Eurographics/ ACM SIGGRAPH Symposium on Computer Animation (2007)
D. Metaxas and J. Popovic (Editors)

Screen Space Meshes

Matthias Müller Simon Schirm Stephan Duthaler

AGEIA

Abstract

We present a simple yet powerful approach for the generation and rendering of surfaces defined by the boundary of a three-dimensional point cloud. First, a depth map plus internal and external silhouettes of the surface are generated in screen space. These are used to construct a 2D screen space triangle mesh with a new technique that is derived from Marching Squares. The resulting mesh is transformed back to 3D world space for the computation of occlusions, reflections, refraction, and other shading effects. One of the main applications for screen space meshes is the visualization of Lagrangian, particle-based fluids models. Our new method has several advantages over the full 3D Marching Cubes approach. The algorithm only generates surface where it is visible, view-dependent level of detail comes for free, and interesting visual effects are possible by filtering in screen space.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism Animation and Virtual Reality

1. Introduction

The Marching Cubes method [LC87] is the most popular technique for generating triangle meshes along iso-surfaces of scalar fields. One important application is the reconstruction of the liquid-air interface in fluid simulations. In Eulerian fluid simulations, the liquid surface is often represented as the zero level set of the scalar level set function which is advected by the velocity field of the fluid [FF01, EMF02]. The Marching Cubes technique is then used to construct a triangle mesh along the zero level set. In Lagrangian, particle-based approaches [GM77, MCG03, PTB*03], the liquid surface is typically defined as an iso-surface of a density field which is the superposition of radially symmetric kernel functions of the individual particles [Bli82]. Again, a triangle mesh is generated using the Marching Cubes algorithm.

Although Marching Cubes is certainly the most popular and useful algorithm to generate 3D triangle meshes for iso-surfaces of scalar fields, there are some disadvantages when used for the visualization of surfaces of liquids. First, since the standard approach is camera-independent, many invisible triangles and surface details are generated. Second, the algorithm operates in three dimensions although what is sought is the front 2D surface. This surface is to be found by marching through a 3D data set.

Off-line tracking of the free surface of liquids is a well understood problem with a large body of work. However, the authors are not aware of any methods for rendering the full 3D air-liquid interface that is fast enough for the use in games. Our method closes this gap. It is significantly faster than previous methods. It does, however, only render the front most layer of the surface. This is not a limitation for opaque liquids like milk or oil and in most use cases with transparent fluids, especially in connection with fake refraction shaders, the artifacts are minimal as our examples show. The advantages of the new screen space mesh approach are

- The screen space mesh resolves parts of the surface which are close to the camera with more triangles than distant parts, yielding camera-dependent level of detail.
- Since it operates in two dimensions, a method derived from Marching Squares can be employed which is substantially faster than the 3D Marching Cubes algorithm.
- In contrast to other screen space approaches such as ray-tracing or point splatting, fast standard triangle shading hardware can be used for state-of-the-art forward shading of the surface and occlusion culling since the mesh can easily be transformed back into world space.

SCA 2007, San Diego, California, August 04 - 05, 2007

Figure 1: *Left: Final rendering. Middle: The screen space mesh. Right: Rotated view of the mesh to show its dependence on the viewing direction.*

- With a mesh defined in screen space, the smoothing of depths and silhouettes can easily be separated and controlled individually.

2. Related Work

Our approach is a screen space technique. Many rendering algorithms operate primarily in screen space. The most prominent among these is *raytracing* [Whi80]. For each pixel, a ray is cast into the scene which is then traced along reflected and refracted directions yielding view-dependent level of detail. The array of pixels could be interpreted as a high-resolution regular grid in screen space which is projected back into world space. However, in contrast to screen space meshes, ray tracing cannot directly make use of the triangle rendering pipeline of modern graphics hardware. Also, the resolution must always match the pixel resolution while the resolution of screen space meshes is adjustable to the time budget for surface rendering.

