was to use texture-based statically generated noise, because that would use very few instructions to evaluate. In cases where the turbulence in the velocity field is not expected to change character during

its lifetime, texture-based noise works excellently. This is most often not the case though, dynamically changing the noise frequency during a particle's lifetime can prove useful when wanting to introduce more turbulence into the velocity field.
Generating smooth noise, and especially Perlin noise, on the GPU is a thoroughly investigated research area. Many have sought to make Perlin noise faster on the GPU, thus a significant number of different methods already exist. To determine which of these methods to use, several smooth noise implementations optimized for the GPU have been investigated. The main requirements for the methods being that they both evaluated to a relatively smooth noise at both high and low frequencies, and that they performed as fast as possible.

Modified Noise[17] (mNoise) is by far the fastest implementation performance-wise, but proved to provide relatively non-smooth results at low frequencies.

Quick Noise[18] is sufficiently fast, but again provides non-smooth results at low frequencies.

Improved Perlin Noise[19] (GPU Gems noise) unfortunately does not provide the same good performance as the other implementations, but it does provide a smooth result in the range of frequencies needed thus far.

IQ Noise[20] is not Perlin noise, but a different smooth noise implementation. It is a texture sampling free implementation. IQ Noise has a relatively good performance, and provides a smooth noise result in the required frequencies. However, IQ Noise requires the use of bit-shifting in shaders, which only the most recent graphics cards support, consequently IQ Noise does not fit within the scope.
These different noise implementations could potentially be swapped in and out depending on the use case, but the aim is to achieve relatively high performance. Based on this *Modified Noise* is used, though it does not produce smooth results at low frequencies.

6 Passive Advection Schemes

Passive advection can be described as fluids which perceptually do not have interacting particles. Examples of such fluids could be fine smoke e.g. cigaret smoke, or steam from a coffee cup, smoke from a chimney or flames from a fire. In order to obtain the appearance of these fluids with the Curl-noise method, the vector field is modified according to specific fluid types. Creating vector fields that produce an advection resembling these kinds of fluids when using Curl-noise, is not necessarily intuitive.
The following sections presents how vector fields are created for two different kinds of passively advected fluids.

Wispy Smoke vector fields represents phenomenon, such as cigaret smoke or steam from a coffee cup. In general the vector field does not require much modification to the main Curl-noise method. Wispy smoke's advection is low frequency, with a tendency to increase in frequency as it evolves. Creating the vector field for wispy smoke requires a modification to move in a desired direction, most likely upwards. To this is added the Perlin-noise based vector field starting with a relatively low frequency and increasingly higher frequency depending on the life of each particle. When nearing boundaries the noise frequency is increased slowly, followed by a slow decrease to give the impression of added turbulence on collision.

Billowing Smoke is what would usually appear from factory chimneys or steam engines on old trains. The smoke is usually high density and fast moving smoke, with a pulsing advection in billows. To create an advection resembling billowing smoke, vortex rings are added to the vector field, as described in [12].
Vortex rings are created by a modification of the vector field given, by multiplying it with a variable dependent on the distance to a point in space, as illustrated by figure 4. However, without a moving point the vortices will stay in the same place, since the vortex generation is dependent on the position of the particles, due to the curl evaluation being position based. Therefore the point to which they are generating the vortices is moved upwards controlled by an external parameter. Furthermore the vector field is altered almost as the wispy smoke, though the noise has a significantly smaller influence on the advection.

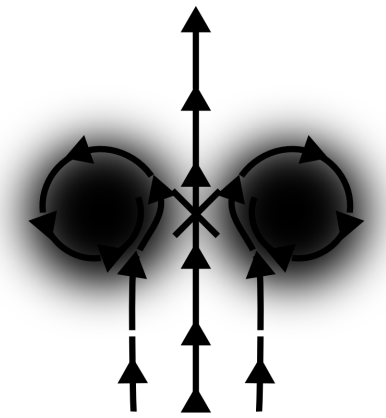


Figure 4: Vortices are generated depending on a distance to a specified point. Using external parameters the vortex point can be moved upwards.

7 Boundary Conditions

The perception of a fluid is also reliant on its interaction with the surrounding environment. For a fluid to react with its environment, boundaries are defined to limit the fluids possible propagation.
Curl-noise contains a method for defining boundary conditions which can be used to steer particles around obstacles. The method is based on the distance to the objects surface and the object normal ($\hat{n}$):

$$\vec{\psi}_{constrained}(\vec{x}) = \alpha\vec{\psi}(\vec{x}) + (1-\alpha)\hat{n}(\hat{n}\cdot\vec{\psi}(\vec{x}))$$

Here $\vec{\psi}(\vec{x})$ is the vector field produced by e.g. the Perlin noise function. This formulation phases out the influence of the vector field $\vec{\psi}(\vec{x})$ as the particle nears the surface of the object while it phases in the influence of the normal. α is the value used to linearly interpolate between the two

