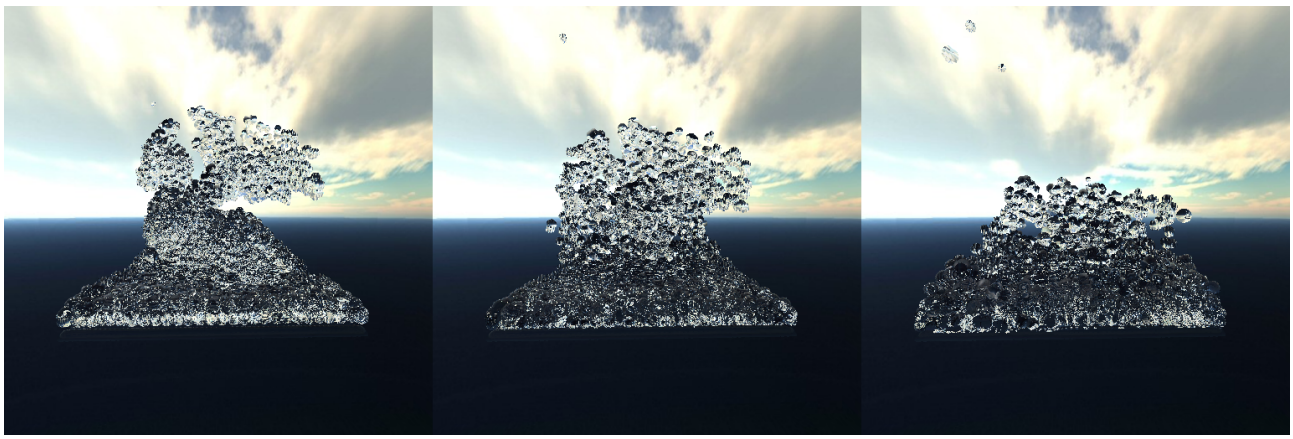
Flow Simulation Controlled by Animated Triangle Meshes

Attila Kárpáti*
Viktória Burkus†
Supervised by: László Szécsi‡

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest / Hungary



Abstract

Fluid behavior in both real-time and production systems must often be physically plausible while also allowing artistic control, possibly forcing otherwise impossible effects. In this paper we address the use of triangle meshes to influence flow behavior, including shaping, dissolving, morphing, separating, and animating liquid bodies.

We discuss options for the underlying flow simulation, and describe an approach based on smoothed-particle hydrodynamics that allows forcing liquids to take shapes dictated by triangle mesh solid models. We discuss algorithms for translating the constraints to simulation features, and elaborate on issues influencing simulation efficiency. We also describe the approach we used to visualize the particle-based fluid simulation. The method reconstructs the liquid surface using metaballs, constructing lists of relevant metaballs for every pixel in every frame. We evaluate alternative solutions to build these lists.

Keywords: Flow Simulation, SPH, Skeletal Animation, Metaballs

*karpati.attila.a@gmail.com
†burkus.viki@gmail.com
‡szecsi@iit.bme.hu

1 Introduction

In computer graphics it is frequently desirable to simulate volumes made of liquid or gas in real time. In some cases, for artistic effect, or to render fantastic features or imaginary technologies, the desired behavior of the fluid is a combination of some non-physical control element and realistic simulation. The technique discussed in this paper provides a solution that alters the realistic fluid simulation in a way that the fluid is forced to fill out objects. To achieve realistic fluid appearance, we reconstruct the liquid surface with metaballs, using the ray marching algorithm. This technique can be used to present objects or characters containing liquid or formed entirely out of fluid material. We also consider and evaluate acceleration options.

We target effects where the fluid has to, at some point in time, fill the volume of a shape. These shapes are assumed to be represented by polygon meshes, as asset pipelines in real-time systems already work with those, allowing for easy integration. However, the mesh is typically not displayed, because it is only used to manipulate the fluid simulation. We assume the mesh is manifold, representing a proper solid. The mesh is used to create new constraints that force the fluid to fill it out.

The typical scenario of possible application would be that the fluid simulation is realistic and free of artificial

constraints initially, but at some point constraints derived from the mesh are introduced, forcing the simulation to fill the shape with fluid, while maintaining plausible fluid behavior. The next step can be the removal of the added constraint. In this case the behavior of the fluid will again become completely physical, dissolving the assumed shape.

