Rendering of Water Drops in Real-Time

Ines Stuppacher*
Peter Supan†

Digital Media
Upper Austria University of Applied Sciences
Hagenberg / Austria

Abstract

In this paper we present a method for simulating physical behaviour and rendering of water drops in real-time. Our algorithm is based on processing water information stored into a texture resembling a height map. Thus calculation on the Graphics Processing Unit is possible. Texels on the height map retrieve how much of their current water amount remains and how much water is provided from texels above. The height map is smoothed with a blur filter before a normal map is created to perform lighting with a Fresnel-model.

Keywords: water drops, real-time, rendering, GPU

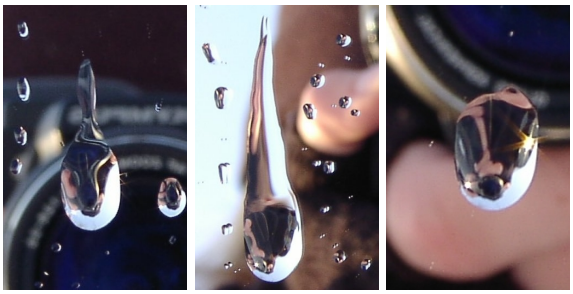
1 Introduction

Water with its many different forms of appearance demands different approaches for simulation. While rendering of water and other weather phenomena becomes increasingly realistic ([Fer04, cha. 1] [CdVLHM97]), there is still need for realistic water drops because small amounts of water behave strongly different to large ones. This can be ascribed to the physical forces of surface tension and viscosity [Dem03]. As the amount of water decreases, the impact of surface tension increases and results in the emergence of drops. These characteristics need to be reproduced by a simulation of water drops.

To find the crucial properties of water drops to create a credible simulation, a look at real drops has to be taken.Observing drops running down a surface, randomness seems to guide their movements on most occasions. But given a closer look many physical influences may be revealed.

Figure 1 shows pictures taken from real water drops displaying the most important characteristics of the drops.

- *Movement.* A drop only starts moving if it has a certain mass and size. Small drops usually just stay in place, but they may be swept along with moving ones. During the movement small droplets are emitted and thus the mass of the drop is reduced. As a result the motion slows down. Figure 1(a) shows the creation of such a small, remaining droplet at the end of the drop.



(a) (b) (c)

Figure 1: Pictures of water drops on a mirror showing characteristic features. (a) A remaining drop is formed. (b) A fast drop with a long tail. (c) A lensflare of the suns reflection.

- *Behaviour of flow.* The remainder of a fluid on the surface can be helpful to identify it. This denotes the viscosity [Dem03]. While water leaves only small droplets, sirup has a thick and viscous tail.

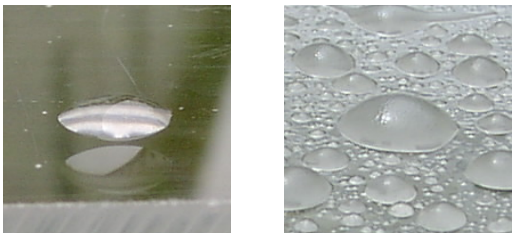
- *Shape.* The velocity of the drops movement influences its form. The faster a drop moves, the longer is its shape (cf. figure 1(b)). Due to surface tension small drops look like hemispheres [Dem03].

  The drops shape is also influenced by the adhesion of water on the surface. On hydrophilic surfaces the drops are flatter than on hydrophobic ones (cf. figure 2).

- *Reflexion and refraction.* Of course, the drops inherit the characteristics of water regarding lighting. They appear transparent because only very little light is absorbed by water.

  Due to the their shape drops act like convex lenses, with according minification and magnification in the refractions (cf. figure 1(a)).

*dm04026@fh-hagenberg.at
†dm04027@fh-hagenberg.at



(a) (b)

Figure 2: Drop on (a) hydrophilic and on (b) hydrophobic surface.