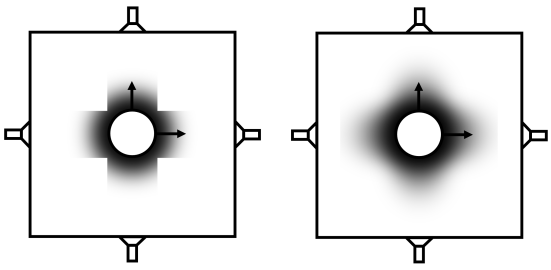
values. It is calculated with the absolute value of a ramp of the distance through -1 to 1. Making it possible to use it on both sides of a boundary. In this implementation the ramp suggested in [12] is replaced by a linear ramp to the surface to reduce computation on the GPU.
As described in [12], using interpolation formulation might produce spikes in the advection occur at sharp edges[12], mainly because it can create a significant change of direction in the vector field.

7.1 Boundary Map

To create the basis for the boundary conditions, a representation of the scene is needed from which each particle can get distance to the objects and their normals. Optimally the representation should support moving objects in the scene, hence making it update in real-time, or close to, is a requirement. The representation could be created with one of many methods, the following determines which of a few of these is applicable.

Constructive Solid Geometry is a completely procedural method for defining geometry as mathematical functions. This method can be used to define the boundaries of the fluids movement by calculating the distance and the normal with mathematical functions, similar to what is done with distance field rendering[21]. However, a CSG representation of the scene does not work for arbitrary geometry and requires each particle to check each object in the scene, thus another method is used.

Voxelization in real-time could be used as an approach to generate the scene boundaries, e.g. using techniques such as the one-pass voxelization method[22]. However, voxelization techniques rarely includes normal information, in this case needed to adjust the advection for the boundaries. It can be produced by either calculating it with central differences or included by increasing the size of the data structure, both requiring either extra computations and texture reads or higher memory requirements. Tradeoffs like these seem unnecessary.
Lastly the scene could be *voxelized* by a slice-map approach as suggested in [23]. This approach can contain both position (by means of depth and texture coordinate) and normal in one rendertexture, though for simplicity of setup, another approach has been used.



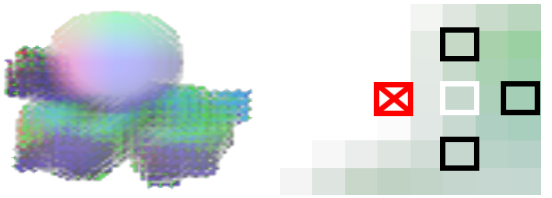
(a) Discontinuous depth (b) Continuous depth

Figure 5: The setup for creating boundary conditions based on depth from six cameras

In this thesis, enabling both real-time updated boundaries and normal information is made possible by creating six orthographic camera based depth-normal buffers encapsulating the scene from the three axes of rotation. It is inspired by the voxelization technique suggested in [24]. Figure 5(b) demonstrates the setup.
In this particular case one rendertexture is created as an atlas of the depth-normal buffers to reduce texture uploads. Furthermore the scene is rendered at a relatively low resolution, to increase performance, but also because very accurate information is not required for the boundaries.
This gives the position and normal information about the scene needed to create a relatively crude physics interaction. While this model for building a scene collision map is easy to set up and has a low overhead for both build up and reading, it does have some drawbacks. For example it does not create a complete representation of the scene if geometry is completely occluded, and testing for collision requires reading the information from all directions written to the collision map.
Creating infinite amounts of depth-normal buffers, from all directions, would yield the most correct representation of the objects, since that would catch all crevices etc.. Though for the complexity of the scenes used in this implementation two axes of rotation or four buffers were deemed enough.

Blurring the Boundary

To determine whether a particle has collided with an object in the scene, the four buffers' depth are checked against the corresponding coordinate component. Checking whether a particle is inside or outside an object will not suffice though, since it will produce a binary switch to the vector field pushing the particles around the obstacle. This could create significant spikes in the advection, as explained earlier.

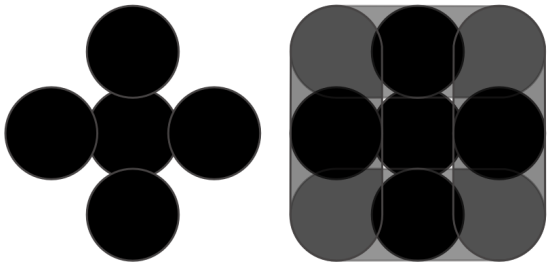


(a) Blurred dragon with sphere on top (b) Only non-zero depth is included in the blur

Figure 6: Boundary map blur

Eliminating the spikes in advection can be done by a smooth ramp up to the boundary which is used to interpolate between the two vector fields. As demonstrated in figure 5(a), naïvely using the depth up to the boundary as ramp will create an improper depth representation because the depth information is not three dimensional, but restricted to the projection towards the camera. An improper depth representation, and especially a discontinuous one, will make the vector field change as if the boundaries were different than the real object.