It is also possible to add multiple constraints using several meshes. We may want the fluid to fill out the union of the objects. It is also a possibility that the constraints are replaced, at some point in time, with constraints from another mesh. In this way objects morphing from one shape to another can be implemented.

Another application could be that the constraint is added only to a subset of the elements thereby the chosen elements can be subtracted from the rest. For example if the chosen subset of elements has a different color when visualized then the liquid will behave as if it were separated by color during the simulation.

The biggest benefit from the usage of fluid simulation to present fluid-based objects is that all of advantages of the fluid simulation can be applied. For example, the fluid can be manipulated in a way that is not only realistic and fills out a shape at the same time, but it also interacts with other objects in the scene.

2 Previous Work

2.1 Options of fluid simulation

In general there are two main challenges of real-time and realistic fluid simulation. The behavior of fluids is described by the Navier–Stokes equations which are complicated and computationally expensive to evaluate in complex scenes. In order to perform the simulation on computers numerically, the problems must be discretized. However, discretization introduces errors that may give rise to artifacts in the simulation.

We excluded fluid simulation without discretization because it is only applicable to special cases where solutions can be obtained analytically. There are two main approaches to discretize the Navier–Stokes equations that define the relation of velocity, mass density, pressure, viscosity, and external force density. The first set of approaches is based on space partitioning. These kind of methods subdivide the space of the simulation and so that the volume is discretized to geometric elements. If the masses in the elements are known, then the mass density can be expressed from the mass and the volume of the element. If the mass density is known in every element, then the pressure can be expressed, too. Thereby the velocity can be computed. In *finite element methods* the subdivision must follow the geometric boundaries (e.g. the solid walls in between which the liquid flows). As the subdivision process is typically expensive, for efficient computation, the boundaries should not be changed or at least not be changed frequently. Grid-based methods do not adapt the subdivision to changing boundaries, but they have restricted resolution.

Therefore, in complex and dynamic scenes the other set of methods using particle based systems are more efficient. The particle based fluid simulations represent the fluid as a set of particles. The discretized quantity is the mass, so every particle has a constant mass. The mass density is computable from the constant mass and the positions of the particles. With the mass density and a chosen desired mass density, the pressure can be expressed, thus the velocity can be computed in this case, too.

2.2 Smoothed-particle hydrodynamics

Smoothed-particle hydrodynamics represents the fluid as a set of particles. The mass of a particle is constant. The mass density at a particle can be calculated from the distance of the near particles and the constant mass. The density of the fluid can be set as a desired rest mass density, or desired volume, or desired distance between the particles. These three quantities can be expressed from one another with the help of the constant mass. We used the desired mass density. The pressure at a particle can be determined using the ideal gas law. The count of mol, the universal gas constant and the temperature can be replaced by one stiffness constant, therefore stiffness is a constant property of the material of the fluid. The pressure at a particle can be calculated from this stiffness constant, the distances of the nearby particles, and the difference of the mass density and rest mass density. The *pressure force* can be computed from the distances and pressures of nearby particles. The *viscosity force* can be expressed from the velocity, distance, and mass density of nearby particles and the viscosity coefficient. The viscosity coefficient is also a constant property of the material. The *gravitational force* at a particle can be expressed from the mass density and gravitational acceleration. A *surface tension* force can be used for smoother fluid surface. It can be approximated from the mass density and spatial locations of the near particles and a surface tension constant. The sum of forces and the constant mass express the acceleration at a particle. With a chosen time step and with the acceleration, the velocity and the displacement can be computed.

Throughout the computations, local values are reconstructed from discrete particle features using special smoothing kernels. The choice of the smoothing kernels influences the accuracy of discretization and the occurrence of artifacts. The basic fluid simulation implemented for this paper is based on the paper by Kelager[9].

2.3 Options for altering fluid simulation

There are many possibilities to alter realistic fluid simulations. One way is to assign transformations to defined parts of the three-dimensional space and when an object enters the defined domain then the transformation is applied to the object. The transformation can be applied only

partially if the whole object is not contained by the domain of the transformation. The transformations and the domains can be organized into more complex structures to implement more detailed manipulations. It is also possible to define energy like functions that penalize not conforming to the transformations so the minimization of the functions can make seamless transition between the domains and can combine intersecting domains. These kinds of methods are called embedded deformations. This technique can be easily used on a particle based systems because the spatial location of a particle is represented by a single point so the defined domain either contains the particle in whole or not. The paper by Sumner et al.[14] present an embedded-transformation-based technique that can be used on particle systems, too. The drawback of using embedded transformations to present fluid objects is that for an arbitrarily chosen object it is hard to define transformations and domains that force the fluid to fill the object out. It is even more complicated to define it in a way that the manipulation would give the effect of realistic fluid behavior.