- *Glow and lens flares.* Reflections of the sun contain a vast amount of light. Hence a glow around it is produced due to the optics of the camera and/or the film (or CCD-chip). An example is visible in figure 1(c).

Finally a lot of information can be retrieved from the mass of a water drop. In connection with the affinity of the surface it is possible to determine radius, height and shape of the drop. Furthermore movement and velocity may be assigned.

The aim of our work is to introduce a new approach for realistic simulation of water drops in real-time. Moreover the implementation is done on the Graphics Processing Unit (GPU). The drops are limited to running down on a flat surface. Nevertheless the presented algorithm is extendible for application on arbitrary surfaces. Our goal was to create a credible simulation - not a physically correct one. We present an approach for application in real-time that may be extended in various ways.

In the next section we describe the related work. In section 3 the method to store water is discussed. Afterwards the algorithm and its implementation are introduced (section 4 and 5) followed by a presentation of its results (section 6). The last section summarizes our approach and future work.

2 Related Work

While a lot of research in the simulation of realistic water drops is available, real-time approaches are hard to find.

Wang et al. describe in [WMT05] a physically based simulation of water drops on arbitrary surfaces based on fluid simulations. Although the rendered droplets are very impressive and realistic, the proposed algorithm takes more than seven days to render an animation of 500 frames.

Other methods use metaballs or particles to achieve fast rendering. Despite this physically correct movement is hard to simulate with these approaches. Kaneda et al. show in [KZYN96] a way to render drops with particles. This method was upgraded by ATI in the "Toyshop" demo [Tat06]. The approach presented in this work is also based upon Kanedas work and thus resembles ATIs algorithm. In contrast to the algorithm of Kaneda et al. ours performs in real-time.

The research of Yang et al. in [YZZ04] concentrates only on the rendering of water drops on glass in real-time, movement is not simulated. They assume that a drop resembles a lense. Everything behind the drop is distorted accordingly.

Rathner describes in [Rat02] an erosion simulation storing water amounts in a grid. The calculation of the cells water height depends on its neighbours values.

3 Water Model

The drops of the presented algorithm are restricted to movement on a surface. Thus their data can be reduced to their position on the surface (x,y) and their height h (cf. figure 3).

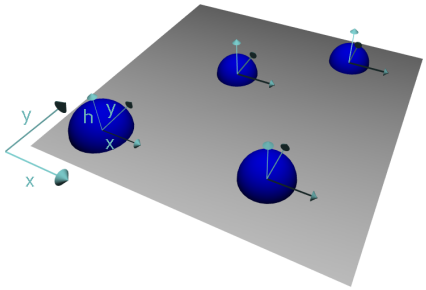


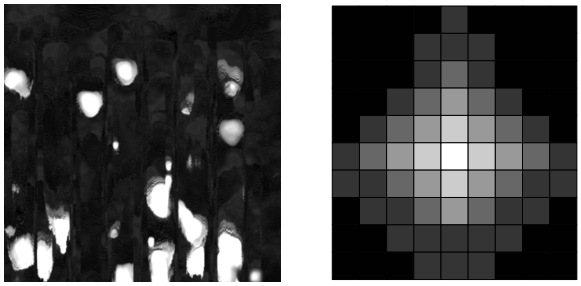
Figure 3: Drops on a surface can be characterised with their position (x,y) and their height h.

As the required data are the same for all drops and are used similarly to process the behaviour of the drops, a particle system would be appropriate to handle it [AMH02, cha. 8.5]. Those systems are particularly suitable to administer phenomena consisting of many small components behaving in the same manner. Latta shows in [Lat04] a particle system implemented on the GPU.

We chose to assemble a drop out of many small droplets. These droplets are atomic and therefore the smallest amount of water that may occur in the system. How much water a drop contains is identified by the amount of droplets forming it. The final shape of the drop is determined by the arrangement of the small droplets. The more droplets stick together (the greater the water mass), the sooner they start to move. Remaining water is created as some droplets move more slowly and are left behind.

