The Journal of the Society for Art and Science Vol. 14, No. 4, pp. 103-116

Procedural Interactive Water in Memory- and Performance-Constrained Systems

Jens Ogniewski

Information Coding Group, Department of Electrical Engineering, Linköping University
jenso at isy.liu.se

Abstract

Particle effects are vital components of computer graphics in modern computer games. While game developers have a choice of several different methods for particle effects on PCs and home consoles, there exist only few solutions for games in the fast growing smartphones/tablets market. This is not only because of the more than a magnitude lesser computational performance of the systems-on-a-chip used there, but especially due to their even much slower memory access, which renders nearly all approaches used on standard PCs unsuitable for smartphones/tablets.

To overcome the bottleneck of the memory access, I suggest using a procedural approach, which will be described fully in this paper, with the example of real-time water. It is based on particle movement in 2D, but by applying physical forces directly to the particles rather than using a pressure-field like in e.g. the popular Navier-Stokes based methods. This has the advantage of avoiding the need for two different data structures, one for the pressure-field and one for the particles themselves, and thus reduces memory usage significantly.

In this paper I will also present a simple way to introduce interactions with the particle effect as well as a comparison with a low-complex Navier-Stokes based approach. To the best of my knowledge, this is the first scientific work investigating particle effects for systems-on-a-chip, like e.g. smartphones/tablets.

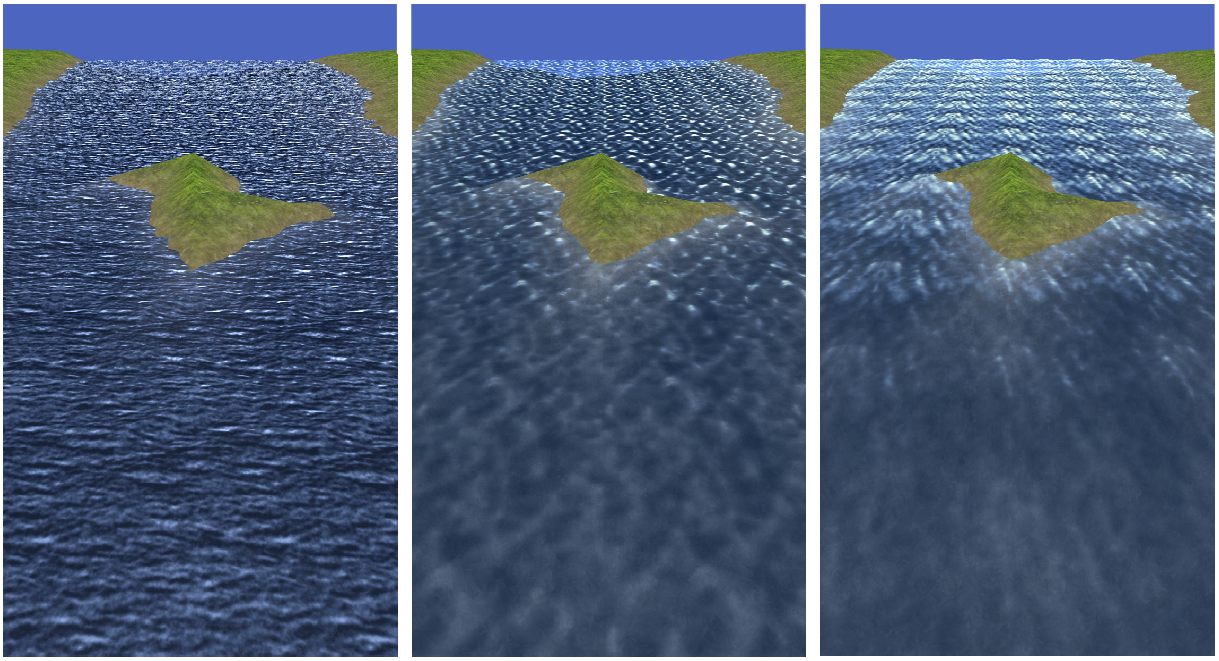

Figure 1. Example images from the test-sequence using a) (left) a simple texture, b) (middle) the force-based approach as presented in this paper and c) (right) a Navier-Stokes based method (as presented in [10]). Pictures taken from the test sequences running on a Samsung Galaxy Note 10.1 2014 Edition.

1. Introduction & Motivation

1.1 Smartphone/Tablet Architecture

To fully understand this paper, it is necessary to be aware of the big differences between PCs-architecture and the one of so called systems-on-a-chip (called SOCs in the following) which are used in basically all modern smartphones/tablets. I will therefore start with a short comparison, based on results from the IceStorm benchmark [8], which is available for both SOCs and PCs. It should be kept in mind that this benchmark slightly favors the SOC GPUs since it does not use many of the more advanced features of modern PC GPUs which are not included in SOC GPUs.

The iPhone 5 for example reaches 5693 points in this benchmark, which puts it in the middle-class of that generation of SOC GPUs. A middle-class PC GPU of the same generation (the Nvidia GeForce GTX 650 TI) reaches 118057, or more than twenty times as much. Of course, since then SOC GPUs have become faster, for example the more contemporary Galaxy Note 10.1 2014 Edition is (compared to the iPhone) 2.5 faster in the same benchmark, putting it in the lower high-end class of the last generation of SOC GPUs. On the other hand, PC GPUs have become faster as well, a comparable last generation high-end GPU, the NVIDIA GeForce GTX 780 TI, reaches 3.3 times the performance of the aforementioned Nvidia GeForce GTX 650 TI. The current generation SOC GPU NVIDIA K1 closed this gap a little however: compared to the Geforce GTX 780 Ti it is "only" slightly more than 13 times slower (365 GFLOPS of the K1 compared to the 5 TFLOPS of the 780 TI). Note that the K1 already uses the same technology used in PCs, and therefore it is unlikely that the performance difference will decrease much further.

The biggest difference between PCs and SOCs is however that in a SOC all components are integrated on the same chip (hence the name), and all share the same bus and memory. This means that the GPU in a smartphone/tablet has to share these with the CPU, but also with the modems, the touchscreen, the camera and all other active component. On PCs however most components have their own dedicated busses and memory. The GPUs in modern PCs for example have roughly the same amount of memory for themselves than most SOCs use for the whole system.

Also, the available memory bandwidth is much smaller in SOCs. The K1 for example reaches only 17 Gb/s (shared between the GPU, the CPU and all other components), while the 780 TI has 336 Gb/s available all for itself, twenty times as much. This difference is nearly 1.5 times bigger than the difference in performance. It can be assumed that the gap between computational performance and available memory bandwidth will become larger still, since computational performance is growing faster than available memory bandwidth.

On the other hand, the SOC architecture has huge benefits in costs and energy savings (note that the battery development has not kept up with the chip development), which are the driving forces behind the current replacement of PCs by smartphones/tablets in normal consumer households. Thus it is highly unlikely that coming generations of smartphones/tablets will apply a different architecture. On the other hand, this architecture and its limited memory access unfortunately renders most PCs algorithms unsuitable for these kind of systems. Thus, it is high time to develop algorithms optimized for the big and fast growing smartphones/tablets market, which means especially to minimize memory usage as much as possible.

