GPU-supported bubble and foam rendering

Tamás Huszár *
Supervised by: László Szécsi†

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

Abstract

Several types of foam can be found both in nature and artificial environments; yet it is rare in computer graphics due to its complexity. Modelling foam structure and dynamics by simulating the underlying bubble structure has a high computational cost. To model such a complex phenomena we need to use serious simplifications while maintaining realism and detail.
In this paper we propose a method for rendering dense soap foam in real time. We first build a foam blob — from realistic soap bubbles — which has a solid inner structure. We use a hybrid method based on ray tracing and 2D billboards to render dense foam constructed from hundreds of these blobs. To model foam behaviour and interaction, we present a simple particle based physics simulation approach. While our method is capable of rendering foam featuring a large number of bubbles, it has certain limitations we also discuss in this paper.

Keywords: Bubble rendering, foam rendering, impostors

1 Introduction

Presenting natural phenomena like smoke, fire or fluids in a realistic way is a tough challenge in computer graphics. Even though the equations describing the physics of these phenomena are known, the exact calculations are too complex to perform in real time. Today's graphics hardware requires some intuitive simplifications or artistic input to efficiently present these phenomena in real-time applications like computer games.

Bubble and foam simulation falls into the above category. The structure of dense foam built of soap bubbles exhibits large complexity. While modelling a physically correct soap bubble is an easy task, building complex foam structures from individual bubbles cannot be done in real time on current hardware.

In this paper, we propose a method to render and simulate dense soap bubble foam. We give an intuitive simplification of foam structure which enables us to simulate realistic foam with the speed, detail and quality necessary for real-time applications. After the introduction and previous work, we give a short overview of bubble simulation, discussing the actual techniques used in our method in section 3, including our solution for efficiently modeling multiple connected bubbles. Section 4 introduces the idea of building foam from blobs of soap bubbles. We discuss the structure of these blobs and provide two different methods of storing and rendering blob structure. In section 5, the technique of building actual foam of the blobs is discussed, including a basic physics simulation described in section 6. In the next section we present our result, and provide possible enhancements and future work in further sections, including the conclusion of this paper.

2 Previous Work

Interference phenomena required to understand bubble physics were described by Dias [1]. Later, Glassner gave a thorough overview on several aspects of soap bubble physics, including soap film interference and geometric structure of multiple soap bubbles [3, 4]. Most attempts to model bubble and foam structures are offline methods based on ray tracing [7], and even these offline approaches [6] use simplified reflection model to render the inner dense parts of the foam to maintain reasonable rendering times.

Recent articles present real-time approaches and use the GPU to simulate bubble formations. Sunkel simulates a magnitude of hundreds soap bubbles in real time using simplified reflection and lighting model [8].

3 Realistic rendering of soap bubbles

3.1 Soap film interference

In order to simulate foam, we must first understand the physics of soap bubbles, and provide an efficient way to render soap films and bubble structures. Basically soap bubbles are gas trapped in a thin fluid layer. Because of surface tension and the inside pressure of the contained gas, the surface of a bubble tends to be minimal. This means soap bubbles can be easily modelled as spheres;

*hthomas92@gmail.com
†szecsi@iit.bme.hu

this is a common simplification used by most approaches. The interference phenomenon on bubble surfaces can be understood by examining soap film reflection — light interference caused by reflection on two parallel surfaces. Usual soap films are 1–2000 nm wide and have a refraction index of 1.4. Given these values, the intensity change of the reflected light can be calculated by the following equations [4, 5].

$$ps = \frac{4\pi}{\lambda} nd\cos\vartheta_i$$

$$R_f = 1 - \cos\vartheta_i$$

$$I_r = I_i 4R_f \sin^2 ps$$

The intensity change I_r depends on the incoming intensity I_i, the reflection factor R_f and the phase shift ps. The phase shift can be calculated using the wavelength λ and the incident angle of the light ϑ_i, the index of refraction n and the film width d. The refraction index is calculated using a Fresnel approximation, the film width and the refraction index are constant (we can perturb the film width with random noise to make the bubble more realistic, simulating film thickness changes caused by air pressure variation). By using these simplifications, the intensity change is only dependent on the incident angle and the wavelength, so it can be easily computed or stored in a texture. We used the representative wavelengths of the RGB components, as the achieved quality is acceptable and it is more efficient than calculating the values over a continuous spectrum. Using these equations and simplifications, a soap film shader can be easily constructed.