The *projected grid concept* introduced recently by Johanson *et al.* [Joh04] is more closely related to screen space meshes. It is based on earlier work on real-time ocean rendering [HNC02]. While the projected grid is designed to render unbounded water surfaces represented by height fields, screen space meshes can be used to render arbitrary three-dimensional (fluid) surfaces. Because the water surface is considered unbound in [Joh04], a projected *grid* is sufficient. No connectivity is generated in screen space and, thus, no silhouettes can be represented.

Since we present screen space meshes as a method to render the surface of point-sampled volumes, the approach is also related to point splatting methods [ZPvBG01]. The generation of the depth map is similar to a point splatting step while the rest of our algorithm deviates completely from point splatting.

3. Basic Algorithm

We expect as input a set of 3D points $\mathbf{x}_1, \dots \mathbf{x}_N \in \mathbb{R}^3$ (in our case the locations of a set of particles simulated with Smoothed Particles Hydrodynamics), the projection matrix $\mathbf{P} \in \mathbb{R}^{4\times4}$, and a set of parameters as shown in Table 1.

The basic steps of the algorithm are:

1. Setup regular depth map
2. Find internal and external silhouettes
3. Smooth depth values
4. Generate a 2D triangle mesh using our Marching Squares related procedure
5. Smooth silhouettes
6. Transform mesh back into world space
7. Render the 3D triangle mesh

These steps are executed whenever the points change their locations or the camera moves. Steps 3. and 5. are explained in Section 4 as they are extensions to the basic method.

Parameter	Description	Range
h	screen spacing	$1-10$
r	particle size	≥ 1
n_{filter}	filter size for depth smoothing	$0-10$
n_{iters}	silhouette smoothing iterations	$0-10$
$z_{\max}$	depth connection threshold	$> r_z$

Table 1: *Summary of parameters.*

3.1. Depth Map Setup

Let W and H be the width and height of the screen in pixels. One of the input parameters of the method is the screen spacing $h \in \mathbb{R}$ which does not need to be an integer. The spacing defines a regular grid of cell size h with $N_x = \lceil \frac{W}{h} \rceil + 1$ nodes horizontally and $N_y = \lceil \frac{H}{h} \rceil + 1$ nodes vertically. The depth map $\mathbf{Z} \in \mathbb{R}^{N_x \times N_y}$ stores depth values $z_{i,j}$ at each node of the regular grid. It is generated from scratch at the beginning of each frame.

First the depth values $z_{i,j}$ are initialized with ∞. Then, the algorithm iterates through all N particles twice. In the first

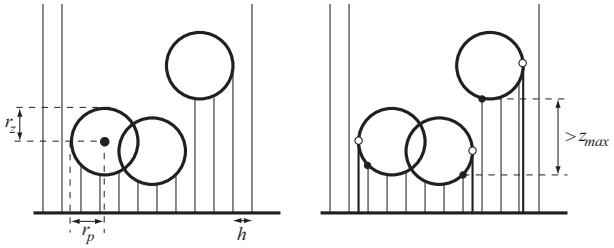

Figure 2: *Left: Side view of depth map generated by three particles. Only front most hits are considered. Large z-differences of adjacent depth values indicate inner and outer silhouettes. Right: Between adjacent nodes at most one additional node (white dot) is stored to indicate the silhouette.*

phase, the depth values are set. In the second phase, additional depth values are generated where silhouettes cut the grid (see Fig. 2). In both phases, the coordinates and radius of each particle have to be transferred from world to screen space. This is done as follows:

Let $\mathbf{x} = [x, y, z, 1]^T$ be the homogenous coordinates of the particle considered. These coordinates are transformed using the projection matrix $\mathbf{P}$ to get

$$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} = \mathbf{P} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{1}$$

We assume the setup of the projection matrix to be defined as in OpenGL and DirectX. In that case, perspective division (i.e. division of x', y' and z' by w) yields canonical coordinates in the range $-1 \ldots 1$ for all three coordinates. However, we perform perspective division only on x' and y' but not on the depth z' because the depth would get distorted non-linearly by this transformation. We compute the projected coordinates as

$$\begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix} = \begin{bmatrix} W \cdot (\frac{1}{2} + \frac{1}{2} x'/w) \\ H \cdot (\frac{1}{2} + \frac{1}{2} y'/w) \\ z' \end{bmatrix}. \tag{2}$$