Foster et al.[4] present methods to manipulate fluid simulation. The paper defines control parameters for clients that helps to easily alter the simulation of the fluid. So the parameters of the fluid are not manipulated directly, rather the control parameters alter the simulations transitively. Like in the previous paper this technique could be used to create fluid based objects but this would be complex and not efficient, because it is hard to adjust the control parameters to force the fluid to fill out the object. This technique uses space-partitioning-based methods to implement the computations.

Wiebe et al.[19] suggest a technique to present a character with liquid skin. Fluid flow is simulated only on the surface of the character. The result is superficially similar to the fluid object effect presented in this paper but the possible applications of the technique are quite different. The liquid skin only decorates the surface of the object but the fluid objects feature fully functional fluid simulation that provide more possibility like complex interaction with other objects in the scene or combinations of multiple fluid based objects.

The paper by Fattal et al.[3] has a very similar objective to the fluid based object but it uses smoke as the material of the presented volume. The desired smoke animation is achieved by adding two more constraints to the ordinary smoke simulation. A driving force term that causes the fluid to carry the smoke towards the target, and a smoke gathering term that prevents the smoke from diffusing too much. These terms are calculated from a starting and a desired spatial density. The calculations are implemented with space partitioning based methods, while our fluid objects are simulated using a particle based system. So in the second case the evaluation is only performed at the particle positions, eliminating the need for gathering a term. Their driving force term and the control particles presented in this paper are very similar, as the filling of objects in this paper is accomplished using control particles placed inside the given mesh. These control particles make artificially low pressure and evaluation positions that force the nearby fluid particles to equalize the differently pressurized areas. This could be considered as a desired spatial density. Consequently, the basic theoretical concepts in the two papers are very similar, but the difference in the fluid representation implies very different field of applications and options for implementations. Our proposed technique represents the fluid as a particle system and therefore our method is preferable in complex and dynamic scenes as discussed in section 2.1. The paper by Hong et al.[6] presents a technique that also forces the fluid to fill out predefined objects. The object is defined as a potential field that can be produced in various ways. The paper uses a space partitioning based Navier-Stokes solver to simulate the fluid. The benefit of the presented technique is that this solver can be used to alter the flow of the fluid according to the desired potential field. But our method has the advantage of representing the fluid as a set of particles which implies the possibility to replace the potential field with control particles. The animation or placement of these control particles is significantly more simple and computationally less expensive than modifying a potential field.

2.4 Metaballs

Metaballs by Blinn[1] and Nishimura[12] are implicit surfaces that are widely used to display simulation results or smooth objects. Each metaball contributes to an overall density according to a radial density function. An isosurface of the overall density can be used to define a smooth fluid surface.

The surface can be visualized by different methods. The *ray casting* method by Nishita et al.[13] produces high-quality smooth images, however, the downside is that the ray-surface intersection test also has a high computational cost. Typically, *ray marching* is applied, scanning the ray linearly for an approximate intersection. The articles by Kanamori et al.[8] and Szécsi and Illés[15] introduce faster ray casting solutions.

Another option is the Marching Cubes algorithm by Lorensen et al.[11]. The algorithm creates triangle models of constant density surfaces. The quality depends on the resolution of the grid, with a low-resolution grid the algorithm is fast, but causes artifacts to appear on the surface. With a high-resolution grid, we can get high-quality results but with extremely high computational and memory costs.

The third option is to use a screen-space filtering by Wladimir J. van der Laan et al.[16], which has real-time performance with a configurable speed-quality trade-off and smoothes the surface to prevent from looking jellylike but not useful for close-ups.

We selected a ray marching method and implemented it, but with a filtered list of metaballs. *Isosurface ray march-*

ing can be considered a special case of *ray casting*. Rays are cast from the position of the eye through every pixel of the output image, and the image is rendered by finding intersection points between rays and the surface.