3.2 Soap bubble geometry

As we saw earlier, a single soap bubble can be approximated by a sphere. However, for modelling bubble structures, we must compute the shared wall film between the bubbles. Based on Glassner's observations, three soap films always meet at 120° angle and the mutual wall is spherical itself [4]. First, considering two intersecting bubbles, we must determine this auxiliary sphere's centre and radius. The easiest approximation would be a simple planar soap film between the two bubbles. This is an acceptable approximation for distant bubble formations but unrealistic when examined closely. In his article, Glassner gives a formula, in which he exploits the aforementioned 120° property. In a general situation without proper physical simulation, when bubbles are spheres of random radii, this rule does not hold. To overcome this we provide a simple but intuitive and visually convincing approximation.

Figure 1 shows the geometry in a 2D slice. We used a simple observation: the tangent T_c is the angle bisector of the angle determined by the intersection of the spheres and the centres ($AMC\angle$). If we extend the bubble model by keeping this rule, but omitting the 120° restriction, we still get acceptable results with reasonable calculation complexity.

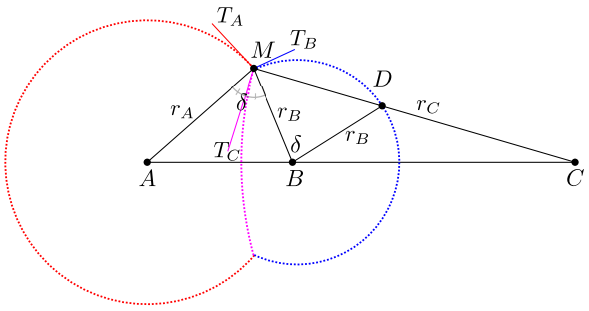

Figure 1: Geometry of bubble walls. Point C is the centre, r_c is the radius of the sphere which forms the common wall of sphere A and B.

First, using the observation above, we realize that the triangles AMC and BDC are similar. This means that $\overline{DC} = r_c r_B / r_A$. As $\overline{MC} = \overline{MD} + \overline{DC}$, we get the following equation (assuming $r_A > r_B$):

$$r_c = \frac{r_A \overline{MD}}{r_A - r_B}$$

The length of MD can be calculated using the cosine rule two times in triangles AMB and MBD.

$$\overline{AB}^2 = r_A^2 + r_B^2 - 2r_A r_B cos\delta$$

$$\overline{MD} = \sqrt{r_B + r_B - 2r_B^2\cos\delta} = r_B\sqrt{2 - 2\frac{r_A^2 + r_B^2 + \overline{AB}^2}{2r_A r_B}}$$

Finally, we can calculate the centre using the law of cosines the third time, in triangle AMC.

$$\overline{AC} = \sqrt{r_A^2 + r_C^2 - 2r_A r_C \cos\frac{d+\pi}{2}}$$

4 Modeling foam using soap bubbles

4.1 Foam structure

While soap foam consists of several soap bubbles, the inner structure of the foam is so complex, that simply modelling it as individual bubbles is not a working solution. Today's real-time methods are capable of rendering hundreds of bubbles, but this is far from the complexity of dense foam. To overcome this, we examined the macrostructure of soap foam. On a large enough scale, below the surface bubbles of dense foams, we see a nearly diffuse and opaque whitish body, hiding the deeper microstructure of the foam. Our idea was to model a foam blob possessing this property. The surface of this blob is built of realistic soap bubbles, and the inside is approximated by an artist-drawn bubble texture representing the inner structure of the foam. This creates the impression of a dense interior with individual transparent bubbles protruding from the blob.