But as the water particles are to be computed on the GPU their data has to be stored appropriately. As a drop is influenced by its neighbours, it needs a possibility to access those. Hence a 2d-grid describing the surface is created. It is saved as a texture on which the water drops flow. The idea of saving the surface into a grid and calculating the water drops with it has been introduced by [KZYN96],

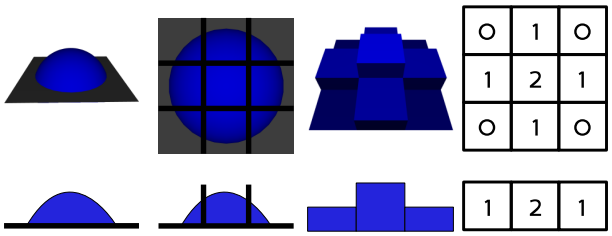
whereas Kaneda et al. used Bezier-patches. Each texel in the created texture stores the amount of water currently being located upon it. Thus it matches a height map (cf. figure 4). Black areas indicate no water, whereas white areas state large amounts of water and therefore an according height. A grey value represents water, the lighter the color the more water is stored in the texel.



(a) (b)

Figure 4: (a) Height map indicating water drops. (b) Sketch of a single drop in the texture.

Figure 5 depicts how a water drop is converted into such a height map. The drop is rasterized due to the resolution of the texture. With a big texture the single drops may consist of many droplets. Also accordingly small drops may be processed and displayed. Each cell determines how much water it contains. The values are quantised to create numerical values as seen in figure 5(d). Actually the values are stored as floating-point values as heights greater than $1.0f$ may occur, that could not be stored with fixed-point textures. So in the end a texture is created where each cell contains the amount of water upon it and thus its height.



(a) (b) (c) (d)

Figure 5: (a) The water drop. (b) A 2d-grid is laid on the drop. (c) The rasterized drop. (d) Numerical values resembling the droplets heights.

This technique enables each water particle to access its neighbours in a fragment shader. Thus it is able to retrieve how much water is provided from its neighbours to compute the water movement. The following section explains the details of the algorithm.

4 The Algorithm

By using the texture storing the water amounts the bulk of the algorithm can be processed on the GPU using fragment shaders. The CPU is left with storage and preparation of parameters.

4.1 The Pipeline

The simulation of water drops can be organized into two parts: the physical simulation of the water movement on one hand and the lighting of the resulting drops on the other hand. The pipeline of the proposed algorithm is displayed in figure 6.

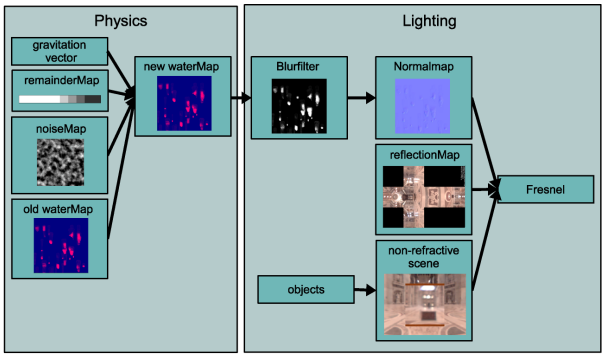


Figure 6: The algorithms pipeline can be divided into the two main parts physics and lighting.

The physics part needs a *gravitation vector* as input to know the direction of gravitation in world coordinates. This is necessary to move the drops downwards even if the object is rotated. The texture with the "old water drops" from the last frame (*old waterMap*) is also needed to calculate the new water amounts and thus the movement. The *remainderMap* determines how much of an amount of water flows down. Using the *noiseMap* this ratio is diversified. With these textures the movement of the drops is created, resulting in the *new waterMap*.