This way, we have $x_p \in [0 \ldots W]$, $y_p \in [0 \ldots H]$ while z_p is the non-distorted distance to the camera. Another parameter of our method is the particle size r. For the projected radii we get

$$\begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} = \begin{bmatrix} rW \sqrt{p_{1,1}^2 + p_{1,2}^2 + p_{1,3}^2}/w \\ rH \sqrt{p_{2,1}^2 + p_{2,2}^2 + p_{2,3}^2}/w \\ r \sqrt{p_{3,1}^2 + p_{3,2}^2 + p_{3,3}^2} \end{bmatrix}, \tag{3}$$

where the $p_{i,j}$ are the entries of the projection matrix $\mathbf{P}$. This equation only holds at the center of the screen. For a wide field of view the particles get distorted far from the center, an effect we ignore here since it only affects the shape, not the location of the particle. In OpenGL and DirectX $\sqrt{p_{3,1}^2 + p_{3,2}^2 + p_{3,3}^2} = 1$ so $r_z = r$ and z_p is the distance to the camera. We assume that the aspect ratio of the projection is chosen according to the viewport (i.e. W/H). In this case we have a single projected radius $r_p = r_x = r_y$ in the screen plane and the particles appear as circles rather than ellipses.

In the first phase, for each particle, all depth values at (i, j) with $(ih - x_p)^2 + (jh - y_p)^2 \leq r_p^2$ are updated as

$$z_{i,j} \leftarrow \min\left(z_{i,j},\, z_p - r_z h_{i,j}\right), \tag{4}$$

$$\tag{5}$$

where $h_{i,j} = \sqrt{1 - \frac{(ih - x_p)^2 + (jh - y_p)^2}{r_p^2}}$. Omitting the square root in $h_{i,j}$ produces an upside down parabola instead of a spherical surface. This version is faster to compute and sufficient in most cases. The reader might want to experiment with other kernels too. At the end of the first phase, the depth map of the point cloud is coarsely sampled at the grid nodes. Similar to Marching Squares we insert additional nodes on the grid edges connecting nodes with large depth differences in order to track the silhouette in more detail as described in the next section (see also Fig. 2 right).

3.2. Silhouette Detection

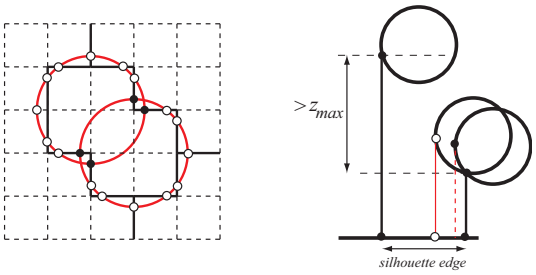

Figure 3: *Left: Top view of the grid. The depth differences at the ends of bold segments are above z_{max}, i.e. they are cut by one or more silhouettes. Right: Side view. The lower two particles generate two different cuts on the edge. Taking the cut (white point) furthest from the end with the smaller depth value (left most in this case) removes this ambiguity.*

For the detection of silhouettes, the algorithm iterates through the particles a second time. In this phase, only grid edges which connect depth values that are further apart then z_{max} are considered (see Fig. 3). We call them *silhouette edges*. The goal of the algorithm is to find one additional node (a *silhouette node*) on each silhouette edge. Each silhouette node is located between the adjacent nodes of its silhouette edge and stores the depth value of the front layer. For each particle p all cuts of the circle at position (x_p, y_p)

and radius r_p in screen space with silhouette edges are computed. Each cut is a silhouette node candidate. Its location is the location of the cut and its depth is the depth z_p of the particle. However, this point is only stored if two conditions hold:

- The depth z_p of the particle is smaller than the average depth of the silhouette edge, i.e. belongs to the front layer. This criterion can be used as an early out to avoid computing the cut point.
- The location of the cut on the silhouette edge is further from the endpoint with the smaller depth value than a potential silhouette node previously stored for this edge. In that case the new node replaces the old one (see Fig. 3 right).