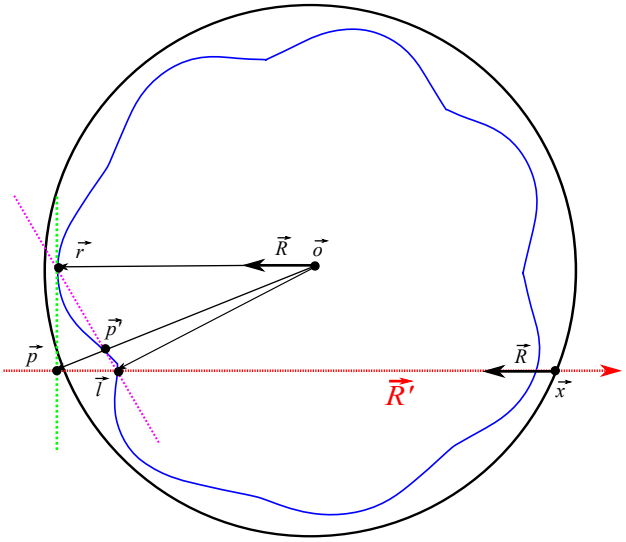

Figure 2: Using the distance impostor technique to model foam blobs. The curve inside the bounding sphere visualizes the distance values stored in the cube map. $\overline{R'}$ is the view ray and $\vec{l}$ is the intersection point provided by the algorithm.

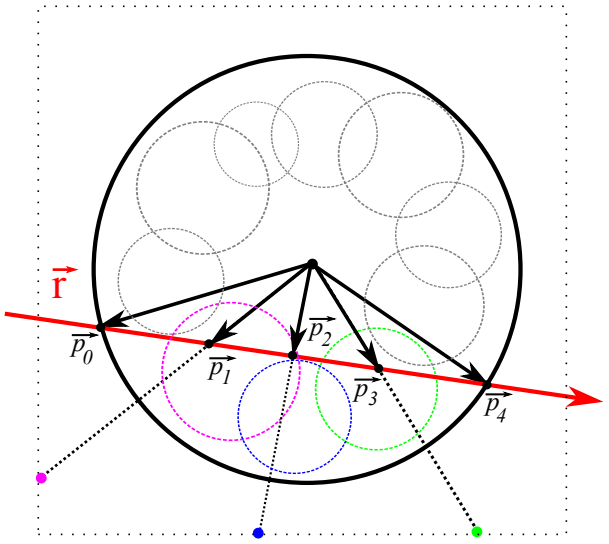

Figure 3: Illustration of the bubble ID technique. The view ray $\vec{r}$ is sampled in points $\vec{P}_0$ - $\vec{P}_4$. The dotted lines show the bubble id stored in the cube map in the designated direction.

The outer bubbles are spheres drawn using the soap film shader, including the walls between neighbouring bubbles. Due to the complex nature of the blobs, classic polygon-based rendering is not a feasible solution. On the other hand, classic ray tracing is too slow, as complex foam can contain thousands of bubbles. We needed a data structure to store bubble data that makes fast and efficient intersection of the blob and view rays possible.

4.2 Blob intersection using a distance impostor

Our first proposed method is based on the distance impostor technique [9]. The original technique is used for calculating position dependent reflections using environment maps. It stores environment geometry in a cube map, and uses it to iteratively calculate surface points intersected by reflection rays. We imagine the foam blob as an entity trapped in a cube, represented by a cube map storing the distance between the blob surface and the centre in every texel. Now finding the intersection of the ray and the blob is the same problem as the one stated above, with one minor difference: our rays come from outside of the cube map, not from the inside.

As seen in Figure 2, R' is the original ray. We reverse the direction of this ray to get $\vec{R}$, and place its origin $\vec{x}$ at the second intersection of the original ray and the blob's bounding sphere. As the new ray points towards the near side of the blob (closer to the origin of R'), the iterative search algorithm will find an intersection $\vec{l}$ on this side. Reading a distance value from the side closer to $\vec{x}$ would give a wrong intersection point. Therefore, running the original algorithm with the reversed ray will give the right result.

The required cube map resolution and iteration count depend on the blob geometry. For a common blob consisting of 64 bubbles, which we used for testing, a resolution of 64×64 and a maximal iteration count of 20 were adequate. When using smaller values, visible artifacts appeared near the intersection of the bubbles.

The cube map itself is generated by rendering the bubbles from the centre of the blob. Using a geometry shader and a render target array, only one rendering pass is required. We render a full screen quad, and calculate the world coordinates of the quad's vertices at the near clipping plane for each render target, using different view matrices (a total 6 cameras covering both directions in all 3 axes). In the fragment shader, we compute the farthest intersection point with each of the bubbles, getting the outside surface geometry of the blob. While the intersection calculation is done by checking all the bubbles in the blob, this is not a performance bottleneck as the cube map has to be generated only once during the program startup.