1.2 Particle Effects

The realistic visualization of particle effects, like water, has been a topic of much interest since the beginnings of computer graphics, going as far back as to the beginnings of the 80s [33]. For many use cases (like movies or computer games) a physical accurate behavior is desirable, but not necessarily needed. It is much more important that the designer can reach the desired effect, and, if possible, can do so easily. For applications like computer games it is of course also necessary that it is possible to run the method in real-time.

An often used approach are volumetric particle systems, as described e.g. in [39]. These are divided into two parts: the simulation of the particle movement, and the rendering of the simulated system. The simulated particle movement takes typically place in a so called voxel grid, which is a discretized, closed space (realized e.g. by 3D-textures) containing a certain number of cells, called voxels. Each voxel can then contain a certain number of particles, which are moved in accordance to a physical model like Navier-Stokes [33] or Lattice-Boltzmann (e.g. [14] [29]). However, the memory usage increases enormously with the resolution of the grids. Therefore, several approaches have been suggested for effective compression of these voxel grids, thus trading lower memory footprint for lower computational performance. A common solution for this is to use octrees, like e.g. in [19].

Although volumetric systems have traditionally been used exclusively in offline methods (e.g. the water in Titanic [35]; an overview of these methods can be found in [4]), the development in GPU architectures in recent years has made it possible to use volumetric particle models even in real-time applications like computer games, at least if using a comparably modern PC or gaming console. Early real-time PC GPU implementations include [16] and [37], which however both concentrated on the simulation part and included only a very basic renderer. More recent work can be found in [6] and [9]. Still, to be able to render volumetric particle systems in real-time, trade-offs have to be made, and the work in that area differs mainly in how these trade-offs should be made to get the visually most pleasing result with the best possible performance. A trivial trade-off is of course the size of the voxel grid or the number of involved particles. Another often use approach would be to use a 2-dimensional pressure field instead of a 3-dimensional one, as already presented in [17]. But, as has been shown in several publications, other trade-offs are possible as well. [18] for example suggested a method combining a low-frequency approximation of the particle system and a

ray-marching method for the light transport. Similar ideas can be found in other work, like [41], were the actual light rendering is precomputed, which allows real-time simulations of comparably large grid-sizes using commodity PC hardware. [30] suggested a variable grid-size, so that cells of a particular interest can have a higher resolution than others. [34] and [13] recommend refinements to add details to comparably coarse grid-sizes. [27] on the other hand suggested to use offline precomputation to build a Markov-type velocity field for the online simulation to reach real-time performance. Finally, [28] showed how using a mathematical description of the grid (instead of an actual grid) can be used to emulate a very high resolution grid in reasonable computing time. This approach can be seen as a bridge to more mathematical equation-based solutions, like e.g. [15], where water is described purely by equations. Another approach combining volumetric and equation based methods can be found in [2], where the notion of wave-particles is introduced.

For the rendering of volumetric simulated liquids, often algorithms like Marching Cubes [20] or Level Set [26] are used. Small detail can be added by using turbulence [40] or advection (see also [22] or [32]), which is usually derived by a random noise which is offset by a turbulence or velocity field, which is normally done in the texture domain. Other possibilities include to augment the rendered particles with textures (like in e.g. [1] or [5]), or to use displacement mapping [42] or animated heightfields [23]. A typical approach describing interactions with such a simulated particle system can be found in [3].

Unfortunately, even the most performant PC methods are badly suited for SOCs, mainly because of a too high memory utilization. In current computer games on smartphones/tablets, most designers implement particle effects through so called particle systems, which in these cases however do not simulate particle movements. Instead, they consist of several (often animated) billboards moving in predetermined pattern (although a little randomness is normally introduced to get a more realistic appearance), much like as described in [12] or [38]. Research in this area typically concentrates on how to simplify the process to get the desired effect, like in [36] or [24]. These approaches have very low computational complexity, but have the disadvantage of using a comparably high amount of memory, which is fast increasing if the effects should emulate dynamic behavior or should have very varying looks. Considering how costly memory access is in SOCs a more procedural approach would be much better suited to these systems. However, the only procedural method that I am aware of that has been applied in SOCs is to use simplified equational methods to simulate interactions with liquids, similar as has been proposed in [15].

There have been a few papers published for volumetric rendering on smartphones (e.g. [21] or [31]), which however omitted the simulation part needed for animated particle systems. Also, the reported frame rate is much too low for real-time applications like games.

A solution would be to use a 2-dimensional approximation to emulate a full volumetric system. This is based on the observation that a fairly good result can be achieved solely based on the number of particles in front of the observer, i.e. without knowing their exact position in depth, only knowing how many particles any possible ray from the observer through the particle volume would hit. Thus, the demands on memory and computational power can be decreased significantly, enough to be able to run such systems even on older SOCs in real-time. A related approach was described by [17] and [10], using a simplified Navier-Stokes solver. Here however I suggest to replace the Navier Stokes pressure field by a low complex force-based physical model, thus saving the pressure field and hence most of the required memory (since the pressure field ideally has to be at least four times bigger than the one containing the particles). The simulation step applied in this paper was already described in [25], which to the best of my knowledge presents the first work where particle effects like smoke, fire and water were generated on SOCs by a purely procedural approach, i.e. without the use of precomputed data like textures, thus minimizing memory usage. This paper will give a short description of the simulation process, but concentrate on rendering aspects, with a focus on water rendering. Furthermore, a comparison is given with texture based methods as well as with the Navier Stoke based solution described in [10]. This paper is based on earlier work I presented at NICOGRAPH 2014 [43].

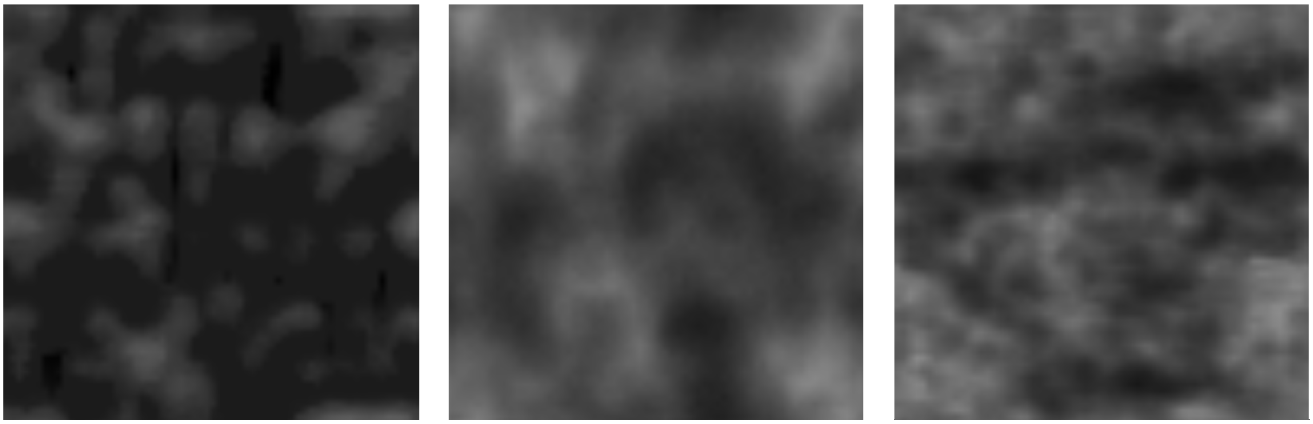

Figure 2: Particle fields derived using different approaches: a) (left) using a Navier-Stokes pressure field of equal size as the particle field, b) (middle) using a Navier-Stokes pressure field which is 4 times bigger, and c) (right) using a force-based approach as described in this paper. Note that the same algorithms were used to produce these pictures as in the test-sequences

2. Particle Simulations

As was already mentioned, memory usage in SOCs should be minimized as much as possible, which can be achieved by simulating the particle movement in 2D rather than in 3D. The structure containing the particles will be called particle-field in the following. To be able to save the field in one color channel of a texture, the maximal number of particles in a cell will be limited to 255.