To determine the intersection between the ray and the surfaces, isosurface ray marching starts at the eye position and moves this point along the ray direction with a predetermined increment. At each step the algorithm checks whether the point is included in the object. If so, an approximate intersection has been found; otherwise, the algorithm moves forward until it reaches the maximum number of steps. It is possible to refine the point of intersection by iterative root finding methods. In any case, the containment test has to be evaluated at every step, and a similar process is required to obtain surface characteristics for shading at an intersection point. Therefore, in case of metaball geometries, performance depends on how many particles must be considered in these computations. Our contribution is an evaluation of some options to filter the metaballs that may contribute to a ray using rasterization hardware. The impact of these solutions manifests when using extremely high particle counts. We target a method that produces high-quality smooth images, but at a lower computational cost than the brute force method.

2.4.1 Visualization and acceleration schemes

The first particle-based implicit surface was introduced in J. F. Blinn's article[1]. In this article they describe that quantum mechanics represents the electron in an atom as a density function of the spatial location. This function can be represented by summing the contribution from each atom separately. The disadvantage is that the calculation of the isosurface is expensive because every blob must be considered when calculating. The metaball model we use is described in the article by Szécsi and Illés[15].

A lot of papers describe how to render metaballs using the GPU. One of them is Loop and Blinn's[10] method which is fast when the count of metaballs is small, however, not really good for a large number of metaballs. Iwasaki et al.[7] describe a new method to display the surface of a particle simulation, but this approach is not suitable when minor details are required. Van Kooten et al.[17] describe another method for rendering metaballs on the GPU. The particles are spread evenly on the surface, but the approach is not good at visualizing small details.

Without any proper implementation of acceleration options, the algorithm always takes into consideration all particles during the ray–isosurface test in each frame. To avoid this, we have considered various options. A common feature of the methods detailed below is that they capture global information of a 3D scene on a per-pixel basis, and the results are stored in buffers. They are helpful to render complex effects such as order independent transparency, volume rendering, trimming, or collision detection and a lot more. In our case the per-pixel arrays are helpful when the ray marching algorithm evaluates the containment test function, since it is sufficient to consider only the particles stored in the result buffer.

The A-buffer by Carpenter[2], a descendant of the Z-buffer (also called anti-aliased, area-averaged or accumulation buffer) was the first method for capturing all fragments per pixel in a frame and to resolve visibility among a collection of transparent, opaque, and intersecting objects. The result of this algorithm is a buffer with the size of the final image. Each position in this buffer corresponds to a pixel. An element of the buffer contains either a color, if that pixel has one fragment, or a pointer to another list, called a fragment list, with all fragments generated to that pixel. Multiple versions have been released since. Most of them use buffers with limited storage.

In contrast the S-buffer by Vasilakis et al.[18] does not rely on linked-lists or fixed-array structures, but uses two passes. We need the additional rendering passes to determine the sizes of the buffers. As the result of this algorithm, we get two buffers. The first buffer contains the data we need (in our case the IDs of the particles participating in the simulation), the second one is called node buffer, contains offset information for every vertex, indexing into the first buffer. The first rendering pass is a fragment count pass, in which we obtain per-pixel fragment counts for allocating the exact amount of memory that we shall need. The second step is to compute a prefix sum on the fragment counts. This helps generate the locations of buffer ranges that belong to individual pixels in the node buffer. In the third step, which is also a rendering pass, we fill up the data buffer with the help of the result from the second step.

3 Our proposal in detail

3.1 Driving force

In our proposed solution, similar to the approach described in section 2.3 by Fattal et al.[3], a driving force make the particles fill the mesh. However in our case this force is a special form of the pressure force. The pressure force according to basic smoothed-particle hydrodynamics is calculated depending on the near fluid particles. In contrast the driving force is calculated from special control particles instead of the regular fluid particles. Therefore, every regular fluid particle has two pressure-like forces. One derived from other fluid particles as usual, and one derived from the control particles. The simulation and consequently the evaluations are performed only on the fluid particles so the position of the control particles do not depend on the fluid simulations. The position of the control particles can be constant or can be animated externally, e.g. using skeletal animation. To calculate the driving force the control particles must have a pressure value. Let us call this pressure the *control pressure*. This value is an arbitrary chosen constant. A well-chosen control pressure makes an artificially low-pressured volume that attracts

fluid particles to equalize the pressure difference. However, if the control pressure is too high, then the simulation can become unstable. This is due to forcing a density on the particles that exceeds multiple times the rest density and the parameters of the fluid simulation are defined to be stable within a given pressure range that should not be exceeded even with the added control force. The smoothing kernel of the driving force can be the same as the one used to compute the pressure force, or its support radius may be larger. In this case the control particle has a greater range of interaction and attracts more fluid particles. So, on the one hand, with the increased support radius, the control particles can more easily interact with the fluid particles, but, on the other hand, the structure of the fluid particles will be smoother and less detailed than the structure of the control particles.