The disadvantage of the method is that we lose the inner structure of the blob as only the outer shell is stored in the texture. As a side effect, this allows the shape of the outer bubbles to be other than spheres. However, when rendering soap bubbles, this does not grant us an advantage, but makes further calculations — like intersection with the inter-bubble wall — impossible. This recognition motivated our second method. Instead of storing distances of the surface, we store the original bubble data, but we heavily reduce the number of necessary intersection calculations.

4.3 Blob intersection by storing bubble identifiers

Once again, we use a cube map to store blob geometry. But instead of distances, we store an ID of the bubble visible from the outside in the given direction. We also have to store the bubble radii and centres in a separate texture or buffer.

First we calculate the intersection of the ray and the bounding sphere, and then we have to find the bubble it intersects first. This would allow us to use an iterative search similar to the one used in the distance impostor technique, but we found that a simple linear search is adequate. This is because finding any texel that contains the ID of the first intersected sphere is sufficient to get an accurate result. As seen in Figure 3, we divide the section of the ray inside the sphere to a fixed number of segments, and then start reading the values from the cube map in the given directions and calculate the intersection of the ray and the corresponding bubble. The first intersection is the one we are looking for.

The advantage of this method is that in most cases, especially if the bubbles are nearly the same size, we will get the result in the first few iterations. Usually the loop ends in the first iteration when the incident angle is high (the ray goes through the middle of the sphere) and the required iteration count increases as the ray gets further from the centre. Also nothing guarantees that we find the right intersection; in theory we can easily skip the right bubble during the linear search. However experiments showed that when using reasonably sized and evenly distributed bubbles, the results are acceptable. A 128×128 cube texture with 10 iterations produced minor artifacts comparable to the distance impostor technique, and it was also slightly faster.

The actual implementation including the cube map generation is similar to the distance impostor technique. The identifiers are stored in the cube map using the same rendering technique, but instead of the distance, the bubble's id is stored. The bubble identifiers and the correspondent radius and centre values are stored in a buffer located in the video memory.

Storing bubble IDs has another advantage over the distance impostor method: not only the first intersection, but also the intersection with interior walls between bubbles can be computed. When using distance impostors, we only preserve surface geometry, which makes the representation of the precise sub-surface structure impossible. When using the bubble ID technique, we have the exact bubble geometry stored in a buffer. We can use this geometry to calculate inner bubble walls. The linear search algorithm will yield a list of bubble IDs along the ray, if we do not stop it after finding the first intersection. Consecutive bubbles in this list are most likely to form a mutual wall, which can be computed as described in Section 3.2 and intersected with the ray. This is also an approximate solution, as internal bubbles not stored in the ID map are not considered and small bubbles can be skipped by the linear search algorithm. However, with similarly sized bubbles of a soap foam blob this rarely happens, and it does not influence visual quality.

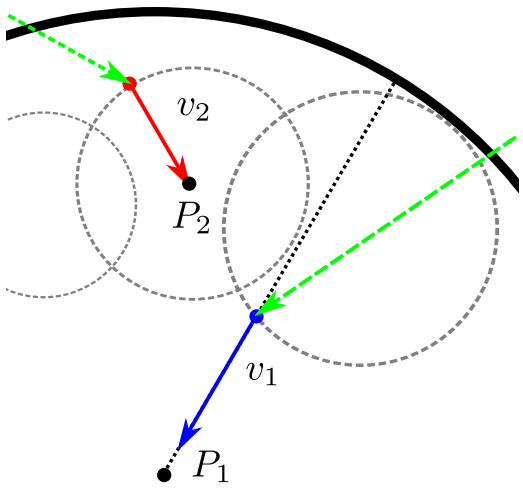

Figure 4: Illustration of the blob normals. The green arrows are light rays. Vector v_1 and v_2 are sampling directions using different calculation methods presented in the paper.

Figure 5: Different foam textures. The texture on the left was used in the final renderings.

4.4 Inner structure and other details

Using any of the techniques presented above, the rendering of the blob is quite straightforward using ray tracing. We calculate the intersection of the blob and the ray coming from the camera. Then we compute the incident angle of the ray and the surface normal in the previously calculated intersection point. We use these values and an environment map to calculate bubble reflections using the soap film shader. We can also calculate the internal walls for neighbouring bubbles.