3.3. Mesh Generation

The next step is to generate the vertices and triangles of the screen space mesh. Each grid node with an initialized depth value ($\neq\infty$) generates one vertex at its location with the same depth value. Each silhouette edge with only one initialized adjacent node generates one additional silhouette vertex at the location and with the depth of its silhouette node. This vertex will be on the outer silhouette. Each silhouette edge with both adjacent nodes initialized generates two vertices, both at the location of the silhouette node. One vertex associated with the end point with the smaller depth (the front vertex) gets the depth of the silhouette node. The other vertex (the back vertex) gets a depth value extrapolated from the neighborhood of endpoint with the larger depth value. These two nodes lie on an inner silhouette (see Fig. 4). In this figure linear extrapolation is used to find the depth of the back silhouette vertex which we found to be sufficient.

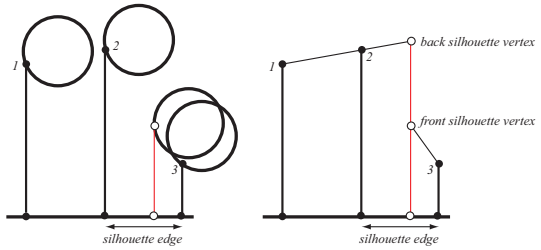

Figure 4: *Side view of the grid. Left: Silhouette node created on a silhouette edge. Right: The mesh vertices and triangles generated for this configuration. Note that two vertices with different depth values are generated for the silhouette node.*

To generate triangles, the mesh generation algorithm visits all grid cells one by one and generates triangles for them. Each of the cell's edges is either a silhouette edge or a regular edge. This leads to 16 cases as shown in Fig. 5. The number of a particular case can be computed by associating each edge with a bit and set the bit to one if the edge is cut, i.e. has a silhouette node.

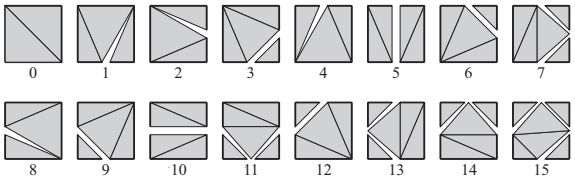

Figure 5: *All the cases for the generation of a 2D triangle mesh from cut edges.*

In Fig. 5 we have opened the cuts on each edge for illustration purposes to distinguish the silhouettes vertex associated with the left and the one associated with the right end node. From the triangles shown, only those are generated for which all three vertices exist. In the case of outer silhouettes only a subset of the triangles need to be added. Special care has to be taken for different groups of cases:

- Case 0 represents inner triangles away from silhouettes. We alternate between the configuration shown and the one which is rotated 90 degrees to get a nicer inner mesh.
- Cases $3,5,6,9,10$ and 12 are the regular situations with the silhouette entering at one edge and leaving at another. No special care has to be taken here.
- Cases $1,2,4$ and 8 are generated by an inner silhouette starting within the cell. The triangulation shown is not unique. To avoid flickering it is important to stick with one arbitrary choice though. The difference of the depth of the two silhouette vertices is bound by $3z_{max}$. Therefore, it is safe to connect the vertices as shown without generating arbitrarily stretched triangles.
- The remaining cases $7,11,13,14$ and 15 are the pathological ones. Have a closer look at case 7. The center triangle is connected to the left two triangles. It is, thus, connected to the same layer as those. The upper right and lower right triangles each belong to different layers which can be arbitrarily far from the layer of the center triangle. Therefore a third vertex needs to be generated for the silhouette node on the right. Its depth needs to be extrapolated from the depths of the four vertices on the left. In case 15 two additional vertices are generated and connected to the layer of the lower left triangle.