To smooth the output of the physics calculation a *blur filter* is used. The lighting is done by creating a *normal map* from this smoothed height map. Using these normals reflections and refractions are calculated and combined with a Fresnel term. The *reflectionMap* resembles a cube map used to retrieve the reflection color values. For rendering of refractions the method of Sousa described in [Pha05, cha. 19] is used. Here an environment map is created including all non-refractive objects. As these are common techniques, this article will not go into great detail about the lighting part of the pipeline.

In the following sections the parts of the pipeline are explained in more detail.

4.2 Water Movement

As already discussed the water particles are stored and processed as a 2d texture. Thus all calculations have to be done in tangent space [ZDA04, p. 400]. Accordingly all necessary vectors have to be transformed from world space to object space and finally to tangent space.

4.2.1 Direction

As a texel needs to know where ”upwards” is located to query the amount of water it receives from there, a gravitation vector $\vec{G}$ is created in the application. This vector describes the direction of gravitation in world space. If an object is rotated, the direction of $\vec{G}$ changes relatively to the object spaces centre (cf. figure 7). As all calculations have to be done in tangent space, $\vec{G}$ has to be transformed into it with the matrix

$$\mathbf{T} = \begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{pmatrix}, \tag{1}$$

where T is the tangent, B is the binormal and N is the normal of the current vertex. Now all other directions may also be computed using the transformed vector (cf. figure 8).

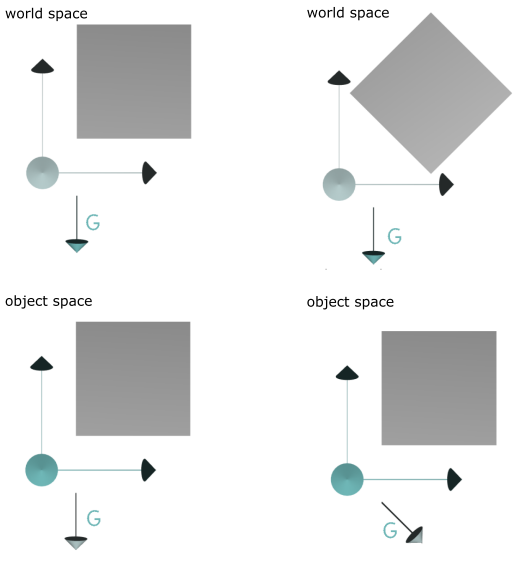


Figure 7: The upper row shows the scene in world space while to lower row shows it in object space. The direction of the gravitation vector $\vec{G}$ is changed in object space as the object is rotated.

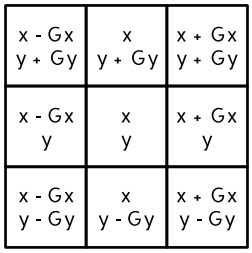


Figure 8: All eight neighbour directions can be calculated using the gravitation vector $\vec{G} = (G_x, G_y)^T$.

4.2.2 Speed

The speed of the drops is influenced by the inclination of the surface. The more inclined the surface, the faster the movement. The inclination of the surface is described relative to the gravitation vector $\vec{G}$ (cf. figure 9) and given by the magnitude of the angle α between $\vec{G}$ and the surface. $\cos(\alpha)$ may then be used to weight the speed of the drops movement by multiplying the gravitation vector with it

$$\begin{pmatrix} G'_x \\ G'_y \end{pmatrix} = \begin{pmatrix} G_x \\ G_y \end{pmatrix} \cdot \cos(\alpha), \tag{2}$$

where $\cos(\alpha)$ describes the angle between $\vec{G}$ and the surface.

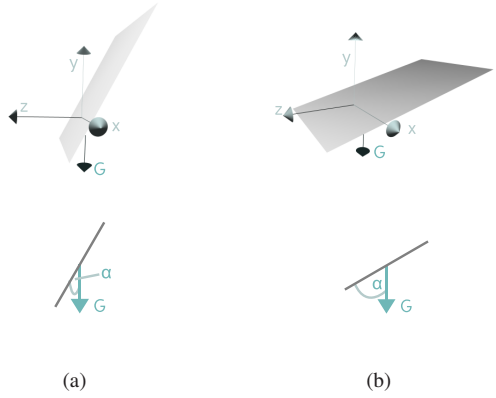


Figure 9: If the angle α between surface and gravitation vector $\vec{G}$ is small, the water moves faster.