Finally we have to draw the inner structure using the bubble texture. It can be an artist-drawn image of small bubbles (5), representing the inner structure of the foam, or a computer generated foam image using a complex non real-time simulation. The foam texture can be stored in another cube map. The sampling direction can be adjusted several ways as it is depending on the given blob and foam type. We used a weighted sum of two vectors (4). The first vector (v_1) is the direction of the second intersection

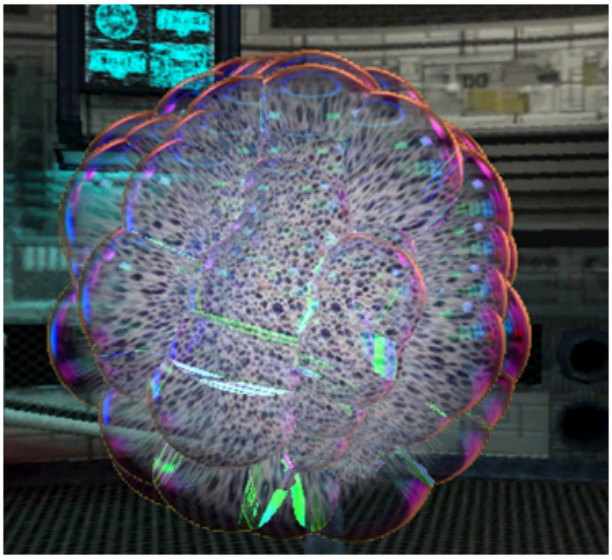

Figure 6: A single foam blob rendered using the bubble id method

of the first intersected bubble (P_1) relative to the centre of the blob (P_1). This spherically projects the texture onto the interior surface of the blob, which is what we get if we remove the outer bubbles. The second vector is the inverse normal of the bubble (v_2) at the first intersection (P_2). Combining these two vectors slightly distorts the original mapping based on the outer bubble geometry.

As stated before, the middle of the blob is opaque, while the outer bubbles are nearly transparent. To achieve this effect, the transparency of the inner texture is set according to the second intersection point's distance from the inner and outer bounding sphere. In the middle of the blob, the second intersection of the current bubble is closer to the centre as the ray is almost perpendicular to the blob's bounding sphere. Near the outer region, the intersection point is farther from the centre, so the blob will be more transparent there.

The last issue is the surface normal of the blob used for lighting equations. The nearly diffuse inner surface should slightly follow the wrapping surface of the outer bubbles. Therefore we used a weighted average of the bubble's normal and the blob's bounding sphere's normal. The normal and the other aforementioned parameters should be fine-tuned and set according to the actual blob structure and the desired foam type. A typical foam blob used in our renderings can be seen on Figure 6.

5 Rendering foam using foam blobs

Realistic dense foam needs to be constructed from many blobs, so we need a fast technique to render them. While ray tracing the blobs could be straightforward, it would be too slow for large foam. We propose a method based on particle systems, that is capable of rendering hundreds of

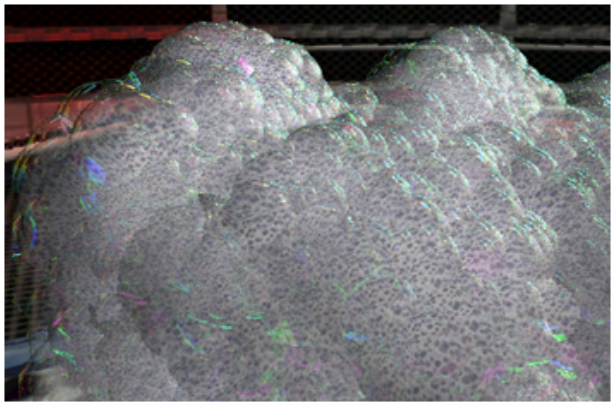

Figure 7: Dense soap foam rendered using the proposed technique

blobs real time. Blobs are rendered as 2D billboards. The ray from the eye position is calculated for all pixels of the billboard, and it is used to render the corresponding blob as described in the previous section. This means we do not have to do the intersection calculations for all blobs, but also means we cannot calculate inter-blob reflections (which would be too slow to use in real time anyway).

The data associated with the blobs are stored on the graphics card in a vertex buffer. The billboards are generated by the geometry shader using this data. The vertex positions in world space are also calculated in the shader, and used in the pixel shader to get the view rays for every pixel. Besides the colour, the depth of the blob is also computed for every pixel to address problems of overlapping particles.