3.2 Placement of the control particles

The control particles attract the fluid particles as described in the section 3.1. Therefore, if a volume is filled with control particles and there are fluid particles near enough, then the filling of the volume with fluid particles is done by the driving force during the simulation. The maximum range of the driving force equals to the support radius of the used smoothing kernel. Therefore, the interaction range between fluid and control particles can be changed by the manipulation of the support radius. To ensure the required proximity of fluid and control particles, it is an obvious option to submerge the control particle structure into the fluid at first. Thereafter, when it is filled with fluid, it is safe to move the control particle structure further away from the rest of the fluid. In any case, managing proximity is the responsibility of the application.

The volume to be filled by control particles is the inside of the given control mesh. The distribution of the control particles inside the mesh should be uniform for smooth simulation. The density of the control particles multiplies the effect of the control pressure. Therefore, if the density and the control pressure are too high, instability in the simulation is more likely to occur. The ideal case would be that the mesh is filled with as many control particles as possible and the control pressure is as low as can be while the derived driving force is still able to compensate the external forces of the fluid, like the gravitational force. Of course, too many fluid or control particles result in computationally expensive simulation.

There are multiple ways to fill a mesh with particles. One of the simplest options is to place the control particles on the vertex positions. The benefit of this option is the low performance impact. The disadvantage is the poor reliability because the distance and the density of the control particles depend on the input mesh. If the uniform distribution of the vertices is not provided, the areas with high vertex density will attract more fluid particles. This phenomenon deforms the fluid representation of the mesh. Furthermore, this option places control particles exclusively on the surface of the object. Therefore, the fluid can cover only the surface of the given mesh during the simulation. In our tests, this technique has worked well with carefully chosen meshes as expected.

Another option is to generate random points inside the given mesh with uniform distribution. The number of intersections between a ray and the mesh indicates whether the origin of the ray is inside or outside of the mesh, if the mesh is manifold as required. This method can be used to filter out generated points outside of the mesh by casting random rays from all points. This operation must be applied before the start of the simulation because the evaluation of intersections can be computationally expensive, and, due to the randomness, GPU implementations are not trivial. In our experiments, this process caused a long setup time but the placement had no performance effect during the simulation and the density of the control particles was independent from the resolution of the mesh.

The triangle–ray intersection can be replaced by rendering. If the mesh is rendered in a way that all the depth information is stored, the intersections along the rays originating from the camera can be reconstructed. The resolution of the render is important because for points that project between pixels the depth values must be interpolated from the data stored for neighboring pixels. Furthermore, the stored depth values can be used to generate points without random number generation. The points can be placed along the rays of the render between surface intersections inferred from the stored depth values. For an isotropic point set, the distance between points sampled along the ray must match the distance between the rays and the projection of the render must be orthographic to ensure the parallelism of the rays. The advantage of this option is that it is fast enough to be used in every frame. Of course if the mesh is not deformed, then the transformation of the mesh can be used on the control particle structure as well, without re-generating them in every frame. However, if the mesh is deformed, like in the case of skeletal animation, the procedural animation of the control particles is complex. A simple procedural solution is to generate the control points every frame with this placement option.

3.3 Morton sort

One of the most computationally expensive parts of the simulation is finding the particles that are near enough to affect the simulated particle. The simulation presented in this paper applies Morton sort to find neighboring particles. Morton sort orders points according to their bitwise interleaved coordinates, thus organizing them into a three-dimensional discrete curve. The three-dimensional space is discretized and mapped to a one-dimensional sequence by the Morton curve. The binary coordinate representation has to be adjusted to the bounding box of the simulation. If the bounding box of the simulation is changed, then the Morton curve can be easily re-adjusted. If the Morton index uses the longest possible binary range then the likeli-

hood of index collision can be reduced. A more complex way of handling index collision is not necessary because the result of the search is not affected and the performance impact is minimal. The particles can be ordered in parallel by using odd-even sort according to the calculated index. A given particle position and a search range define a sphere. A minimal bounding box that contains the sphere can be determined easily. If both the bounding box and the Morton curve are axis aligned then the minimum and maximum Morton index inside the bounding box equals to the Morton index belonging to the position of two specific corners of this bounding box. Therefore, it is enough to search the particles in the one-dimensional array between these minimum and maximum Morton indices.

