# Interactive Visualization and Exploration of SPH Data

John Biddiscombe[1], David Graham[2], Pierre Maruzewski[3]

[1] Swiss National Supercomputing Centre (CSCS), Manno, Switzerland
[2] School of Mathematics and Statistics, University of Plymouth, UK
[3] Laboratory for Hydraulic Machines, Ecole Polytechnique Fédérale Lausanne, Switzerland

## Abstract

Advances in graphics hardware in recent years have led not only to a huge growth in the speed at which 3D data can be rendered, but also to a marked change in the way different data types can be displayed. In particular, *point based* rendering techniques have benefited from the advent of vertex and fragment shaders on the GPU which allow simple point primitives to be displayed not just as dots, but rather as complex entities in their own right.

We present a simple way of displaying arbitrary 2D slices through 3D SPH data by evaluating the SPH kernel on the GPU and accumulating the contributions from individual particles intersecting a slice plane into a texture. The resulting textured plane can then be displayed alongside the particle based data. Combining 2D slices and 3D views in an interactive way improves perception of the underlying physics and speeds up the development cycle of simulation code.

In addition to rendering particles themselves, we can improve visualization by generating particle trails to show motion history, glyphs to show vector fields, transparency to enhance or diminish areas of high/low interest and multiple views of the same or different data for comparative visualization. We combine these techniques with interactive control or arbitrary scalar parameters and animation through time to produce a feature rich environment for exploration of SPH data.

## 1. Introduction

The rapid development of the power of GPUs in desktop computers in the last few years has produced an explosion in the number of triangles that can be rendered per second. In addition, advances in the architecture of graphics processors have led also to programmable shaders which allow a far more flexible approach to the generation of images. Instead of representing particles as collections of triangles, it is possible to render them directly to screen by supplying only a position, radius and colour (or other combination of scalar parameters of interest). The shader which resides as a small program on the GPU can evaluate a sphere function and perform an intersection between an eye ray through the screen with the sphere (for each pixel) to produce an exact image of the particle data. Observation of this capability leads naturally to the idea of evaluating more sophisticated functions such as the SPH kernel itself on the GPU and producing an even more useful image. In fact evaluating the kernel on the GPU has been done many times [3,4,5], but in these cases, the SPH field equations are evaluated for the purpose of animating the particles themselves rather than displaying the results of the simulation. In [6], the kernel is evaluated and the particles are re-used directly from the GPU to generate the surface using point splatting.

The work of Sigg et al [2] provides the starting point for our implementation of a particle renderer and for an SPH slicing algorithm on the GPU. Their key development is to represent a quadric (sphere, cylinder, cone, ellipsoid or even parabolic surface) as a 4x4 matrix that can be combined with the usual transform and lighting pipeline of graphics hardware. Using their approach we have developed a particle rendering tool which produces high quality spheres at interactive frame rates. Since the particles are already passed to the GPU for display, we wish to extend the capabilities by performing a second pass of the renderer which can produce slices through the data and display the field as a continuum rather than as discrete points.

The SPH kernel is spherically symmetric – having one parameter of interest (from the point of view of visualization) – which is the cut-off radius, we may therefore represent our particles as spheres and render them with two clip planes positioned equidistant from the slice plane – one in front, the other behind with the distance from the slice to clip plane chosen to be exactly one radius of the kernel used. Any particle further than this distance on either side is automatically removed and no evaluation of the kernel is required. **Figure 1** shows a simple schematic of the regions of interest of the particles between clip planes.

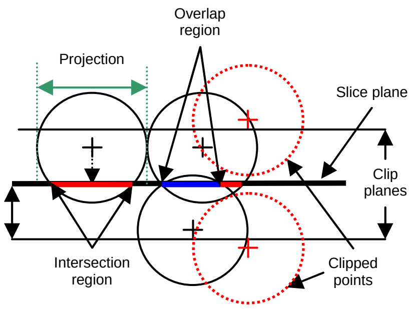

**Figure 1** : *Particles rendered between two clip planes.*

Since we are only interested in the field strength on the plane itself, only portions which overlap the slice plane need be evaluated.