The solution applied here is to expand the objects boundary by using a depth aware blur on the depth-normal map. The depth aware blur is only averaging non-zero depth values into the resulting texture in order to ensure depth does not decrease around edges. Figure 6(b) on the previous page demonstrates the blur method. Seen from one of the axes of rotation the blur only increases the range of the objects in the orthogonal plane, meaning that the depth of the objects will not fit the range applied in one of the other axes. In other words, the blur will only work in two dimensions but from different axes. Hence a constant value is subtracted from the depth of the objects, increasing their range along all axes. Figure 7 illustrates the problem in two dimensions.
With the depth blur added, the linear interpolation between the two vector fields goes from the *thickened* depth of the boundary and inwards with a proper range fitting the actual boundary.



(a) Without depth subtraction (b) With depth subtraction

Figure 7: Exaggerated blurred depth in two dimensions

This solution does have several drawbacks to consider. First of all it further occludes other geometry, increasing the possibility of wrong scene representation. Secondly blurring does not necessarily give a proper representation of the object, see e.g. figure 6(a) on the previous page where the blur representation is relatively different from the actual object, and also blurs into objects nearby. Lastly, while it does create a continuous representation, it is still a somewhat a skewed representation, because the ramp is still projected linearly from the camera, as demonstrated in 5(b) on the preceding page.

8 Rendering

When creating fluid dynamics effects for real-time applications, there is often a need for rendering these in a way which represents the visual properties of the fluids. The rendering technique usually has a significant impact on the performance and visual quality. Thereby the rendering of the effect is also important to consider when evaluating the fluid dynamic technique. As the method implemented here is Lagrangian, it is relevant to interpret the results of the advection as positions in space, opposed to e.g. a volume of density.
To determine which rendering method to use, the visual properties of the fluid have been analyzed, e.g. how dense the fluid is and if shading, lighting and shadowing of the fluid makes the fluid more visually convincing. The visual appearance of most passively advected fluids such as most gaseous phenomenons, usually rely on the density of the fluid. Radiance through each pixel on screen is a factor of the thickness through the fluid volume, which again is a factor of the particle density and each particles absorption of incident irradiance.
Particle systems, or particle rendering is a very common technique in real-time applications. It is a good candidate for rendering semi-transparent volumes with discontinuous surfaces like most passively advected fluids, since the particles easily disperse and create non-continuous areas. On the other hand, the density of the particles can be a factor which creates the illusion of a coherently flowing fluid. Therefore particle rendering does not always present the best visual result. The following sections present different types of rendering techniques for the visual properties of two different passively advected fluids, which are applied in this thesis.

Semi-transparent Smoke is e.g. cigaret smoke or steam from a coffee cup. These have a low density of particles, and can be rendered without relying on lighting and shadowing. Lighting and shadowing does affect the appearance of this type of smoke, but in only dramatically in few situations, so for the sake of simplicity these situations have been disregarded.
Two methods for rendering semi-transparent smoke have been applied. The first method uses alpha blended particles which, depending on their lifetime, are slowly faded to complete transparency. Furthermore the particle size is continuously growing controlled by the lifetime of the particle. This produces the effect of the particles dispersing towards the end of their lifetime, because the size increase together with the increasing transparency gives a blur-like visual appearance. The end of the lifetime of a particle is normally towards the top of the smoke stream, which is also where the smoke particles would be more dispersed.
The second method renders the advected points as vertices on a mesh. Using a mesh captures the features of the advection more continuously since each point is connected, which makes the volume of smoke more visually coherent. As with the particle rendering method the mesh is also alpha-blended and faded to transparent towards the end of each points lifetime. Animating a mesh with Curl-noise is only possible due to the deterministic nature of the method. This implies that the vertices on the surface of a mesh can be expected not to create overlapping triangles. To properly animate the mesh, the emission map (or texture) of the particles is set to the points of the vertices, and each vertexś second texture coordinate it set to read at the respective position in the emission map.

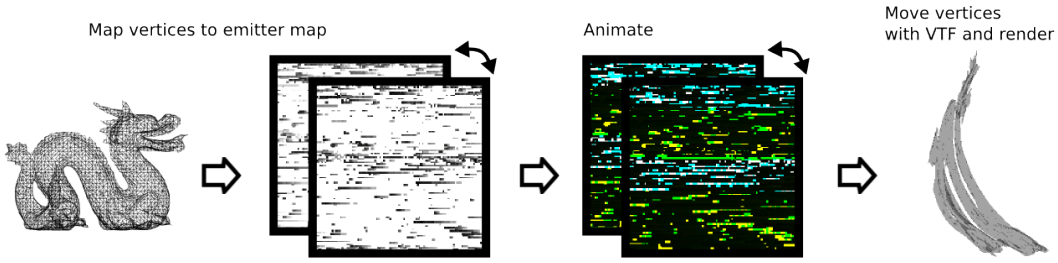


Figure 8: Pipeline from original mesh, to emission map and on to animating the mesh.