3.4 Surface reconstruction using metaballs

We reconstructed the implicit surface using the metaballs field function introduced by Wyvill [5]. We implemented two different fluid visualizations based on ray-marching. The faster one colors the implicit surface of the metaballs with its normal. This method executes the ray-marching until it intersects the implicit surface first, where the surface normal is evaluated. The other visualization presents the structure of metaballs as if it was made out of water. This requires recursive ray-marching. If an intersection is reached during the ray-marching, then the marching continues in both the refracting and reflecting directions. The recursion ends if the ray escapes the bounding box of the fluid or the depth of the recursion has reached the specified threshold. In both cases, the environment map is evaluated in the direction of the last march step and composited together depending on the Fresnel function of water. Ray-marching can benefit from the acceleration options after reflection or refraction, too, if the current marching position is back-projected to the screen. Furthermore, we implemented the A-buffer and S-buffer accelerations options in order to compare them.

3.4.1 A-buffer

In the first step a billboard is placed at every metaball location. The size of the billboard is chosen to match the range of the metaball. Instead of storing the color and opacity values as usual, the ID of the metaball is stored in the linked list. During ray-marching, when testing for the implicit surface, instead of iterating through all metaballs, the shader only tests the filtered metaballs that are stored in the corresponding linked list.

3.4.2 S-buffer

Similar to the A-buffer a billboard is placed at every metaball location in the same way. The S-buffer does not use linked lists, therefore only the number of billboards are counted per pixel in the first render. The second step is to execute a parallel prefix sum on the result of the previous step. The billboard counts after the prefix sum are the indices for storing the input values of the S-buffer. Therefore, in the last step the billboards are rendered again and the IDs of the billboards are stored according to the prefix summed indices. Ray-marching gains the advantage of the filtered metaball list in this case, too, but the IDs are not stored separately like in the linked lists, because the S-buffer stores the IDs sequentially. The adding of an extra rendering step is compensated by the improved cache coherence of the sequential memory usage.

4 Results

We have implemented our simulation system using the C++ programming language with DirectX 11 API for rendering. The simulation was tested on a Windows PC with 32 GB of RAM and an Intel 4790k processor, using an NVIDIA GTX 1080 Ti graphics card with 11 GB of video RAM.

Table 1 shows the measured frames per second in various setups. There are eight different scenes. The first scene does not contain any control mesh, therefore only the visualization is computed. The next two scenes can be seen in the second row of Figure 1 and in Figure 2. These scenes present a character with and without animation. The first row of Figure 1 shows the fourth scene. It contains a giraffe mesh. The last four scenes use the same drake mesh. The sixth and seventh scene contain 4096 fluid particles, the rest contain 2048. The sixth scene is rendered from double distance. The eighth scene is visualized with double sized metaballs, and consequently the billboard sizes are doubled, too. The figure below the title presents the seventh scene, where the control constraints are suddenly turned off. The first column of the table indicates whether the scene is presented with the gradient shading or with the ray tracing method. The third column presents the depth of the recursion during ray tracing. The fourth column shows that how many binary steps are applied for more accurate intersections. The last three columns show which acceleration method was used. The giraffe mesh was filled with 3400 control particles, the drake with 8000, the character between 2700 and 3000 depending on the animated pose. The resolution of the rendered images are 512x512. In the shown test scenes the gravitational force was turned off for fluid particles that were affected by the control force, to help the fluid particles fill the mesh faster.

5 Conclusions

By varying the control mesh various interesting scenes can be produced with the help of the presented technique. The opaque visualization is real-time in most of the setups, but recursive ray tracing is slow if the recursion is too deep.