3.4. Transformation to World Space and Rendering

At this stage, we have a triangle mesh with the correct connectivity and vertices in screen space, e.g. with coordinates $[x_p, y_p, z_p]^T$. In order to render this mesh and to compute reflections and refractions of the 3D environment, the vertices are projected back into world space while the connectivity is kept fixed. We, therefore, need to invert the transformation given in Eq. (1) and Eq. (2). Let $\mathbf{Q} \in \mathbb{R}^{4\times4}$ be the inverse of the projection matrix, i.e. $\mathbf{Q} = \mathbf{P}^{-1}$. With $\mathbf{Q}$, the world coor-

dinates $[x,y,z]^T$ can be computed via the inverse projection equation

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \mathbf{Q} \begin{bmatrix} (-1+2x_p/W)w \\ (-1+2y_p/H)w \\ z_p \\ w \end{bmatrix}. \quad (6)$$

At this point, we do not have the projective divisor w. However, it can be retrieved (using the last row of the above equation) as

$$w = \frac{1 - q_{4,3} z_p}{q_{4,1}(-1+2x_p/W) + q_{4,2}(-1+2y_p/H) + q_{4,4}} \quad (7)$$

from known quantities only, before the inverse transformation is executed.

After the transformation of the mesh, we generate per-vertex normals in world space. There are several ways to compute vertex normals for a triangle mesh. We use the normalized sum of the normals of adjacent triangles weighted by the adjacent angles. Finally the triangles and normals are sent to the standard graphics pipeline.

4. Extensions

4.1. Depth Smoothing

The procedure described so far produces bumpy depth maps. Fortunately, the depth map can easily be smoothed by applying an appropriate filter. We use a separable binomial filter of user specified half-size n_{filter} (see Fig. 6), both, in i and j direction.

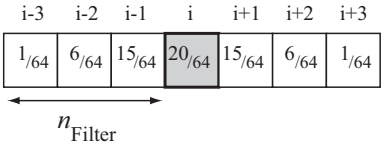

Figure 6: *Separable binomial filter with half-length $n_{filter} = 3$. The center depth value at (i,j) is replaced by the weighted sum of neighboring depth values. The weights used are shown inside the squares.*

In a first pass, the horizontal filter is applied to all values $z_{i,j} \neq \infty$ of the depth map. In a second pass the values $z_{i,j} \neq \infty$ are filtered again using the vertical version of the filter.

When applying the filter, special care has to be taken near silhouettes. When the filter is applied to a certain node, we only consider depth values within the z_{max} range from the depth of the node. However, with this procedure the boundary of the mesh can get tilted towards the camera as Fig. 7 shows. This happens because the symmetry of the filter is lost and the central value is pulled towards the values on the valid side of the filter. This problem can be solved easily. If the value at position $i+k$ is omitted, *both* position $i+k$ and position $i-k$ are ignored in the weighted sum. When the depth of a node is changed, the depths of the silhouette nodes associated with it are changed by the same amount.

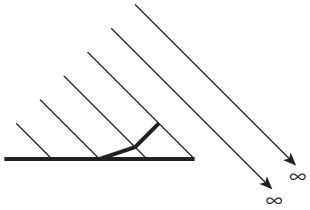

Figure 7: *If certain depth values are not considered for filtering, the mesh gets tilted towards the camera.*

Figure 8: *Top: The standard approach produces bumpy surfaces. Bottom: A flat surface is generated by smoothing the depths in screen space.*

4.2. Silhouette Smoothing

Smoothing of the depth values does not influence the appearance of the silhouettes of the mesh. To smooth the boundary of the mesh we smooth the screen space coordinates $[x_p,y_p]^T$ of the nodes of the mesh before the transformation to world space. We use a very simple but effective iterative scheme. In each iteration, the screen space coordinates of each vertex are replaced by the average of its own coordinates and the coordinates of all adjacent vertices. The regular internal mesh resulting from case 0 in Fig. 5 and its flipped configuration is a fix point of this smoothing procedure. Thus, it only affects the silhouettes as desired. To avoid the opening of internal silhouettes corresponding silhouette

vertices are "glued" together in screen space during this process. The number of iterations n_{iters} is a user parameter. Silhouette smoothing causes shrinking of outer silhouettes. In most cases this effect is desired. In order to cover the area of a puddle the fluid particles have to have a certain size. When small groups of particles separate, the drops appear as rather large blobs. Silhouette smoothing makes the boundary of small sets of connected particles look more realistic (see Fig. 11).