This is done by rendering points, that are positioned on screen at the proper uv-coordinate, into a floating-point rendertexture. To make sure that the precision of the vertices' position are maintained, the on screen points have the vertex position recorded in their uv-coordinate. Subsequently the uv-coordinate is written to the render-texture as the color of the point. The mesh used for rendering is highly tessellated to ensure that the animation of the points do not force the vertices too far apart, which creates non-smooth rendering of the animation. Figure 8 illustrates the pipeline.
Rendering the advected points as a mesh does not come without drawbacks. There is currently no ideal way to handle collisions with the world since triangles might end up intersecting with world geometry because of points moving on either side of a boundary. Furthermore there is currently no way to randomize the points lifetime or position, because that can make points move to far from each other. In general it is tentative to make sure that the points do not move too far apart to achieve a visual smoothness of the animation.

Opaque Smoke is smoke with a high density of particles, such as smoke from factory engines or large fires. In this kind of smoke the shading, lighting and shadowing has a larger role in defining the visual properties, since the density of the smoke creates more light scattering and transmittance. Capturing the self-shadows and backlighting effects the density of the smoke creates is traditionally done with opacity maps. Opacity maps slices the scene in slabs projected from the light, recording the objects in the current slab and the objects in the slabs in front into the map.
The most wide-spread technique for real-time opacity maps is the *Deep Opacity Map* technique[25]. This technique is applied here by first determining the beginning depth of the first layer with a single depth pass. Afterwards a second pass progressively accumulates the particles of sequential layers seen from a light into the RGBA channels of a rendertexture.
The shading of the particles is *deferred* as in each particle is not shaded by the opacity map while being rendered, but a *shade map* where each particle has its own pixel for coloring is used. Since all positions of the particles are stored in a position map already, the shading of the particles can be determined in a pixel shader. Thereby reducing the overhead of computing the logic of determining the shading from the opacity map in each particles pixel shader. This shade map can also be used for other shading purposes, such as lighting and lifetime based coloring. The same technique is used in [14]. The following pseudo code presents the steps to achieve this.

```
for each pixel in opacity slice
 add up particle count
 add with previous slices particle count
 write to channel based on slice

for each particle in position map
 get position from position map
 convert to light screen space
 read opacity map based on depth from light
 write color of particle to color map

for each particle in vertex buffer
 read color from particle color map
 render with color
```

9 Results

The results presented in this section are divided into two different evaluations. First the performance is evaluated by comparing to an existing real-time fluid dynamics method and to a CPU-based particle system in a real-time graphics application. Subsequently the quality is compared to an offline simulation and render or real-video footage.

9.1 Performance

Due to the scope defined of keeping hardware requirements low, it is first of all important to evaluate whether the performance of the method used is acceptable.
Acceptable performance is of course highly subjective, it is therefore defined here as a maximum of 16 ms, for both simulating and rendering, which is 60 evaluations of the method per second.
Testing the simulation and rendering was carried out using the most performance intensive setup of the method; animating with the billowing smoke animation with boundary conditions enabled and rendering with deep opacity maps. Executing this setup with 1 million particles (1024x1024 rendertexture) has proven to execute at an average of 40 ms with a NVIDIA GTX260 on a Core 2 Duo 2.4 Ghz CPU running in a 1024x768 resolution.

Table 1: This test was executed on a Core 2 Duo 2.4 GHz with a NVIDIA GTX260 graphics card at 1024x768 resolution. The test is using the billowing smoke animation the most expensive in the advection schemes. For boundary conditions a scene with one sphere was set up, and for rendering, the deep opacity maps were used. Some tests failed due to too high memory requirements.

No. of Particles	Animation only	Animation + Physics	Animation + Physics + Render
16384	0.556 ms	1.040 ms	1.733 ms
65536	0.473 ms	1.036 ms	2.356 ms
266752	4.169 ms	6.605 ms	16.483 ms
1048576	4.171 ms	8.054 ms	39.660 ms
4194304	16.437 ms	36.951 ms	N/A
16777216	64.452 ms	276.366 ms	N/A

This is well above the acceptable performance, but also using the most taxing setup of the method. For one pass of this setup the particles will need to be rendered once on screen and twice for deep opacity maps. The application is mostly constant in frame time, but one variable factor is the depth normal map generation for the physics which is dependent on the amount of meshes and their tessellation. The simulation by itself can perform with the pre-defined acceptable rates with up to 4 million particles (2048x2048 rendertexture), and with boundary conditions enabled using up to 1 million particles. With rendering enabled acceptable performance rates can be achieved with up to 260 thousand particles (512x512 rendertexture). As table 1 demonstrates different tiers can be used for different performance requirements, e.g. enabling and disabling different parts of the method and changing amount of particles.