4.2.3 Water Amount

Section 3 describes that each texel on the height map saves the amount of water upon it. If this amount is greater than a threshold L, the water starts moving. Hence the current texel reduces its water amount. But if the water amount of the above texel is also greater than L some of its water flows down to the current texel. Figure 10 shows the computation with four sample texels.

The percentage flowing down is determined by the `remainderMap`. The amount of water is used as texture coordinates to access this texture. The retrieved

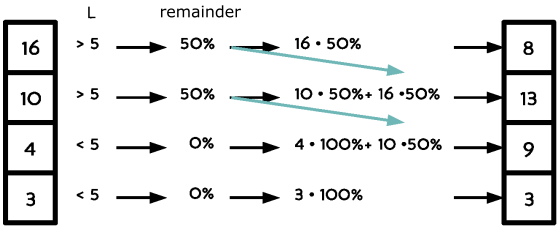


Figure 10: Water amounts of the last frame are on the left side, while those of the current frame are on the right. At the beginning the amount of water staying on the texel is retrieved. Afterwards the water provided from above is added.

value tells how much water remains on the texel. The `remainderMap`, like the one in figure 11, is created with an image editing program. White indicates that 100% of the water stay at the current texel. The darker the remainder value, the more water flows downwards.

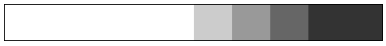


Figure 11: Example of a `remainderMap`.

To add randomness to the simulation a noise texture (e.g. Perlin noise), like the one in figure 12, is used. It is also created with an image editing program and then fed into the simulation. Of course, it could also be generated at runtime. Only a part of the texture is used in the calculation, but this cut-out is shifted across the texture over time. The retrieved noise values are used to alternate the remainder of the drops. A texel now calculates its remainder value and multiplies it with the value from the `noiseMap`.

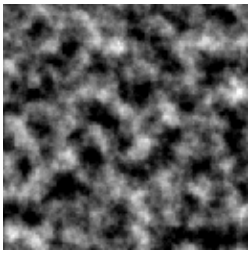


Figure 12: Example of a `noiseMap`.

Thus the formula to compute a texels current amount of water is created as

$$\begin{aligned} \text{newWater}(\vec{T}) \quad \leftarrow \quad & \text{water}(\vec{T}) \cdot \\ & \text{remainder}(\text{water}(\vec{T})) \cdot \\ & \text{noise}(\vec{T}) + \\ & \text{water}(\vec{T}+\vec{G}') \cdot (1.0 - \\ & \text{remainder}(\text{water}(\vec{T}+\vec{G}')) \cdot \\ & \text{noise}(\vec{T}+\vec{G}')) \end{aligned} \tag{3}$$

where $\vec{T} = (T_x, T_y)^T$ texture coordinates,
$\vec{G}' = (G'_x, G'_y)^T$ gravitation vector multiplied with $\cos(\alpha)$,
$\text{remainder}(\text{water})$ remaining amount of water,
$\text{water}(\vec{T})$ amount of water at $\vec{T}$ and
$\text{noise}(\vec{T})$ noise value at $\vec{T}$.

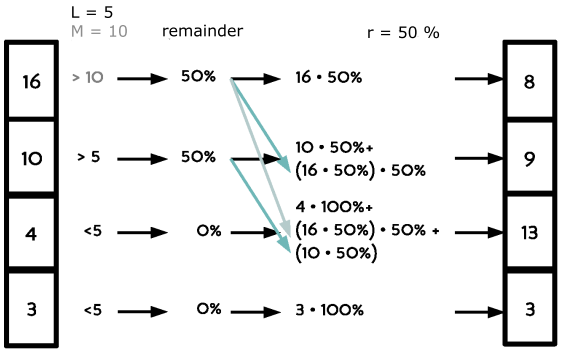


Figure 13: Calculation of water movement taking into account texels two above the current one. Values from the last frame are on the left side, while the new ones are on the right side.