5. Results

All scenes were run on a Intel dual core CPU at 2.6GHz with 2GB of RAM. In most cases the SPH simulation was the bottleneck. The car wash scene (Fig. 1) contains 16K SPH particles. It runs at 20 fps without surface generation. The frame rate drops to 19 fps for screen spacing $h = 6$ and to 18 fps for screen spacing $h = 3$ with a triangle count of 13K and 40K respectively. The dungeon scene (Fig. 11) contains 10K particles. Its frame rate drops from 23 fps to 22 for $h = 6$ with 15K triangles and to 21 fps for $h = 3$ with 60K triangles.

In both the slide scene (Fig. 9) and the wheels scene (Fig. 10) the particle count of 5K is small with respect to the number of triangles generated. Since 5K SPH particles can be simulated very efficiently – 65 fps in the slide scene – the time for the generation for the screen space mesh becomes more significant. For $h = 6$ the frame rate drops to 55 fps with 5 K triangles and to 40 fps for $h = 3$ to 40 fps with a triangle count of 20 K.

6. Conclusions and Future Work

Screen space meshes provide an efficient way to construct and visualize surfaces. We focused on the visualization of particle-based fluids. However, the approach is more general and can be applied to other related visualization problems.

The approach trades speed for certain limitations. The 3D mesh generated from the screen space mesh is only valid for the current camera position. This causes problems with shadow casting. A solution to this problem would be to generate the fluid mesh separately from the positions of the light sources. Also, only the front most surface layer is generated which does not produce visible artifacts in most scenarios as already mentioned.

Most of the algorithm steps, such as the depth map generation or the transformation of vertex positions, are well-suited to be computed on a GPU. We expect a GPU implementation to be much faster than the CPU version we have, although the CPU version already allows the generation of quite complex surfaces in real-time as our results show.

References

[Bli82] BLINN J. F.: A generalization of algebraic surface drawing. *ACM Trans. Graph. 1*, 3 (1982), 235–256.

[EMF02] ENRIGHT D., MARSCHNER S., FEDKIW R.: Animation and rendering of complex water surfaces. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), ACM Press, pp. 736–744.

[FF01] FOSTER N., FEDKIW R.: Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM Press, pp. 23–30.

[GM77] GINGOLD R. A., MONAGHAN J. J.: Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society* (1977).

[HNC02] HINSINGER D., NEYRET F., CANI M.-P.: Interactive animation of ocean waves. In *Proceedings of the ACM SIGGRAPH symposium on Computer animation* (2002), ACM Press, pp. 161–166.

[Joh04] JOHANSON C.: Real-time water rendering - introducing the projected grid concept. *Master of Science Thesis (Lund University)* (2004).

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (1987), ACM Press, pp. 163–169.

[MCG03] MÜLLER M., CHARYPAR D., GROSS M.: Particle-based fluid simulation for interactive applications. *Proceedings of 2003 ACM SIGGRAPH Symposium on Computer Animation* (2003), 154–159.

[PTB*03] PREMOZE S., TASDIZEN T., BIGLER J., LEFOHN A., WHITAKER R. T.: Particle-based simulation of fluids. *Eurographics 22*, 3 (2003), 401–410.

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Commun. ACM 23*, 6 (1980), 343–349.

[ZPvBG01] ZWICKER M., PFISTER H., VAN BAAR J., GROSS M.: Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM Press, pp. 371–378.