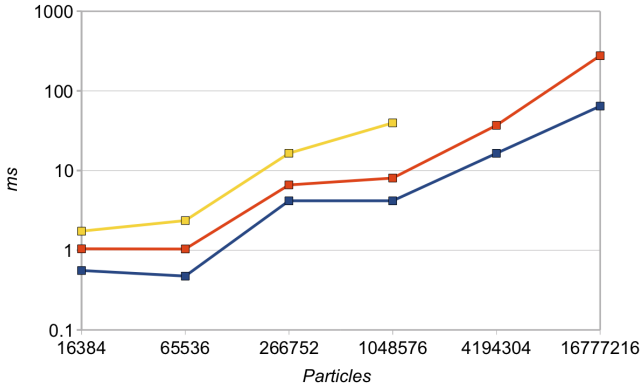


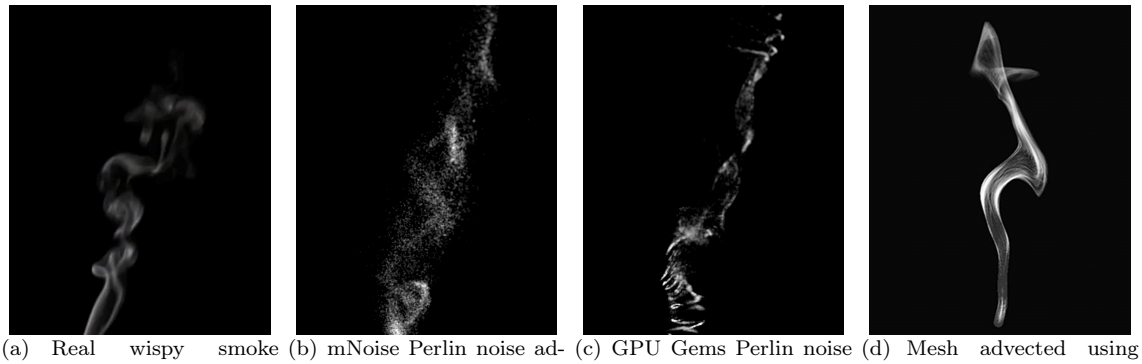
Figure 9: Performance evolution using different numbers of particles. Blue: Animation only. Red: Animation + Physics. Yellow: Animation + Physics + Render. The scale is logarithmic.

In all configurations of the simulation the application's performance was proven to be GPU-bound. Enabling boundary maps for the simulation approximately doubled the acceleration in performance when increasing particle amount. The same tendency can be observed when enabling the rendering of the particles. Figure 9 illustrates the performance increase.
With physics, animation and rendering enabled at 1024x768 resolution, the application was proven to be fill-rate bound. Which point towards performance of different rendering techniques giving different outcomes in this regard.
Comparing the simulation numbers with a modern grid-based solver computing a 64 x 16 x 64 grid size (65536 cells) on a NVIDIA GTX285 graphics card, the method applied here shows a significant performance advantage. The grid-based solver uses 14.8 milliseconds per frame on slightly better hardware, where this solution computes the same amount of points in 1.0 milliseconds, 14.8x performance difference. Comparatively this method will simulate approximately 2 million points in the same amount of time assuming the same evolution of performance represented by the test results.

To further evaluate the performance a *real* usage scenario was applied. The scenario consists of an existing real-time graphics application, more specifically the *Boot Camp* game production developed in-house at Unity Technologies[26]. The game already contains several smoke simulations using the Unity Engine's own particle system. This made the game ideal for comparing *real* usage scenario performance, since the particle system could be exchanged with the one implemented here. The tests were created by exchanging the billowing smoke from a fire, with the billowing smoke from this solution.
The CPU particle system was using 65 particles, compared to 65536 particles using the system implemented here. The tests were run on the same hardware as above. As table 2 presents, the performance overhead in using the implemented method is 1.225 ms compared to Unity's built-in particle system. Compared to the Boot Camp demo without any particle system the implementation's overhead is 1.418 ms.

Table 2: This test was executed on a Core 2 Duo 2.4 GHz with a NVIDIA GTX260 graphics card at 1024x768 resolution. The test is using Unity Technologies[26] Boot Camp production. The results presents a test without particle system, a test with the Unity Engine's built-in CPU-based particle system and a test with the GPU-based method applied here.

No particle system	50.148 ms
CPU (65 particles)	50.341 ms
GPU (65536 particles)	51.566 ms



(a) Real wispy smoke image[27].
(b) mNoise Perlin noise advection.
(c) GPU Gems Perlin noise advection.
(d) Mesh advected using GPU Gems Perlin noise.

Figure 10: Animations and renderings of wispy smoke.