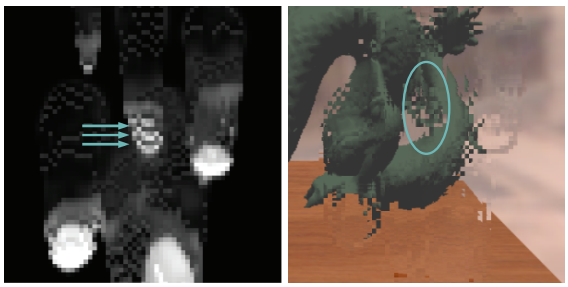
As stated in section 1 the velocity of the drops is influenced by their mass. To be able to move very large amounts of water the current texel also retrieves water from the texel above its above texel. If the amount of water from the texel two above is greater than the threshold M it contains much water. Figure 13 shows how the calculation is changed. To prevent all of the water from moving directly into the current texel it is weighted with a coefficient r. Therefore water from Texel A flows to Texel B and C. The formula 3 has to be extended accordingly. Further generalizations to calculate with more texels above the observed one have not been implemented.

4.3 Water Lighting

The results of the previous simulation of water movement depend heavily on the chosen parameter values. Due to the processing with thresholds bad distributions of the water amounts may occur. Those have to be smoothed with a blur filter.

For example a cell being slightly higher than the threshold is processed and afterwards only little water remains there. A neighbouring cell may be slightly smaller than the threshold value and thus not processed. Compared with the processed cell much more water remains in the unprocessed one. Figure 14(a) shows the resulting lines in the height map. As these lines lead to unaesthetic normals, the lighting also looks bad (cf. figure 14(b)).

To compensate for this effect the average of the current texel and its eight neighbours is calculated and saved into a new texture. Figure 15 shows the cut-out of figure 14 after application of the blur filter.



(a) (b)

Figure 14: Because of the processing with thresholds (a) bad distributions of water amounts and thus (b) bad lighting may occur.

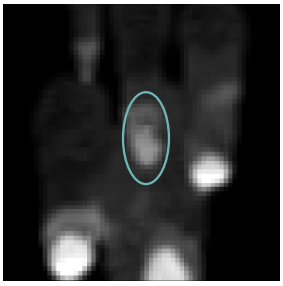


Figure 15: Height map after blurring.

Using the blurred height texture a normal map can easily be created [ZDA04, p. 397]. Afterwards lighting is performed with the Fresnel model [FK03, p. 189] combining reflections and refractions. This actually is a technique of environment map bump mapping [Eve03]. For reflections a cube map is used, while for refractions a dynamic 2d environment map is created. Therefore all non-refractive objects are rendered in an extra pass and saved to this texture. Like in [Fer04, cha. 19] refractions are computed from this texture and the normal map.

5 Implementation

The implementation was done in C++ with OpenGL and *Cg* shading language. As water flows from one texel to the next, values greater than $1.0f$ may occur. Thus it is necessary to use floating-point textures to save the water amounts.

The implementation of the pipeline (see section 4.1) was done in five render passes (cf. figure 16). At the beginning all non-refractive objects are rendered as an environment map for refraction. The second pass includes the water movement simulation, the generation of the texture with the new water heights. Afterwards this texture is blurred. The fourth render pass calculates normals from the blurred height map. Finally the background scene and the water drops are displayed in the framebuffer.