In [10] particles are moved using a Navier-Stokes pressure field, which is set to be 4 times larger than the particle field. Own experiments showed that using a smaller pressure field leads to very slow movement, and tends to concentrate the particles in a few locations, probably because of the high smoothness of the pressure field, see also figure 2a). For comparison, figure 2b) shows the same Navier-Stokes based approach using a pressure field 4 times larger than the particle field and 2c) the result of using the approach proposed in this paper. Note that 2b) and 2c) use the actual algorithms used in the later described test sequences.

If the large pressure field is omitted to minimize memory footprint, the particles have to be moved in a different way, which can be done by applying forces directly to the particles. This will be called force-based approach in the following.

Physically, the movements of particles are the result of a number of different forces, like inertia (i.e. along the current trajectory), diffusion (from places where a lot of particles reside to places where fewer particles are), and external forces (e.g. gravity). These are the three forces which are included in the approach described here, along with a random force to emulate small scale effects. This method has the additional advantages that the simulation can be easily controlled by choosing the blend-weights between the different forces, thus simplifying the generation of the desired effects, as well as that the simulation can be done in one step, unlike Navier-Stokes where the pressure field has to be updated first before the particles can be moved.

The particles are only moved from one cell to the 8 directly adjacent cells, and the directions are not calculated for each particle but for all in a cell at once, both to save computations. This can of course lead to very monotonous movements where many particles travel along the same path. To conquer this, I suggest moving particles not only in the exact direction of the force, but even in nearby direction, which can be seen as using a more stochastic approach. To determine how many particles should be sent in which direction, e.g. a gauss distribution could be used. I chose however to use a cosine function instead, since this means that the number of particles which should move in one direction can be calculated by a dot product between this direction and the force in question, which can be computed very fast. This is possible since all forces and the candidate directions can be easily described as vectors, with the exception of diffusion, which is simply expressed by the differences in the number of particles in neighboring cells.

Thus, we arrive at the following equation to describe the movement along the direction **D** from the center cell to a neighboring cell:

$$M_p = \frac{1}{|\mathbf{D}|}\left(\sum w_i \frac{\mathbf{D}\cdot\mathbf{F_i}}{|\mathbf{D}|} + w_d(p - p_i)\right),$$

with $\mathbf{F_i}$ the different force vectors, w_i and w_d the different blendweigts, as well as p and p_i the number of particles contained in the center cell and the number of particles contained in the neighboring cell the direction vector points to. The second term calculates the movement due to diffusion, the first the other forces. For the random movement we need to evaluate the first term two times, since each of the two cells has its own random vector. Note that M_p will become negative if particles should be moved to the center cell, rather than away from it. I considered only the 8 nearest cells, i.e. a 3x3 neighborhood. The division by the length of the candidate direction in the beginning of the equation is done to compensate for the higher distance to the cells in the corners of the neighborhood.

To make sure that no particles are created or destroyed, not more particles should be moved from a cell than reside in it, and no particle should be moved to a cell that is already full. Thus, it might be necessary to scale the number of particles that move between two neighboring cells. This is done by using scale factors applied to all particle movements involving the cells in question, to make sure that the proportions of the particles traveling in the different directions will not be changed by this scaling.

To be able to calculate the inertia in the next simulation step, the velocity of the particles in the current step has to be saved. It was decided to save the average velocity of all particles contained in the cell instead, to save both on computations and required memory; in fact it could be saved in 2 color channels of the particle field.

The different forces are shown in figure 3, and the output of a simulation using a Navier-Stokes pressure field as in [10] is given for comparison. As can be seen, the particles spread out more and faster using the force-based approach. This is because each combination of force and candidate direction is evaluated individually, thus allowing particles to move in many different directions in the same simulation step. In case of Navier Stokes the particle movement is calculated in the same way (by using dot products), but the pressure field provides only one vector, thus the number of directions the particle move in in each step is limited.

It should be pointed out that it is possible to get a result that looks similar to the Navier-Stock result with the force-based approach. It seems that the force-based approach provides a higher flexibility than Navier-Stokes.

3. Rendering

Although in some situations a 2-dimensional particle effect might be enough (like e.g. a fire in a fireplace), in most cases a 3-dimensional one would be preferable. Since the simulations have been done purely in 2D, this 3-dimensionality has to be introduced in a different way. [11] suggests to add an additional field during simulation to get a so called 2.5-dimensional effect, which however introduces an additional data structure and thus increase the memory footprint significantly.

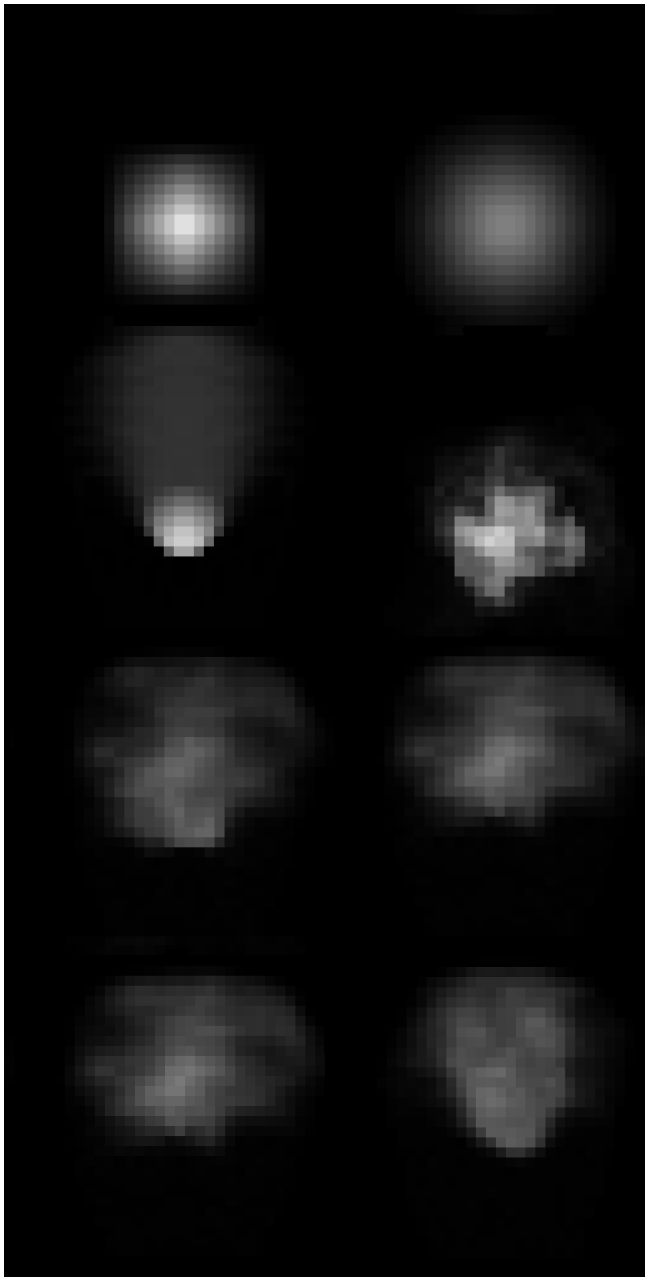

Figure 3: Example to illustrate the different forces used in the suggested approach. The size of the particle fields were 32x32, and the blendweights were 0.1 for the external and random forces, and 0.15 for the others. Note that these parameters have been chosen solely to allow for a good demonstration.
1st row: a) (left) input particle field,
b) (right) movement according to entropy,
2nd row: c) (left) movement along a common external force,
d) (right) random movement,
3rd row: e) (left) all three forces combined,
f) (right) with additional inertia,
4th row: g) (left) same as f),
h) (right) Navier-Stokes as in [10] for comparison