If the solution was the bottleneck in this case, the performance overhead was expected to be approximately 2.356 ms, as demonstrated in the simple scene above, but it seems there are other bottlenecks in this production. Therefore the result is a smaller overhead.
Though the solution implemented here has an overhead compared to the existing solution, the increased capabilities vs. the performance tradeoff is deemed to be worth considering for real-time applications.

9.2 Quality

The quality of the animation and rendering of the final results are evaluated by comparing either offline renderings, using known techniques, or real moving images to the renderings produced by this technique. Preferably a comparison with existing real-time techniques should be performed, but it has proven difficult to find renderings which were easy to reproduce for comparison. Therefore real moving images have been used for comparison where the images have not included too many disturbing elements, e.g. background and lighting being hard to reproduce. Where no real moving images have been possible to find for comparison offline renderings are used as benchmark, since they have significantly longer time to produce the same images, hence potentially enabling higher quality. These benchmarks enables the features of the method presented here to be compared for characteristics with images that either have a strong similarity to or are real phenomena.
The quality results are divided into two comparisons, one for each of the types of passive advection implemented here; wispy and billowing smoke.

Wispy Smoke
The wispy smoke has been tested by comparing it to real video footage from [27], enabling an assessment of the characteristics of the synthesized smoke versus the real video footage. Overall the results did not catch the characteristics of wispy smoke in a very high degree, but using the mesh rendering technique the animation and rendering is deemed to be perceived as a plausible passively advected fluid. Figure 10 presents the results.
As the tecnique presented here uses the mNoise Perlin noise[17] to add smooth turbulence to the vector field, the first quality evaluation was performed using this technique.
Assessing the animation of the wispy smoke (figure 10(b)), it generates animation which has some of the same turbulent characteristics as the real video footage, however with much less detail. Furthermore the animation seems relatively repetitive, and does not achieve the same rapid change in the larger scale animations.
Because of the coarse animation produced by the mNoise Perlin noise advection, a second test was created to determine whether a Perlin noise technique with smoother noise at low frequencies would produce an animation closer to the characteristics of the real smoke. The GPU Gems Perlin noise technique[19] was used to perform this test. The performance of using the GPU Gems Perlin noise technique has not been taken into account here, but it can be expected that using it will affect the performance of the simulation, merely because it has a significantly higher amount of instructions.
Figure 10(c) presents a still image from using the GPU Gems Perlin noise. The animation produced using this technique creates more fine scale animation in the advection and also more turbulent animation, a direct result of the smoother noise in the low frequencies. There is a tendency towards clumping of the advection in fine lines, which is a product of the fine scaled animation. Thereby the animation seems to loose some of the bigger advection streams which is a characteristic of the real video footage, and to some degree also perceivable with use of the mNoise Perlin noise. This could however be a product of rendering with particles.
Lastly both techniques do not contain the same clear animation of rolling vortices through the animation as observed in the real smoke.
Evaluating the rendering, the particle based rendering of the wispy smoke does not represent the characteristics of the real smoke as convincingly as the mesh rendering presented in figure 10(d). This is foremost a product of the coherent visual look produced by the mesh rendering's *connected* particles, which is how the real smoke mostly appears.

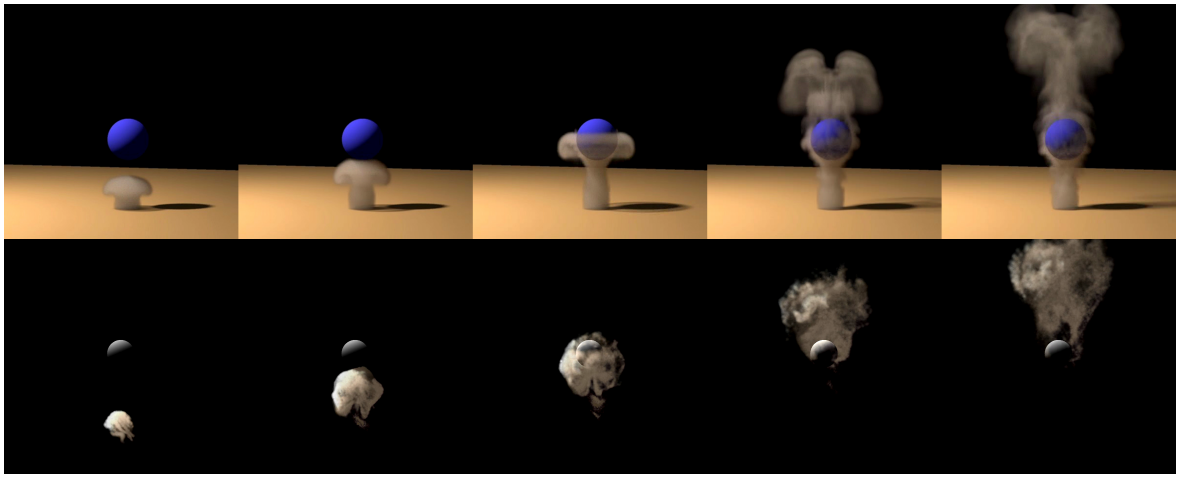


Figure 11: Comparison with five frames of an offline rendered smoke plume[28]. Top is offline rendered smoke, bottom is the technique presented here.