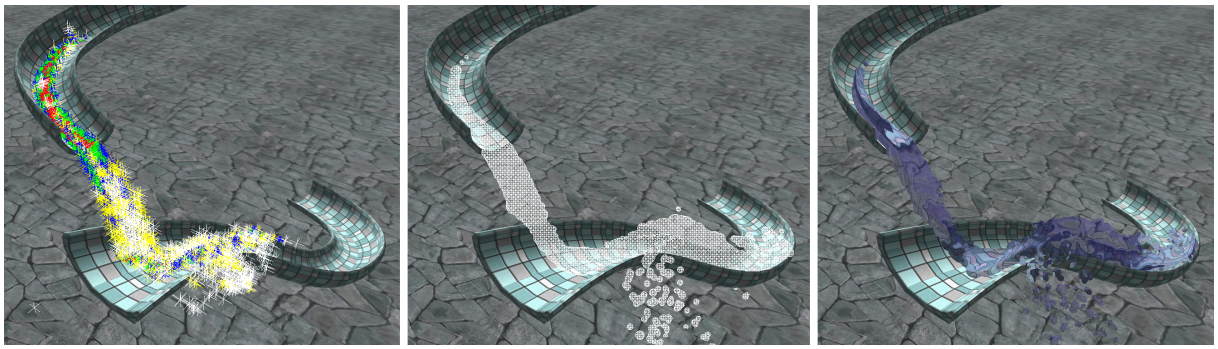

Figure 9: *Left: The original set of particles (their SPH density is color coded). Middle: Screen space mesh. Right: Final rendering.*

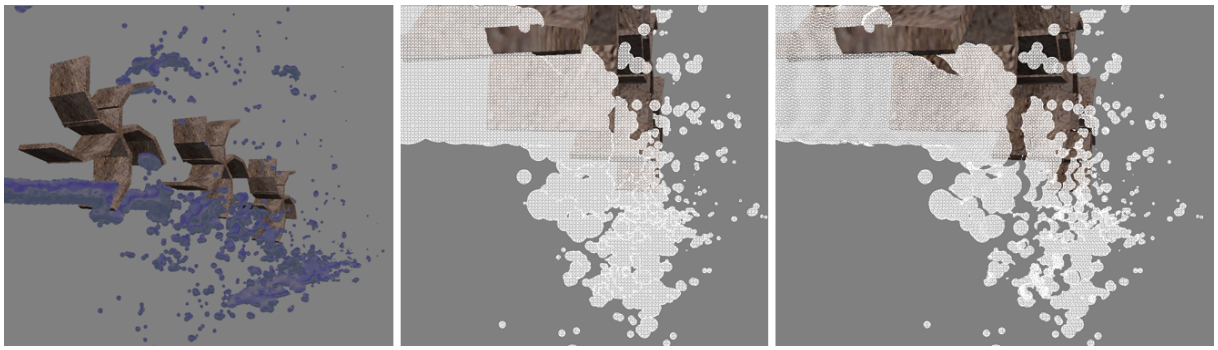

Figure 10: *Left: A scene with a complex 3D liquid boundary. Middle: The screen space mesh with internal silhouettes. Left: A rotation of the frozen mesh reveals that the internal silhouettes are disconnected in the viewing direction as desired.*

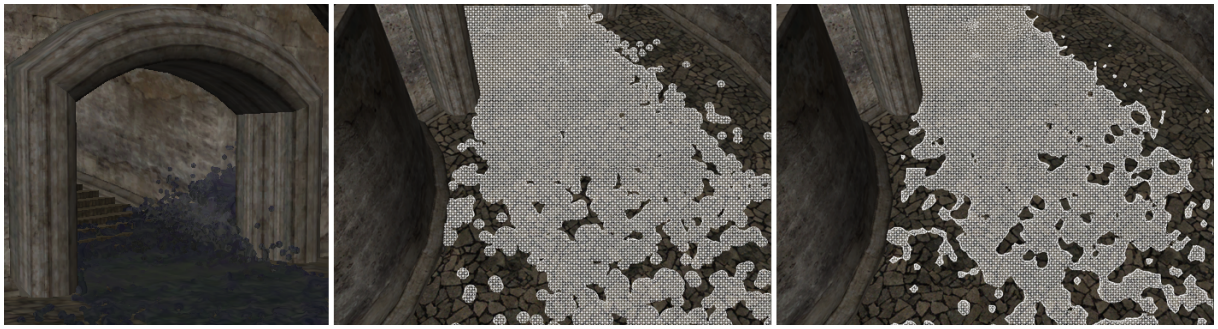

Figure 11: *Left: Flooding a corridor. Middle: Top view of the screen space mesh without silhouette smoothing. Right: Silhouette smoothing generates a surface tension effect.*