Instead, the particle field could be used as a displacement map (like e.g. in [42]) of an arbitrary object, i.e. the object will appear thicker where more particles reside, and thinner were fewer are. This assumes that the number of particles in the voxels in the core of the object doesn't change during simulation, which is a reasonable simplification. This method has the additional advantage that the designer can choose roughly which shape the effect should have.

For water, this displacement is trivially done by starting with a flat surface, than adding to the y-component of each vertex the value found in the particle field at its positions, multiplied with a constant. Similarly the normals needed for specular lightning can be calculated in the fragment shader: by calculating the heights in neighboring points, and use these to calculate the normal of the plane spanned by the neighboring points and the point we are currently coloring. Because the reflection of a water surface is directly proportional to the amount of light that will pass through its surface, the specular lightning can be used for transparency calculation as well.

Since the vector field is treated in a wrap-around fashion (i.e. particles leaving the field on one border will enter it immediately on the opposite border, this is done to keep the number of particles contained in the field constant), it is possible to tile this height field, i.e. it can be rendered several times directly adjacent without any visual seems. This makes it possible to simulate a large body of water using one single, comparably small particle field. Due to the lightning effects which vary depending on the pixel position, the repeating patterns are barely visible when the water is moving. To make the reappearing patterns even less visible, the texture coordinates and the vertices used for displacement mapping could be spread in a less regular pattern as was done in this work.

To further minimize the size of the particle field and improve the effect, local advection (like in [22] or [32]) can be used for the small scale effects. Since the average velocity vectors are already included in the particle field (to be able to calculate the inertia), it makes perfect sense to use them as turbulence for the advection as well. The degree of advection can be varied by the distance between the pixel in question and the camera, i.e. a stronger effect can be used when it is near the camera, and a lesser or even none at all if it is farther away. In this way a mipmapping like effect is achieved. The advection process is visualized in figure 4.

For absorption and refraction effects, the scene would normally needed to be rendered at least twice. Since this would add heavily to both the computational burden and the memory footprint, this should however be avoided. Instead, I suggest here to use an approximated light transport based on ray-casting. Outscattering, absorption and inscattering as needed for light transport could be approximated locally using only the number of particles in the neighboring cells. Assuming a constant number of particles in the voxels in the center (as described earlier), outscattering and inscattering cancel each other out and therefore the approximated function only depends on the distance the light traverses through the system. This allows to introduce the

absorption effect of the water in the following way: when a pixel belonging to the bottom of the river is rendered, a ray is casted from it to the camera position, and it is determined where it would hit the water surface assuming an average water height (i.e. the water surface which has been displaced with the number of simulated particles divided by the number of cells). The length of this line can then be used to calculate an approximated absorption and to darken the pixel on the ground accordingly.

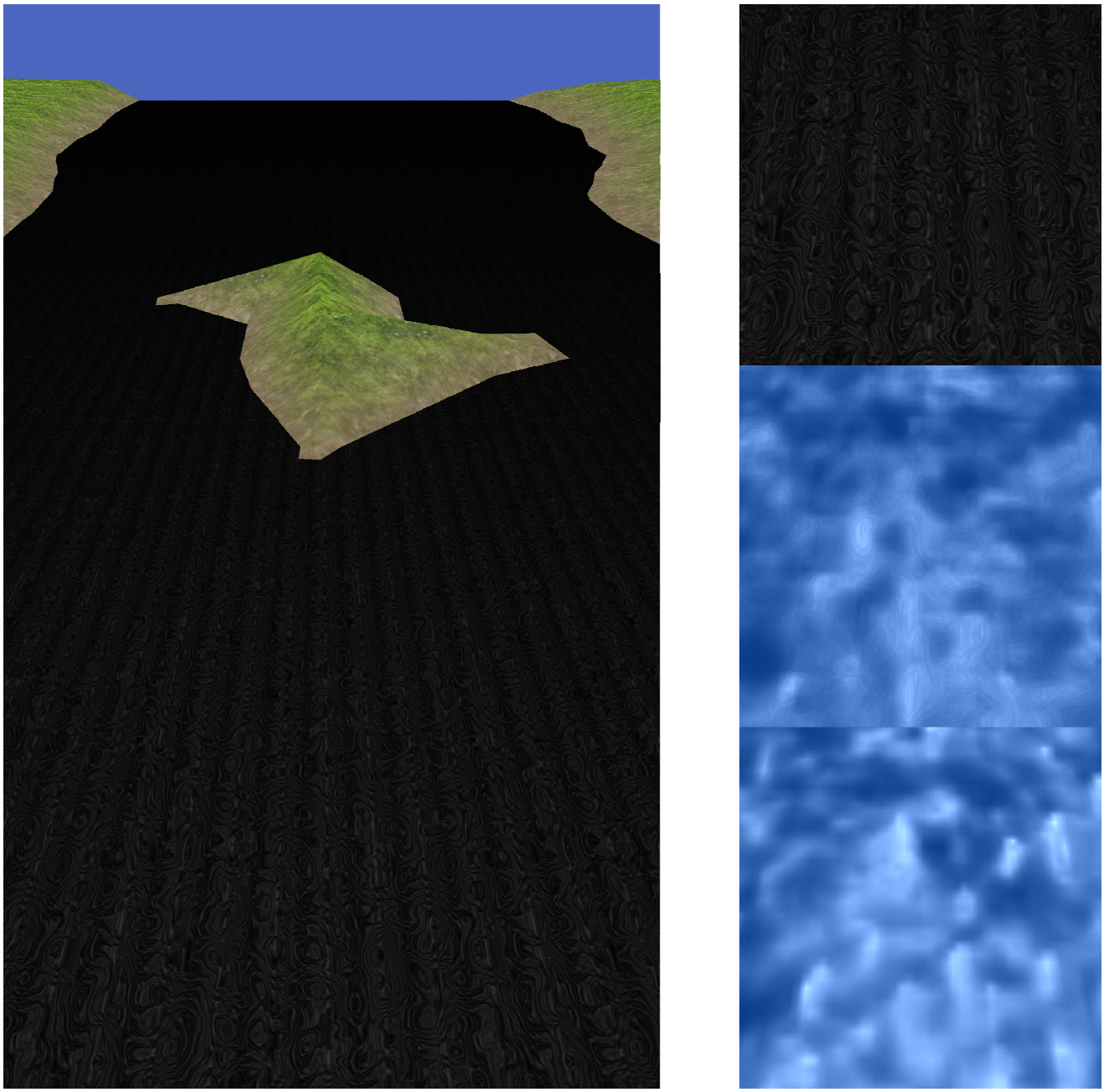

Figure 4: Example to illustrate the advection process: a) (left) advection based on the distance to the camera, b) (right, up): close-up, c) (right, down): water without advection, d) (right, middle): water with advection. Pictures taken from the test-sequence running on the Galaxy Note, using the force-based approach, however with the transparency of the water turned off.