Since the blobs are transparent, we need to sort the particles according depth, in order to use alpha-blending. However, depth is different in the pixels of the billboards, so we have to do the sorting at pixel level. Depth peeling [2] is a technique rendering translucent objects. It basically accomplishes pixel level sorting by using multiple rendering passes to store multiple depth levels for every pixel. We use two buffers to store depth data. First we render the scene depth into the first buffer, and then render it again into the second buffer, but only those pixels that are further than the stored depth in the first buffer. Then we flip the two buffers and repeat. We store the blob identifiers in a third buffer, in a different colour channel for each iteration. In the end, the n closest bubble identifiers will be in the final buffer. In the final rendering pass we render the blobs in order with proper transparency. While this method uses multiple rendering passes and it is generally slow, it provides a real-time alternative to ray tracing.

This method has one serious shortcoming in cases where multiple blobs overlap each other. Let's assume that we use three rendering passes (and store 3 layers of blob depths). Now imagine that the first three blobs are rather transparent, but there is a fourth, solid blob behind them. In this scenario, the resulting foam would be transparent,

resulting a transparent hole in the foam. To overcome this limitation, we use an extra rendering pass to calculate the maximal opaqueness for all visible blobs (not just the 3 closest to the viewer). When we draw the diffuse foam texture, we use this value to plug the unwanted holes in the foam. The main disadvantage of this method is the resource consumption, as we have to calculate intersection points for all blobs (however we do not calculate surface interference, reflections or wall geometry for these blobs). Figure 7 shows dense foam rendered using this technique. All the free parameters were set to grant visually appealing result.

It was not stated explicitly before, but blob rendering techniques require the blob structure to be static. The blob texture is prerendered once and then used for all blobs. This means that all the blobs are the same (it is possible to use several blob textures to construct a fixed number of different blobs). To counteract this limitation, we used transformations to change the size and orientation of individual blobs. This can be easily done real time in the pixel shader during the intersection calculations and grants us more diverse foam. To store these transformations, only one additional float vector is required in the vertex buffer. We can represent the orientation as a quaternion and store it in a four-dimensional vector. The blob size can be the fourth, previously unused coordinate of the position vector. We can also use a transformation matrix to store more general affine transformations, but in our implementation it was unnecessary to do so.

6 Foam physics

To provide even the most basic physical simulation, we must render solid objects beside the foam. As we used environment mapping for reflections and refractions, solid objects must be rendered using a blending technique. We first render these objects, and then blend the foam over them without clearing the depth buffer. This is efficient but this does not handle reflections of solid object on bubbles, or the rendering of transparent objects like smoke or glass.

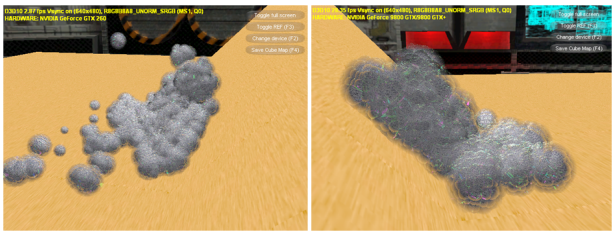

Figure 8: Foam formations sliding down on a slope

To simulate foam dynamics and present the characteristics of this rendering technique we created a basic but fast physical simulation based on particle dynamics. The simulation is able to handle inter-particle forces and outside objects. In the technical demonstration we presented a foam mass sliding down on a slope (see Figure 8). Each blob has a position, mass, and velocity, with forces acting between blobs and other objects. The blobs are represented by their bounding spheres. If two blobs get too close, two types of force can affect them: if they are far enough an attractive elastic force arises, modelling the different parts of the foam sticking together. However, if two blobs get too close, they collide, resisting collapse and giving the foam a solid structure. By properly adjusting these forces and the gravity, a good approximation can be achieved for the desired foam type. The simulation is done on the CPU. Further exploration in this topic could yield more realistic results and boost performance by implementing a physical simulation on the graphics card.

7 Results

We implemented the technique using DirectX 10 and Shader Model 4.0 on an NVIDIA Geforce GTX 260 graphics card. We achieved real-time simulation (32 FPS) of 100 blobs and a total number of 22700 separate bubbles, using the bubble ID technique for calculating intersections. The images were rendered at the resolution of 640×480. The various parameters like iteration count, texture size, and the number of bubbles in a blob were set to imitate soap foam to the highest possible fidelity without visible graphical glitches. Further tweaking these parameters could result in performance increase, while maintaining acceptable graphic quality.