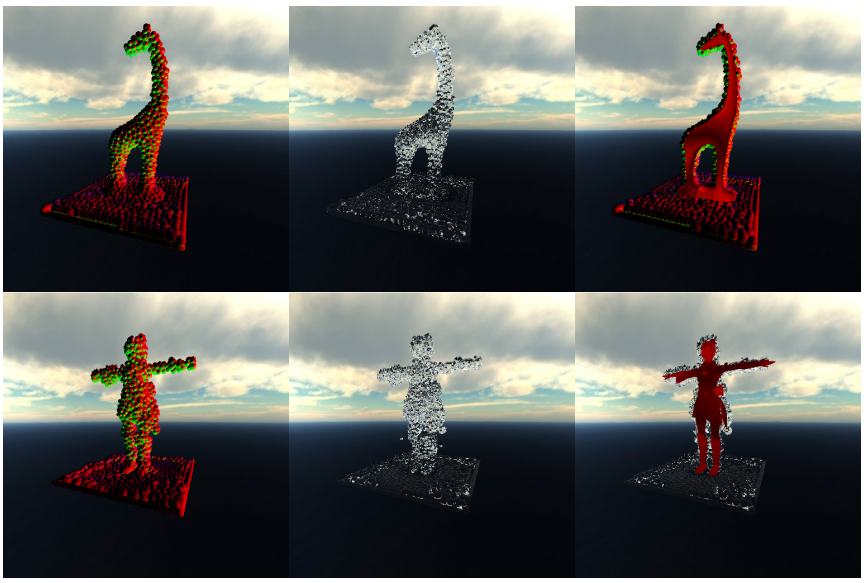


Figure 1: Fluid visualized using gradients as colors, with recursive ray tracing, and with the flat shaded control mesh superimposed.

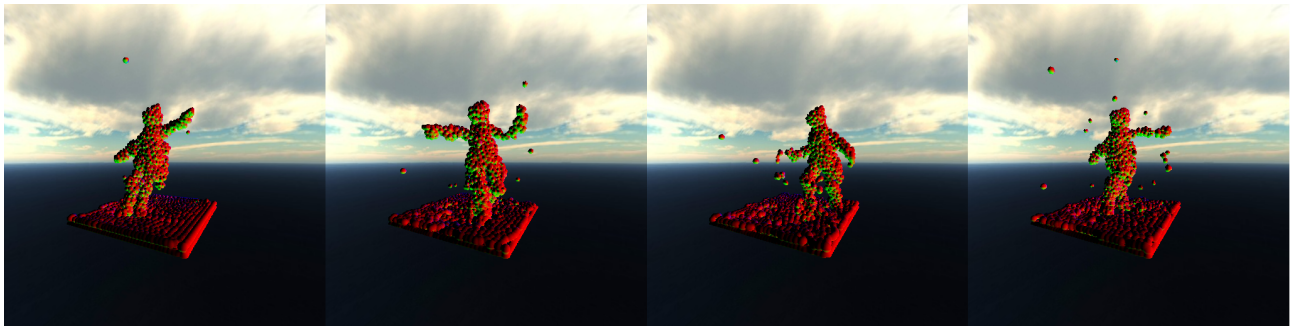


Figure 2: Frames of a character animation sequence using an animated control mesh, with the fluid surface visualized using gradients as colors

The acceleration options proved successful with reasonably small metaball ranges. In case of ray tracing the more significant acceleration is due to the multiple readings of the created per-pixel structures. Based on the measurements visualization takes most of the frame time.

With the high parallel computing capacity of modern video cards it is possible to use particles not only to represent the fluid but also for flow controlling purposes. This paper demonstrates that voxel based approaches to represent spatial densities can be replaced by control particles. These control particles can be placed and applied efficiently enough to be used in real-time applications. Further research could increase the advantages of control particles over voxel based structures. For example, with the direct animation of the control particles the reiteration of the placement step could be avoided. In this case the velocity of the animated control particles could be used as a vector field to improve the tracking ability of the fluid.

5.1 Limitations

Without canceling the external forces conditional on the magnitude of the control force, it was difficult to find a proper support radius and control pressure for every different scene. Therefore, an algorithmic definition of these constants would greatly improve the usability of our technique.

References

[1] James F Blinn. A generalization of algebraic surface drawing. *ACM transactions on graphics (TOG)*, 1(3):235–256, 1982.

[2] Loren Carpenter. The a-buffer, an antialiased hidden surface method. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 103–108, 1984.