Furthermore, the normal found in this intersection point with the water surface, together with the length of the line as used for the absorption, can also be used to calculate a first degree approximation of refraction effects. This could be used in two different ways: 1. to move under-water vertices to where they appear to be and 2. to modify texture accesses accordingly. While the first method can lead to an overall more correctly looking scene, many vertices would be needed to get an accurate result, especially for the typical undulated lines caused by fast moving water. A good solution would therefore be to combine both methods. This is an approximate solution, however in

practice the accuracy lost should very hardly be visible, at least in the case of fast moving water.

Refraction and Absorption are visualized in figure 5.

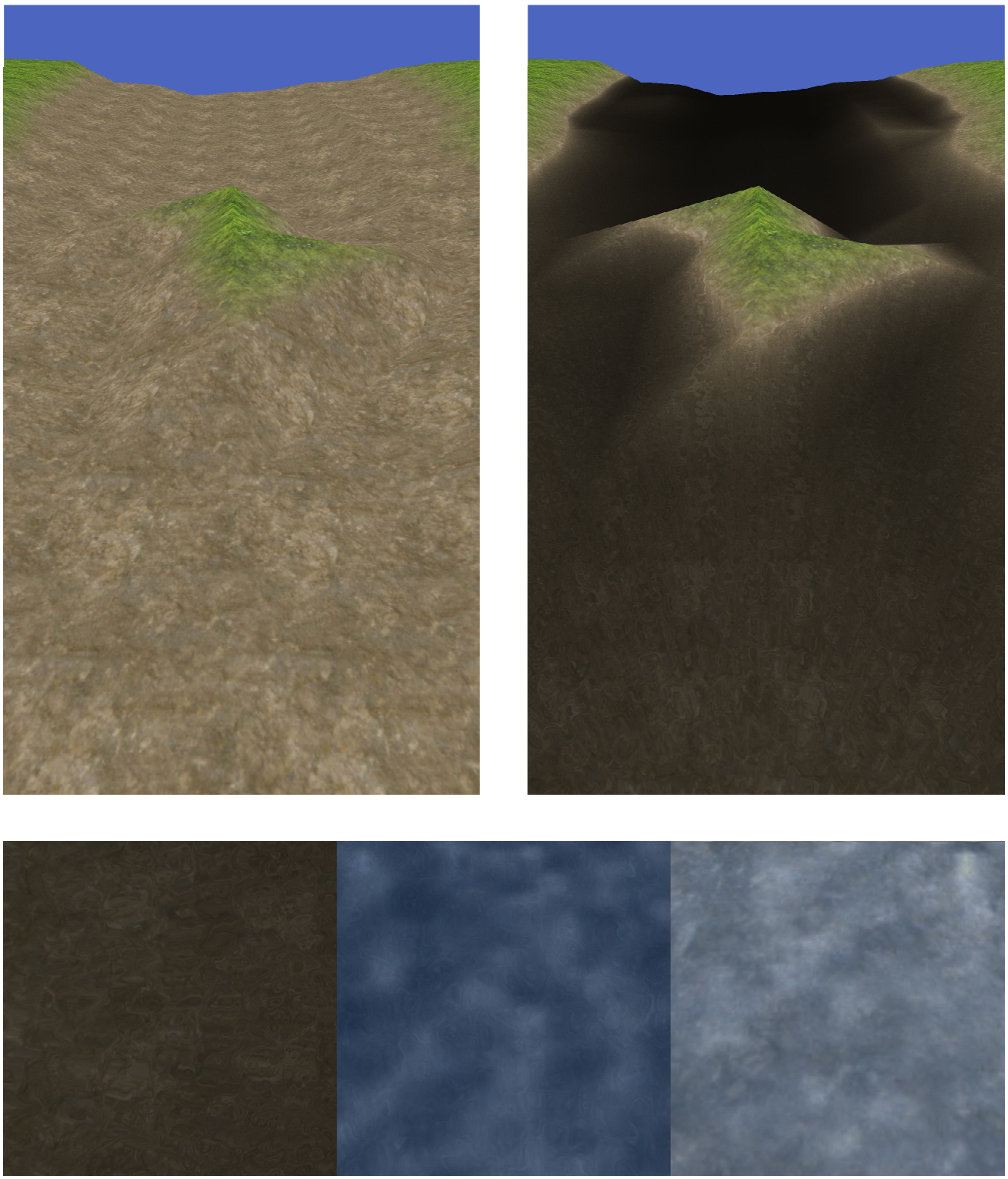

Figure 5: Example to illustrate absorption and refraction. a) (up, left): ground rendered without absorption or refraction, b) (up, right) ground rendered with absorption and refraction, c) (down, left): close up of ground with absorption and refraction, d) (down, right): ground and water rendered without absorption or refraction, e) (down, middle) ground and water rendered with refraction and absorption. Pictures taken from the test-sequence running on the Galaxy Note, the force-based approach was used.

Finally, interaction with the water can be easily introduced using an equation based method. For example, the typical ripples caused by an approximate spherical object hitting the water surface can be described by a number of rings, which can be calculated by the combination of two functions: the shortest distance of the pixel in question to the center ring, and a wave function (e.g. a cosine) to introduce a number of ripples. This whole method can be controlled by 4 different parameters: the 2D coordinates of the center of impact, the current radius of the center ring (which will grow over time), and the force of the impact (which will be reduced over time). To further increase the phasing-out effect, the used wave function should be steeper directly after the impact, and more spread out later. The results of these simulated impacts are visualized in figure 6. Note that it is of course possible to have interactions between the ripples of different impacts. Furthermore, note that these interactions influence the displacement mapping, the local normal, and the turbulence used for the advection.

This method introduces a high number of additional computations and is therefore not suitable for a high number of interactions, as will also be demonstrated in the evaluation. However, it shows how easily an additional method for water interaction can be added, like for example equation-based approaches as described in e.g. [3], which have become a popular method in games for smartphones in recent years. These two methods complement each other rather well, since equation-based systems are very good in modeling interactions, but can get computational expansive if they have to simulate the normal water flow with a high visual quality. The method described in this paper on the other hand handles normal water flow with very little resources, but has to rely on an additional method for interactions, as for example the simple water drops described earlier in this chapter.