In the following section we describe the operation of the vertex and fragment shaders which compute these regions and colour them appropriately.

## 2. Vertex and Fragment Shaders

### *2.1. Vertex Shader*

The computation of the kernel must be avoided whenever possible and we therefore wish to limit all calculations to regions where the particle projects onto the slice plane as shown in **Figure 1**. This calculation is performed in the vertex shader - the calculation of the bounding box projection for an arbitrary quadric is given in [2], but we can make the following assumptions

- The projection onto the slice is orthographic and therefore we need not consider any perspective correction
- The particles are spherical and the bounding box is simply the scaled transformation of the particle radius from world coordinates to the pixel coordinates of the slice.
- The slice has isotropic rectilinear pixels and we can ignore ellipsoidal terms.

This considerably simplifies the expression for the necessary pointsize *R* and reduces it to

$$R = \frac{2r_k l_w}{l_s w_w}$$

Where $r_k$ is the kernel radius, $l_w$ is the distance in world space along the primary axis of the slice, $l_s$ is the length in pixels of the slice, and $w_w$ is the homogenous clip coordinate of the transformed point. The transformation used for the points is given by placing the eye *e*, and viewpoint *v*, at

$$\vec{v} = \{c_x, c_y, c_z\}\text{, } e = v - \vec{n}_s r_k$$

Where *c* represents the slice centre and $\vec{n}_s$ is the slice plane normal. All that remains is to specify the front and rear clip planes at {0, $2r_k$} and the 'up' vector is along the second slice axis. The vertex shader computes the transformed coordinate and any points lying outside the clip region are removed before being passed to the fragment shader.

To save time in the fragment shader we compute the (non-varying) distance $d_p = w_z / w_w$ from the point to the plane using the coordinate $w_z$ in clipping space.

### 2.2. Fragment Shader

The fragment shader operates on a square region of pixels given by the pointsize computed in the vertex shader. For each pixel we must compute the distance from the point centre and plug this into the kernel if it lies inside a unit circle inscribed within the box – since the box has been generated using the kernel smoothing radius, we know that the incircle exactly fits the smoothing radius. Off axis particles have a smaller intersection with the slice plane and we must only consider pixels with distance $d_i$ within a smaller ring within $r_p$

$$d_i = |x_i - x|, \qquad {r_p}^2 = 1 - {d_p}^2$$

For $d_i > r_p$ the fragment is outside the intersection ring and is discarded. For $d_i <= r_p$ we must use the true distance $d_k$ to the kernel centre given by ${d_k}^2 = {d_i}^2 + {d_p}^2$

To correctly display the results of the kernel evaluation, we must sum the values from overlapping kernels. This is done by rendering the scene as described into an OpenGL FrameBuffer object using a floating point texture and enabling blend mode to sum the incoming pixels. We have currently implemented cubic spline and cusp kernels, others can easily be added.

The result of the fragment shader is a correctly smoothed representation of the particles along the slice. Compare the images of **Figure 2** which show the same data rendered using a triangulation of the points and the output of the SPH slice render. The flaws in the triangulated version are obvious and the smoothed one is clearly superior.

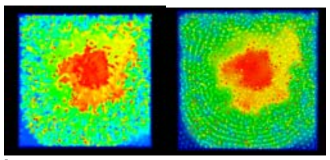

**Figure 2** : *(a) Continuous field plot using Delaunay triangulation of the raw points. (b) Particles overlayed on a field plot produced by SPH smoothing.*

## 3. GUI Implementation

The output of the shaders is a correctly smoothed representation of the particles along the slice. We have embedded the tool within the *ParaView* [1] and *sparticles* GUIs using a *PlaneWidget* to interactively allow the user to freely move the slice plane through the data at any orientation. The output of the shader is stored in a texture which is mapped onto the plane and displayed interactively with the data. This gives the user the ability to view any slice through the data. **Figure 3**(b) shows an example of a large injection dataset (1 million points) rendered using transparency to give a volumetric effect. For volumetric plots, the opacity, colour and even radius of particles may be controlled on a per particle basis using any sets of scalar variables. The plot has been enhanced with a slice plane (the interactive handles are also shown) which in this case reveals features away from the primary jet.