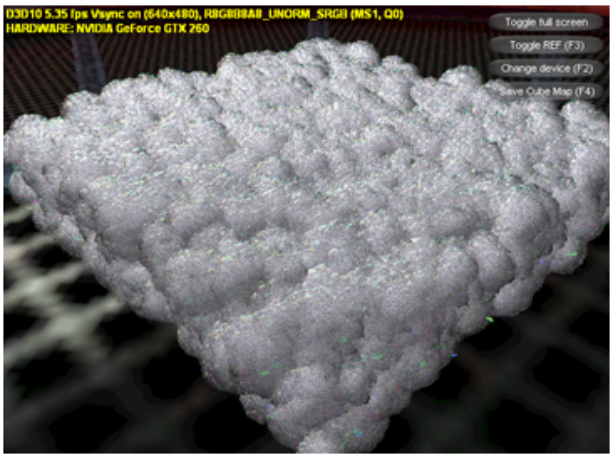

Figure 9: A box shaped form consisting of 5000 blobs

Using these same parameters, simulating between 100 and 1000 blobs the FPS stays above a reasonable rate (around 10). Figure 9 shows a foam formation of 5000 blobs and a total number of 1 135 000 bubbles, rendered at 5 FPS. Given these numbers, in the near future with further optimizations the real-time simulation of foam consisting of hundreds of thousands of bubbles could be possible.

8 Future work

The most serious limitation of the proposed technique derives from the particle based approach. Our blobs are static entities with fixed size and structure, and when we start to divide the foam into smaller pieces comparable to the blob size, it leads to artificial and unnatural results. To overcome this, we propose a possible direction. First, we need a model to adjust blob size dynamically, based on some physical observations, but not too complex to undermine the performance. Another possible way is to locally increase and decrease the simulation resolution (the blob size in this case) by the local foam characteristics and viewer distance. Blobs inside a dense foam or far from the viewer could be merged together, or their simulation and render quality should be otherwise decreased, while larger blobs broken out of the foam should break up into smaller blobs.

9 Conclusion

Bubbles and foam are extremely complex natural phenomena, the formation, motion and optics of which obey complex physical laws practically impossible to simulate in real time. A visually convincing result, however, is feasible with decent data structures and subtle approximations. As with a wide range of natural geometries, the concept of impostors is very helpful. We have shown how a generic impostor technique – the distance impostors – can be modified to represent bubble clusters, and we also proposed a specialized representation that exploits the fact that bubbles are spherical, and allows not only for a more accurate representation, but also for an approximation of internal foam walls. Furthermore, we described algorithms for the simulation and rendering of massive foam composed of the bubble clusters, based on particle systems and the billboard visualization technique. Our method is capable of real-time rendering dense foam consisting of ten thousands of bubbles on modern graphics hardware.

Acknowledgement

This work has been supported by the Teratomo project of the National Office for Research and Technology, and OTKA K-719922 (Hungary).

References

[1] L.M. Dias. Ray tracing interference color. *IEEE Computer Graphics and Applications*, 11(2):54–60, 1991.

[2] C. Everitt. Interactive order-independent transparency. *White paper, nVIDIA*, 2(6):7, 2001.

[3] A. Glassner. Soap bubbles: Part 1. *IEEE Computer Graphics and Applications*, 20(5):76–84, 2000.

[4] A. Glassner. Soap bubbles: Part 2. *IEEE Computer Graphics and Applications*, 20(6):99–109, 2000.

[5] K. Iwasaki, K. Matsuzawa, and T. Nishita. Real-time rendering of soap bubbles taking into account light interference. In *Computer Graphics International*, pages 344–348, 2004.

[6] S. Rosenbaum and M. Bergbom. Foam. http://cs.stanford.edu/people/rosenbas/foam, 2007.

[7] Y. Sun, F. D. Fracchina, T. W. Calvert, and M. S. Draw. Deriving spectra from colors and rendering interference. *IEEE Computer Graphics and Applications*, 19(4):61–67, 1999.

[8] M. Sunkel, J. Kautz, and H.-P. Seidel. Rendering and simulation of liquid foams. In *Vision, Modelling and Visualization*, 2004.

[9] L. Szirmay-Kalos, B. Aszodi, I. Lazanyi, and M. Premecz. Approximate ray-tracing on the gpu with distance impostors. *Computer Graphics Forum*, 24(3):695–704, 2005.