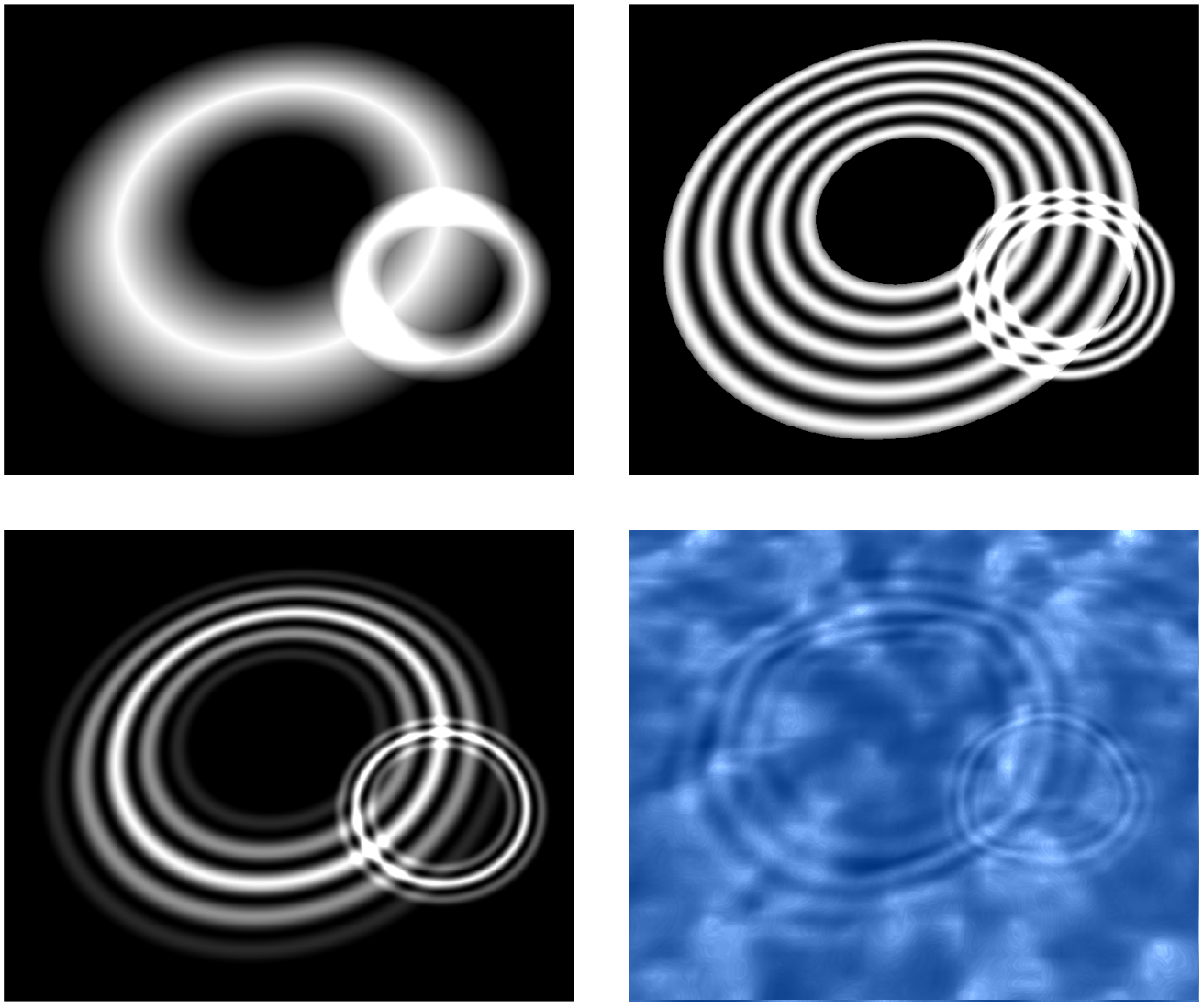

Figure 6. Example for a simple way to introduce interaction with the water surface. a) (up, left): distance function to the center ring, negative values are set to zero, b) (up, right) cosine function to introduce several ripples, c) (down, left) the two functions blended together, d) (down, right): the ripples blended with the water surface (with additional weights applied based on the force of the impacts and the time since they occurred). Pictures taken from the test-sequence of the force-based approach on the Galaxy Note; the transparency was turned off to give a better view of the effect.

4. Evaluation

For an evaluation, the described algorithm was implemented on an iPhone 5 (representing a middle-class SOC GPU from third-to-last generation) and on a Samsung Galaxy Note 10.1 2014 edition (which includes a last generation lower high-end GPU). The screen resolution of the iPhone is 1136x640, the one of the Samsung 2560x1600. Note that these native resolutions were used in the test-sequences as well.

The scene consists of 420 vertices for the ground and 8820 vertices for the water, and is only nearly completely inside the viewing fustrum. However, no techniques like fustrum culling etc. have been used. The ground is rendered using a blending of two different textures (using the y-positions as blendweights), with a size of 128x128 each.

Since a refraction in the vertex shader added little to the result (due to the comparably low number of vertices used in the ground as well as its very uniform appearance), this was not included. It would also be very difficult to introduce this to the iPhone, since its operating system did not permit texture access in the vertex shader. For this reason, to be able to use the displacement mapping on the iPhone the particle fields had to be read back to the CPU and send to the vertex shader as uniforms. This is possible since the texture spreads evenly over 4x8 vertices, i.e. only 32 values have to be sent to the vertex shader. However, it is not possible to read only single values back to the CPU, so the whole texture has to be read back, thus heavily increasing the number of memory accesses.

For the noise, it would be possible to use a noise algorithm like e.g. [7], but even these highly optimized solutions proved to be too slow. Therefore it was chosen to use a noise texture instead, which is a texture that contains random values and is a common solution. This has the additional advantage that the noise can be custom tailored for the effect that should be reached, e.g. in the case of fire concentrating high values in one point and lesser values in the rest lead to more flame like structures. The disadvantage of this method is of course that it introduces an additional data structure, but it turned out that the random texture can be chosen to be relatively small, because:

1. normally the random texture should retain a certain smoothness, which can be approximated by using a small texture with completely random values and scaling it up at runtime using the inbuilt texture interpolation.
2. The texture can be repeated several times instead of using a texture that is several times larger.

Note that this proved to be sufficient for the rendering especially since the particle systems move too fast for the human eye to depict all the details, and that it has the big advantage of having a very low computational complexity and a comparably small memory footprint at the same time.

Since the particle field only uses 3 color channels (one for the particles and two for the average velocity), the random- texture can be added to it as the alpha-channel, and thus the system can be rendered with advection using only one texture. However, it proved to be beneficial to add a second texture containing precomputed normals based on the current particle field, mainly because this reduces texture accesses: only one texture access is needed to determine the normal instead of at least two which would be the case if the normals were calculated at runtime.

All together three different methods were implemented: one using the force-based approach, one using Navier-Stokes as in [10], and one using a single texture. If not stated otherwise, the rendering for all was done in exactly the same way.

The size of the single texture was 256x256 which is a compromise between a small size and reaching a detail level comparable to the advected particle fields. No advection could be used in case of the single texture due to the absence of an animated turbulence field. The free three color channels of the texture were used to include the local normal vectors for a similar lightning and refraction calculation as in the procedural methods (Navier-Stokes as well as the force-based method). Note that the detail of the single texture is less in the places where maximum advection is used, but too high in the places where only low advection occurs..

Table 1. Performance results from the test-sequences running on a Galaxy Note. All values given in Frames/sec.

	Simple Texture	Force-based	Navier-Stokes
Ground	119.9		
Only water	99.2	88.1	77.8
Incl. Interactions	62.5	49.0	47.5
Incl. refraction/ absorption	32.2	41.6	41.0
Incl. Interactions and refraction/ absorption	27.8	31.0	30.0

Table 2. Performance results from the test-sequences running on an iPhone 5. All values given in Frames/sec.