Visual	Scene	Rec.	Bin.	Norm	A-Buff	S-Buff
Grad	No Mesh	1	3	10	26	29-30
Grad	Character	1	3	9-10	25-26	28-30
Grad	Anim.Char.	1	3	9-10	25-26	28-30
Grad	Giraffe	1	3	9-10	25-26	28-30
Grad	Drake	1	3	9-10	25-26	28-30
Grad	Drake 4k,2d	1	3	8-9	20	20
Grad	Drake 4k	1	3	4-5	9-10	14-15
Grad	Drake 2b	1	3	9-10	12	9-10
Real	No Mesh	2	2	4	10	14-15
Real	Character	2	2	3-4	9-10	14-15
Real	Anim.Char.	2	2	3-4	8-10	13-15
Real	Giraffe	2	2	3-4	9-10	14-15
Real	Drake	2	2	3-4	9-10	14-15
Real	Drake 4k,2d	2	2	3-4	11-12	14-15
Real	Drake 4k	2	2	1-2	3-4	7-8
Real	Drake 2b	2	2	3-4	4-5	4-5
Real	No Mesh	4	3	2	5	9
Real	Character	4	3	1-2	4-5	8-9
Real	Anim.Char.	4	3	1-2	4-5	7-9
Real	Giraffe	4	3	1-2	4-5	8-9
Real	Drake	4	3	1-2	4-5	8-9
Real	Drake 4k,2d	4	3	1-2	4-5	4-5
Real	Drake 4k	4	3	0-1	0-1	4-5
Real	Drake 2b	4	3	1-2	2	1-2

Table 1: Scene and optimization scheme comparison measured in frames per second

[3] Raanan Fattal and Dani Lischinski. Target-driven smoke animation. In *ACM SIGGRAPH 2004 Papers*, pages 441–448. 2004.

[4] Nick Foster and Dimitris Metaxas. Controlling fluid animation. In *Proceedings Computer Graphics International*, pages 178–188. IEEE, 1997.

[5] Wyvill Geoff and Wyvill Brian. Data structure for soft objects. *The visual computer*, pages 227–234, 1986.

[6] Jeong-mo Hong and Chang-hun Kim. Controlling fluid animation with geometric potential. *Computer Animation and Virtual Worlds*, 15(3-4):147–157, 2004.

[7] Kei Iwasaki, Yoshinori Dobashi, Fujiichi Yoshimoto, and Tomoyuki Nishita. Real-time rendering of point based water surfaces. In *Computer Graphics International Conference*, pages 102–114. Springer, 2006.

[8] Yoshihiro Kanamori, Zoltan Szego, and Tomoyuki Nishita. GPU-based fast ray casting for a large number of metaballs. In *Computer Graphics Forum*, volume 27, pages 351–360. Wiley Online Library, 2008.

[9] Micky Kelager. Lagrangian fluid dynamics using smoothed particle hydrodynamics. *University of Copenhagen: Department of Computer Science*, 2, 2006.

[10] Charles Loop and Jim Blinn. Real-time GPU rendering of piecewise algebraic surfaces. In *ACM SIGGRAPH 2006 Papers*, pages 664–670. 2006.

[11] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM siggraph computer graphics*, 21(4):163–169, 1987.

[12] Hitoshi Nishimura. Object modeling by distribution function and a method of image generation. *Trans Inst Electron Commun Eng Japan*, 68:718, 1985.

[13] Tomoyuki Nishita and Eihachiro Nakamae. A method for displaying metaballs by using bézier clipping. In *Computer Graphics Forum*, volume 13, pages 271–280. Wiley Online Library, 1994.

[14] Robert W Sumner, Johannes Schmid, and Mark Pauly. Embedded deformation for shape manipulation. In *ACM SIGGRAPH 2007 papers*, pages 80–es. 2007.

[15] László Szécsi and Dávid Illés. Real-time metaball ray casting with fragment lists. In *Eurographics (Short Papers)*, pages 93–96, 2012.

[16] Wladimir J van der Laan, Simon Green, and Miguel Sainz. Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 91–98, 2009.

[17] Kees van Kooten, Gino van den Bergen, and Alex Telea. *Point-based visualization of metaballs on a GPU*. University of Groningen, Johann Bernoulli Institute for Mathematics and . . . , 2007.

[18] Andreas Vasilakis and Ioannis Fudos. S-buffer: Sparsity-aware multi-fragment rendering. In *Eurographics (short papers)*, pages 101–104. Citeseer, 2012.

[19] Mark Wiebe and Ben Houston. The tar monster: Creating a character with fluid simulation. In *ACM SIGGRAPH 2004 Sketches*, page 64. 2004.