We have also implemented *particle trails* which display the pathlines of individual particles. The trail length is configurable and a subset of particles may be used for clarity of presentation. **Figure 3**(a) shows particle pathlines for the intermediate stages of a lid-driven cavity flow simulation (Re=1000). The pathlines are not quite closed, indicating that the solution has not yet reached a steady-state. The lack of pathlines in the bottom corners is an indication of the very weak recirculation in these areas.

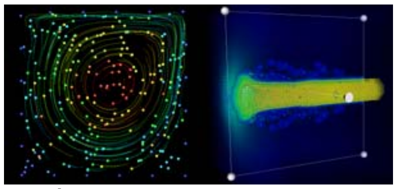
**Figure 3** : *(a) Particles with pathlines (b) Transparent rendering of particles.*

*ParaView* has many useful features in visualising SPH results. For example, comparisons of results using different numbers of particles can be displayed simultaneously. This is useful in determining whether the higher resolution is necessary. **Figure 4**(a) displays three aspects of the same computation at the instant when a wave impacts against a vertical wall. It shows that high values of pressure, turbulence viscosity and density are found at the moment of impact. **Figure 4**(b) shows particle mixing as a result of paddle motion and wave breaking, displaying results at three different times. Such figures and associated animations can be generated easily in *ParaView*. Many other features such as vector plots, glyphs, streamlines, and contouring can be found within the extensive library of filters available in *ParaView*.

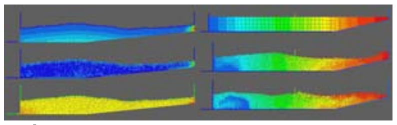
**Figure 4:** *(a) Simultaneous view of (top to bottom) pressure, viscosity and density at wave impact. [Red=high, blue=low]. (b) Particle mixing during breaking of 2s waves (t=0, t=c5s, t=c10s)*

## 4. Conclusions

We have implemented a particle renderer which produces high quality output and enhanced it with capabilities for producing SPH specific plots. We have built this functionality into a custom rendering tool, *sparticles*, and also into a customized version of the *ParaView* visualization package making it not only accessible to many users, but also providing a range of other visualization algorithms and animation tools that allow the SPH results to be combined with other conventional data and visualizations.

## Acknowledgements

The second author acknowledges Dr Jason Hughes at the University of Plymouth for some of the computations and Dr Songdong Shao at the University of Bradford for the original I-SPH code.

## 5. References

[1] A.H. Squillacote, *The ParaView Guide: A Parallel Visualization Application*, Kitware Inc. 2006; www.paraview.org

[2] Ch. Sigg, T. Weyrich, M. Botsch, M. Gross, *GPU-based ray-casting of quadratic surfaces,* Eurographics Symposium on Point-Based Graphics 2006

[3] T. Amada, M. Imura, Y. Yasumuro, Y. Manabe and K. Chihara, *Particle-Based Fluid Simulation on GPU*, ACM Workshop on General-Purpose Computing on Graphics Processors and SIGGRAPH 2004, LA. California, 2004.

[4] Andreas Kolb, Nicolas Cuntz . *Dynamic Particle Coupling for GPU-Based Fluid Simulation.* Proc. 18th Symposium on Simulation Technique, 2005

[5] M. Muller, B. Solenthaler, R. Keiser, M. Gross. *Particle-Based Fluid-Fluid Interaction* SIGGRAPH/Eurographics Symposium on Computer Animation 2005

[6] Müller M., Charypar D., Gross M.: *Particle-based Fluid simulation for interactive applications*. In SIGGRAPH/ Eurographics Symposium on Computer Animation 2003, pp. 154-159.