	Simple Texture	Force-based	Navier-Stokes
Ground	115.2		
Only water	121.9	85.7	83.5
Incl. Interactions	50.2	29.2	28.7
Incl. refraction/ absorption	18.9	43.7	42.4
Incl. Interactions and refraction/ absorption	14.1	22.5	22.3

This could of course be avoided by using mipmaps, however at the cost of adding texture data. In a similar way the movement of the water could be implemented by using several textures, which could also be used to introduce more variations to the movement. It is however very doubtful that this would reach a higher performance than simply

continuously increasing the texture coordinates, which was therefore chosen for the comparison

For a simulated interaction with the water, two different animated impacts have been added, each of which is replaced with a new one as soon as it becomes invisible. All their parameters are chosen randomly, but it is made sure that the centers of the impacts lie inside the viewing fustrum. It was chosen to include two to be able to emulate interactions between them as well. Note that these impacts influence both the normals for the lightning and transparency calculation as well as the displacement mapping. In case of the two procedural approaches they also influence the advection.

The performance was measured in 4 different test-sequences:

- Only the water itself, i.e. only displacement mapping and heightmapping. In case of the procedural approaches the simulation of the particle fields and the advection was added in the measurement as well.
- The water itself (as in the first test-sequence), as well as the animated interactions
- The water itself (as in the first test-sequence), as well as the approximated light-transport and refraction
- The water itself (as in the first test-sequence), as well as both and the animated interactions and the approximated light-transport and refraction

The measurement was done by first measuring the average time needed to only render the ground (without light-transport and refraction), then measuring the average time for the whole render-loop for all test-sequences (note that this includes the particle field simulation in case of the procedural approaches). The averages of each test-sequence were subtracted by the average time needed to only render the ground. The results are given in table 1 (for the Galaxy Note 10.1 2014 Edition) and table 2 (in case of the iPhone 5).

Comparing the two devices it should be kept in mind that the Galaxy Note has to render more than 5.6 times as many pixels as the iPhone, which is important since the overall performance is mostly limited by the performance of the fragment shaders rendering the ground and the water. The Galaxy Note is especially faster in sequences where a lot of computations are done (e.g. the ones including the animated interactions), which shows that computational power in SOC GPUs has grown more rapidly than the available memory bandwidth.

It is also of interest that the procedural methods outperform the simple-texture method wherever light-transport and refraction are used. This means that the higher memory usage of the texture based method (note that the ground has to access the water texture/particle field as well for the calculation of refraction and absorption) actually harms performance more than the simulation of the particle fields and the advection together, and proves that procedural methods are preferable to those relying on precomputed data, at least in systems based on SOC architecture.

The performance of the force-based approach and Navier-Stokes are nearly the same. This was expected, since the bottleneck lies in the fragment shaders used to render the scene, which are the same in both approaches, while the actual particle simulations do not have a big influence on the performance.

The force-based approach has on the other hand a much smaller memory footprint, as can be seen in table 3, which sums up the memory usage of the different approaches. The size of the textures used by the particle fields were 64x64, the size of the Navier-Stoke pressure field 128x128, and the random values needed by the simulation were stored in a 16x16 texture, which is big enough for the same reasons as mentioned earlier (note that the noise texture used during simulation is a different one than the one used for the advection due to different requirements of the respective algorithms). Also, the randomness was increased by reading this random texture with an offset, which was chosen randomly in each simulation step.

Table 3. Memory comparison of the three different approaches

	Simple Texture	Force-based	Navier-Stoke
Total	(256x256)*4 = 256 kByte	(64x64*4+16x16)*4 = 65 kByte	(128x128*2 +64x64*3 +16x16)*4 = 177 kByte
Percentage	394%	100%	272%

For a visual comparison, a limited subjective test has been performed. The subjects were asked to look at a video which depicted all three different systems side-by-side, and asked to rate the quality on a scale of 1 (lowest) to 7 (highest). The side-by-side presentation was chosen to make it easier to compare the sequences. The resolution of the systems had to be reduced drastically (to 720x426), and the video-compression introduced some minor artifacts, while removing some of the finer details. However, since this effect occurred evenly in each of the different systems, their results can still be compared.

The subjects in question where master students which participated in our computer graphics course, and they did the test voluntarily and anonymously, thus also without receiving any kind of compensation. 23 subjects participated, and they rated the texture-based system with an average of 4.2, the Navier-Stokes based system with an average of 3.4 and the force-based system (as presented in this paper) with an average of 4.3. Thus, it can be stated that the method presented in this paper reaches a visual quality which is a least equal to other, more memory consuming methods.

The reader is invited to do an own comparison using the figures 1, 7 and 8 as well as the attached video. Note however that the visual quality of these examples is degraded due to lossy video/image compression and a much lower resolution.

5. Conclusion

SOCs, which are used in basically all smartphones/tablets (and which will be used there with a very high probability even in the

future), need a slightly different approach to algorithmic design than if aiming for standard PC hardware. In this paper a very first method has been presented for procedural generation of water effects on these devices. Comparisons with another, more traditional but highly optimized approach show that the method of this paper is only slightly faster, but (more importantly for SOC software design) has a much lesser memory footprint. Also, it seems to be more versatile and simpler to use.

In practice, procedural approaches can outperform methods relying on precomputed data like textures (e.g. particles systems). Compared to precalculation-based approaches, procedural methods have the additional advantage that they make it simpler to introduce dynamic effects like a high variation in movement or adaptation of small-detail effects, e.g. based on the distance to the viewer.

Thus, using a novel, memory conservative and procedural approach it was possible to simulate a comparably large body of interactive water in real-time even on the limited hardware used in last-generation smartphones/tablets.

References

[1] Busking, S., Vilanova, A., van Wijk, J.J.: Particle-based non-photorealistic volume visualization. The Visual Computer, vol. 24, iss. 5, pp. 335-346 (2008)

[2] Cem Yuksel, House,D.H., Keyser, J.: Wave particles. SIGGRAPH Conference Paper, Article No. 99 (2007)

[3] Cords, H., Staadt, O.: Real-Time Open Water Environments with Interacting Objects. Eurographics Workshop on Natural Phenomena (2009)

[4] Darles, E., Crespin, B., Ghazanfarpour, D., Gonzato, J.C.: A Survey of Ocean Simulation and Rendering Techniques in Computer Graphics. COMPUTER GRAPHICS forum, Volume 30 (2010), number 1, pp. 1–18

[5] Dong, F., Clapworthy, G.J.: Volumetric texture synthesis for non-photorealistic volume rendering of medical data. The Visual Computer, vol. 21, iss. 7, pp. 463-473 (2005)

[6] Deukhyun Cha, Sungjin Son, Insung Ihm: GPU-Assisted High Quality Particle Rendering. Eurographics Symposium on Rendering (2009)

[7] McEwan, I., Sheets, D., Gustavson, S., Richardson, M.: Efficient computational noise in GLSL. Journal of Graphics Tools, vol. 16. iss. 2, pp. 85-94 (2012)

[8] http://www.futuremark.com/benchmarks/3dmark/all

[9] Fraedrich, R., Auer, S., Westermann, R.: Efficient High-quality Volume rendering pf SPH Data. IEEE Transactions on Visualization and Computer Graphics, vol. 16, no. 6. (2010)