While the mesh rendering has the advantage of being able to produce a coherent rendering, it has the disadvantage of not being able to produce the same turbulence as the particle based rendering, due to each point being connected as polygons.
Overall the particle animation and rendering is not obtaining a particularly strong likeness of the real smoke, neither using the mNoise or GPU Gems Perlin noise, which might be a product of the particle rendering itself. This seems to be evident in the mesh rendering, since it produces characteristics much closer to the real smoke, though it is using the same animation technique. However, the animation of the mesh rendering creates an animation far slower and less turbulent than the video of the real smoke, so it remains to be seen if it could produce the same turbulent results.
Increasing the amount of particles could increase rendering quality, but, as demonstrated, would have a significant impact on the performance.

Billowing Smoke
Figure 11 presents a comparison between the *Semi-Lagrangian* fluid computation from [28] and a billowing smoke rendering carried out with the technique presented here. The primary goal of the rendering was to create a plume which would compare to the offline rendering. The rendering and animation was done with approximately 260 thousand particles. In the performance results this seemed to provide a good middle-ground between performance and amount of particles, at least on the hardware used for testing.
The real-time rendering alone resulted in a relatively plausible passively advected fluid animation and rendering of the billowing smoke, however comparing it to the characteristics of the offline rendering, differences were found.
Beginning with the animation there are some quite significant differences between the two renderings.
First of all there is a significant visual difference between the two methods in the beginning of the animation. The offline rendered animation is significantly less noise-induced in the beginning, which makes the plume shape more distinct. The difference seen with this technique is a direct result of the starting positions defined by a Perlin noise-induced emission map. A more *designed* emission map might produce a more clean result in the beginning.
Later in the animation the collision with the sphere does not create the same distinct widened vortex ring as the offline rendered version, it only seems to diffuse the advection more. Here the quality of the animation seems to suffer the most, making it appear more as advected noise than an actual collision with generated vortices. Again it might be possible to further *design* the collision by creating an additional vortex point at the sphere's center, which of course would introduce more computation.
Post-collision, the animation does not create the same wake behind the sphere as the offline animation, meaning the animation does not move in behind the sphere in the same manner, making a comparatively wider stream. The same vortex point at the sphere center suggested above might be able to alleviate this issue as well.
Overall it has proven quite difficult to achieve the exact same visual animation results as the offline rendered version. The animation produced here simply does not seem to be as physics driven. There is also a lack of seemingly physics driven detail in the animation, the animation seems less advected by pressure and more by a constant force, which is also the case. These issues might be possible to alleviate with more custom animation curve driven advection and vortex points.
The rendering of the animation compared to the offline renderer does not obtain the same detailed rendering. The particle rendering obscures some of the details which are visible in a pixel by pixel offline rendering. More detail can be added to the rendering by adding more particles and reducing their size, thereby allowing for more possible animations to stand out. Adding more layers to the deep opacity render, and possibly increasing the size of the rendertexture used for the opacity map, can add more detail to the rendering as well, since it will produce more self shadowing in the volume.

Still, evaluating the animation and rendering by itself, without comparing it to the offline renderer, provides a relatively plausible animation and a believable rendering.

10 Conclusion

This thesis has presented a method for creating plausible animation and rendering of passively advected fluids while keeping the hardware requirements relatively low.
The method includes methods for designing different animations using a combination of vortex points, animation curves and Perlin noise. Furthermore it includes a method for making the fluid crudely interact with arbitrary moving or static geometry. All while keeping the animation divergence free through the use of the curl identity.
While the implementation could be optimized, it has been demonstrated that the solution performs at acceptable performance rates, while keeping a relatively plausible animation and rendering. This makes the method ideal for inclusion in real-time applications.

11 Future Work

In the future, different improvements and extensions to the method could be investigated.
Quality wise the particle rendering and animation of the wispy smoke in particular could be improved, possibly by adding deep opacity maps to the rendering, and tweaking the animation. Sorting of the particles could also aid in improving the visuals when alpha-blending. Using post-process driven techniques, such as motion blur, the coherence of the smoke rendering could for example be improved. Overall, alternate rendering techniques, such as ray-marching[29] or screen-space based methods, could also be explored to either increase quality or performance.
The curl animation tools could be extended by introducing new techniques for creating other animations than the vortex points and the animation curves. Furthermore the animation of vortex points could be made more dynamic by driving the points with the Curl-noise technique itself, also making it possible to respect boundaries in animation.
A technique for smoothly changing the noise in a way that would not introduce swaying in the animation is an important step forward for making more dynamic fluid advection. In this regard other smooth noise techniques than the Perlin noise could be investigated.
Performance wise, it might be feasible to offload some work to the CPU, for example by building the boundary map on the CPU. Furthermore newer hardware features such as compute shaders or transform feedback[30] of newer GPU's, could be used to retrieve the new vertex positions back from the GPU before the render stage. This could ease some of the extensive managing of skinning with each needed render. It is questionable though what impact it might have on performance.
Lastly, investigating other phenomena's, such as fire or snow storms, applicability to the method might be valuable. Additionally the method might be able to simulate some parts of actively advected phenomena such as streams or waterfalls.