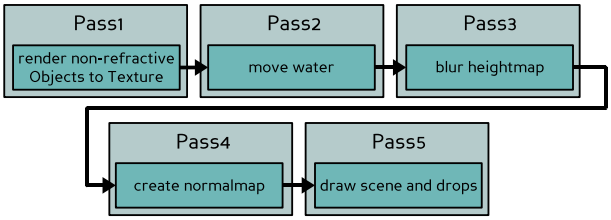
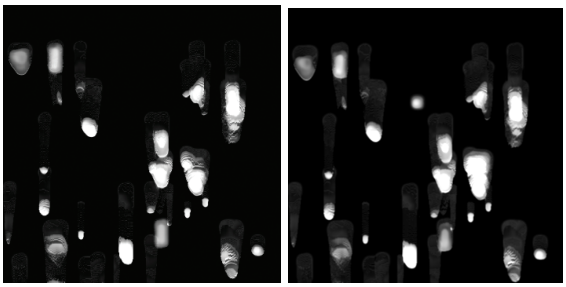


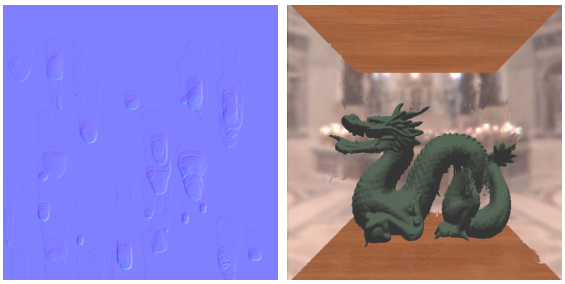
Figure 16: The render passes of the algorithm.

6 Results

During rendering different modes may be activated to display intermediate results. Figure 17 shows the different steps resembling the render passes of the algorithm.



(a) (b)

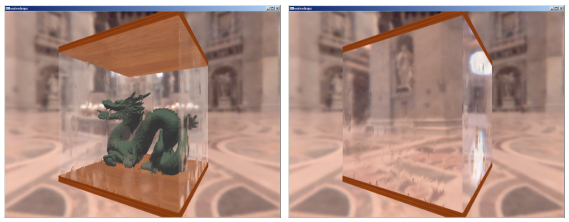


(c) (d)

Figure 17: (a) Height map of the water. (b) Blurred height map. (c) Normal map. (d) Final rendering with 3DS-model in the background.

The application renders a box with side frames made of glass or mirror (cf. figure 18). On these sides the water drops are flowing down. Inside of the cube different 3DS-models may be displayed. Of course, they are only visible if the side frames are rendered as glass.

Despite of five render passes the algorithm performs in real-time on an nVidia GeForce 7800. Table 1 lists the frame rates with different models displayed. The 3DS-models are quite detailed and thus influence the frame rate. Rendering the most complex models the application performs 25% slower.



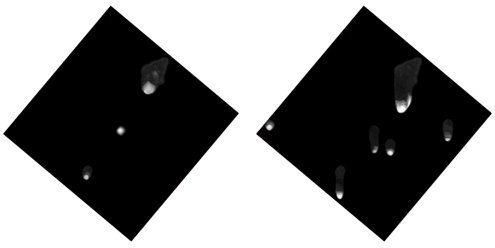
(a) (b)

Figure 18: In the application water drops may be displayed on glass or mirrors.

Scene	Vertices	Faces	fps
Mirror	24	6	160
Glass with Dragon	93.044	100.000	120
Glass with Buddha	92.299	100.000	120
Glass with Bunny	36.873	69.666	125
Glass with Planck	25.445	50.801	146

Table 1: Frame rates with different models displayed.

The cube may be rotated around all axes while the drops still move downwards (cf. figure 19). At the end of the texture the water amounts get smaller and become invisible. With the current algorithm they may not move on to another surface. But ideas for an extension of the method for usage with more surfaces and arbitrary objects have already been created. However the drops may not drop off the surface into air (cf. figure 20). But as the normal of the surface points more and more upwards the movement is slowing down and finally stops. The drops stay on the surface until it is pointed downwards again.