[10] Guay, M., Colin, F., Egli, R.: Simple and Fast Fluids. GPU Pro, 2 (2011), pp. 433-444

[11] Guay, M., Colin, F., Egli, R.: Screen Space Animation of Fire. Proceeding of SIGGRAPH Asia 2011 Sketches (2011)

[12] Harris, M.J., Lastra, A.: Real-Time Cloud Rendering. EUROGRAPHICS 2001, vol. 20, no. 3 (2001)

[13] Horvath, C., Geiger, W.: Directable, high Resolution Simulation of Fire on the GPU. ACM Transactions on Graphics, 28, 3, Article 41 (2009)

[14] Judice, S.F., Coutinho, B.B.S., Gilson A. Giraldi, A.G.: Lattice methods for fluid animation in games. Computers in Entertainment (CIE) - SPECIAL ISSUE: Games archive, Volume 7, Issue 4, Article No. 56 (2009)

[15] Kallin, D.: Real Time Large Scale Fluids for Games. Proceedings of SIGRAD (2008)

[16] Kolb, A., Latta, L., Rezk-Salama, C.: Hardware-based Simulation and Collision Detection for Large Particle Systems. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (2004)

[17] Krüger, J., Westermann, R.: GPU simulation and rendering of volumetric effects for computer games and virtual environments. Proceedings of Eurographics (2005)

[18] Kun Zhou, Zhong Ren, Lin, S., Hujun Bao, Baining Guo, Heung-Yeung Shum: Real-Time Smoke Rendering Using Compensated Ray Marching. ACM Transactions on Graphics 27, 3, Article 36 (2008)

[19] Laine, S., Karras, T.: Efficient Sparse Voxel Octrees. IEEE Transactions on Visualization and Computer Graphics, vol. 17 , iss. 8, pp. 1048-1059 (2011)

[20] Lorensen, W.E., Cline, H.E. : Marching cubes: A high resolution 3D surface construction algorithm. Proceedings of the 14th annual conference on Computer graphics and interactive techniques (SIGGRAPH), pp. 163-169 (1987)

[21] Moser, M., Weiskopf, D.: Interactive volume rendering on mobile devices. Vision, Modeling, and Visualization (2008)

[22] Neyret, F.: Advected Textures. Proceedings of the Eurographics/Siggraph Symposium of Computer Animation, pp. 147-153 (2003)

[23] Nielsen, M.B., Söderström, A., Bridson, R.: Synthesizing waves from animated height fields. ACM Transactions on Graphics, Volume 32, Issue 1, (2013)

[24] Nielsen, M.B., Christensen, B.B.: Improved Variational Guiding of Smoke Animations. Proceedings of Eurographics 2010, vol. 29, no. 2 (2010)

[25] Ogniewski, J., Ragnemalm, I.: Realtime Particle System Simulation and Rendering in Embedded Systems, Proceedings of MCCSIS 2013. Online version available at: http://people.isy.liu.se/icg/jenso/

[26] Purkhet Abderyim, Tadahiro Fujimot , Norishige Chiba, Mamtimin Geni: Surface Reconstruction for Particle Simulation Using Level Set Method. The Journal of the Society for Art and Science, Vol. 6, No. 3, pp 154-166 (2007)

[27] Purevtsogt Nugjgar, P., Fujimoto, T., Chiba, N.: Markovtype velocity field for efficiently animating water stream. The Visual Computer, vol. 28, iss. 2, pp. 219-229 (2012)

[28] Rasmussen, N., Duc Quang Nguyen, Geiger, W., Fedkiw, R.: Smoke Simulation for Large Scale Phenomena. Proceedings of SIGGRAPH 2003, pp. 703-707 (2003)

[29] Raay, D., Sturt, C., Bossomaier, T.: Lattice Boltzmann Method for Real-Time Simulation of Lava Flows. Geometric Modeling and Imaging--New Trends, pp. 97-106, (2006)

[30] Rinchai Bunlutangtum, Pizzanu Kanongchaiyos: Enhanced view-dependent adaptive grid refinement for animating fluids. Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry (VRCAI'11), pp. 415-418 (2011)

[31] Rodriguez, M. B., Alcocer, P.P.V.: Practical Volume Rendering in Mobile Devices. Advances in Visual Computing (2012)

[32] Qizhi Yu, Neyret, F., Bruneton, E., Holzschuch, N.: Lagrangian Texture Advection: Preserving both Spectrum and Velocity Field. IEEE Transactions on Visualization and Computer Graphics, vol. 17, no. 11 pp. 1612-1623 (2011)

[33] Reeves, W. T.: Particle Systems Technique for Modeling a Class of Fuzzy Objects. ACM Transactions on Graphics, vol. 2, iss. 2, pp. 91-108 (1983)

[34] Selle, A., Rasmussen, N., Fedkiw, R.: A Vortex Particle Method for Smoke, Water and Explosions. SIGGRAPH 2005, ACM TOG 24, pp. 910-914 (2005)

[35] Tessendorf, J.: Simulating ocean water. SIGGRAPH 2001 Course notes (2001)

[36] Upchurch, E.M., Semwalk, K.S.: Dynamic cloud simulation using cellular automata and texture splatting. Proceedings of the 2010 Summer Simulation Multiconference , pp. 270-277 (2010)

[37] Venetillo, J.S., Celes, W.: GPU-based particle simulation with inter-collisions. The Visual Computer, vol. 23, iss. 9, pp. 851-860 (2007)

[38] Wei, X., Wei Li, Mueller, K., Kaufman, A.: Simulating fire with texture splats. IEEE Visualization (2002), pp. 227 - 234

[39] Wrenninge, M., Bin Zafar, N., Clifford, J., Graham, G., Penney, D., Kontkanen, J., Tessendorf, J., Clinton, A.: Volumetric Methods in Visual Effects. SIGGRAPH 2010 Course Notes (2010)

[40] Yuan, Z., Zhao, Y., Chen, F.: Incorporating stochastic turbulence in particle-based fluid simulation. The Visual Computer, vol. 28, iss. 5, pp. 435-444 (2012)

[41] Yubo Zhang, Zhao Dong, Kwan-Liu Ma: Realtime volume rendering using precomputed photon mapping. Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '12), p. 217 (2012)

[42] Yuri Kryachko: Using Vertex Texture Displacement for Realistic Water Rendering. GPU Gems 2, Addison-Wesley Professional (2005)

[43] Ogniewski, J.: Procedural Interactive Water in Memory- and Performance-Constrained Systems, Proceedings of NICOGRAPH 2014.

Jens Ogniewski received his M.S. in Electrical Engineering from Linköping University in 2007. After a brief period working in the industry he returned to Linköping University as a PhD student to continue his research. Apart from computer graphics, his research interests include video compression, parallel computing, and embedded systems. He is a (student) member of ACM, and a full member of the Society for Art and Science.

Figure 7 (next page): example images from the test-sequence including both refraction/absorption and interactions with the water surface, using a) (up) a single texture, b) (middle) the force-based approach and c) (down) Navier-Stokes, as in [10]. Screenshots taken from the Galaxy Note.
Figure 8 (below): zoomed-in detail images of figure 7: a) (left) single texture, b) (middle) force-based approach and c) (right) Navier-Stokes