References

[1] J. Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, volume 18. Citeseer, 2003.

[2] N. Foster and D. Metaxas. Modeling the motion of a hot, turbulent gas. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, page 188. ACM Press/Addison-Wesley Publishing Co., 1997.

[3] M. Harris. Fast fluid dynamics simulation on the GPU. *GPU Gems*, 1:637–665, 2004.

[4] J.M. Cohen, S. Tariq, and S. Green. Interactive fluid-particle simulation using translating Eulerian grids. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 15–22. ACM, 2010.

[5] K. Crane, I. Llamas, and S. Tariq. Real-time simulation and rendering of 3d fluids. *GPU Gems*, 3:633–675, 2007.

[6] J. Krüger and R. Westermann. GPU simulation and rendering of volumetric effects for computer games and virtual environments. In *Computer Graphics Forum*, volume 24, pages 685–693. Wiley Online Library, 2005.

[7] JJ Monaghan. SMOOTHED PARTICLE HYDRODYNAMICS. *Annu. Rev. Astron. Astrophys*, 543:74, 1992.

[8] M. Müller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, page 159. Eurographics Association, 2003.

[9] S. Green. Cuda particles. *NVIDIA Whitepaper, November*, 2007.

[10] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on GPUs. In *Computer Graphics International*, pages 63–70, 2007.

[11] W.J. van der Laan, S. Green, and M. Sainz. Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 91–98. ACM, 2009.

[12] R. Bridson, J. Houriham, and M. Nordenstam. Curl-noise for procedural fluid flow. *ACM Transactions on Graphics (TOG)*, 26(3):46, 2007.

[13] K. Perlin. Improving noise. *ACM Transactions on Graphics (TOG)*, 21(3):681–682, 2002.

[14] Matt Swoboda. a thoroughly modern particle system. http://directtovideo.wordpress.com/2009/10/06/a-thoroughly-modern-particle-system/, November 2010.

[15] L. Latta. Building a million particle system. In *Game Developers Conference*, volume 2004. Citeseer, 2004.

[16] Unity Technologies. UNITY: Web Player Hardware Statistics: 2010 Q2. http://unity3d.com/webplayer/hwstats/pages/web-2010Q2-shadergen.html, November 2010.

[17] M. Olano. Modified noise for evaluation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 105–110. ACM, 2005.

[18] D. Smith. Quick Noise for GPUs. *ShaderX*, 7, 2009.

[19] S. Green. Implementing improved perlin noise. *GPU Gems*, 2, 2005.

[20] Inigo Quilez. Shader Toy. http://www.iquilezles.org/apps/shadertoy/, October 2010.

[21] J.C. Hart. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12:527–545, 1996.

[22] E. Eisemann and X. Décoret. Single-pass gpu solid voxelization for real-time applications. In *Proceedings of graphics interface 2008*, pages 73–80. Canadian Information Processing Society, 2008.

[23] I. Llamas. Real-time voxelization of triangle meshes on the GPU. In *ACM SIGGRAPH 2007 sketches*, page 18. ACM, 2007.

[24] Evaggelia-Aggeliki Karabassi, Georgios Papaioannou, and Theoharis Theoharis. A fast depth-buffer-based voxelization algorithm. *journal of graphics, gpu, and game tools*, 4(4):5–10, 1999.

[25] C. Yuksel and J. Keyser. Deep opacity maps. In *Computer Graphics Forum*, volume 27, pages 675–680. Wiley Online Library, 2008.

[26] Unity Technologies. UNITY: Game Development Tool. http://unity3d.com/, January 2009.

[27] J. Park, Y. Seol, F. Cordier, and J. Noh. A Smoke Visualization Model for Capturing Surface-Like Features. In *Computer Graphics Forum*. Wiley Online Library, 2010.

[28] R. Fedkiw, J. Stam, and H.W. Jensen. Visual simulation of smoke. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 15–22. ACM, 2001.

[29] K. Zhou, Z. Ren, S. Lin, H. Bao, B. Guo, and H.Y. Shum. Real-time smoke rendering using compensated ray marching. In *ACM SIGGRAPH 2008 papers*, pages 1–12. ACM, 2008.

[30] Khronos Group. http://www.opengl.org/registry/specs/EXT/transform_feedback.txt, November 2010.