(a) (b)

Figure 19: Drops are always flowing in the direction of the gravitation vector.

The number of drops on the surface does not influence performance, because all calculations are always done for each texel. Hence it is not relevant if water is on a texel or if the water amount of the texel is changed. However the frame rate is influenced by the size of the height map. Also the resolution of the drops depends on the textures

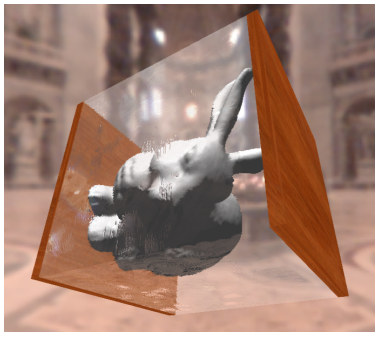
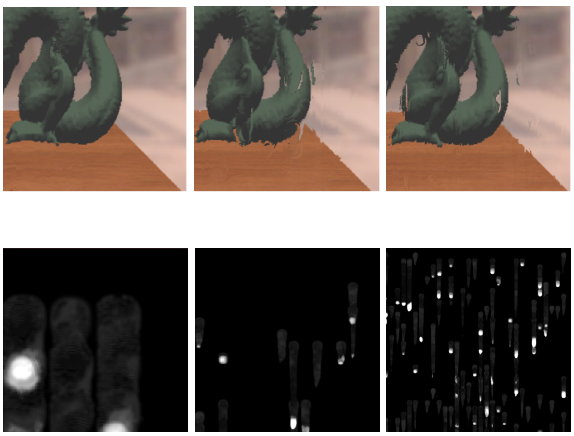


Figure 20: Drops on the lower side should actually fall off the cube.

resolution. To achieve higher quality a larger texture has to be used. Frame rates for different texture resolutions rendering a mirror are shown in table 2. For all previous screenshots a 512x512 texture was used. Figure 21 shows screenshots with three different texture resolutions. The same drops have been blended onto them. The smaller texture doubles the frame rate while it is only a third with the larger texture. A levels of detail technique may be implemented to adapt the texture resolution dynamically to the distance between object and camera. So an object near the camera will use a higher resolution height map than one far away.

texture resolution in px	fps
128 x 128	480
512 x 512	160
1024 x 1024	65

Table 2: Frame rates with different texture resolutions.



(a) 128x128 (b) 512x512 (c) 1024x1024

Figure 21: Due to the resolution of the height map the frame rate changes.

In contrast to existing work our algorithm performs in

real-timeand is calculated on the GPU. Only ATIs approach also runs in real-time, but was published after completion of our work.

Most observers found the results quite plausible, whereas they cannot keep up in comparison with real water drops (cf. colorplate). An important point for criticism is that too few small drops stay on the surface. Furthermore the drops do barely change their direction of movement. In comparison with real drops they display too little accident. But as in nature this randomness is usually based upon physics, these physical interactions should be added to the simulation.

7 Conclusions and Future Work

Physical forces affecting large amounts of water differ from those acting on water drops. Consequently different techniques have to be used to achieve credible simulations. In this article a new approach for rendering water drops in real-time was introduced. As the implementation is done on the GPU the water particles are stored into a texture. Thus it is possible to calculate water movement in a fragment shader. To perform lighting with the Fresnel-model a normal map is created.

The new algorithm is promising but still needs some enhancements to achieve greater realism. Currently water drops are limited to flow on flat surfaces, which can be overcome with some modifications. It would be necessary to lay out the normals of the whole geometry onto a flat surface. Problems may occur as not all neighbouring surfaces are connected on the 2D layout.

Future work will include saving the direction of movement for each texel. Thus texels will also consider their neighbours' direction of movement when calculating their own. Hence it will be possible that texels sweep away their neighbours. A similar approach has been taken by ATI in the "Toyshop" demo [Tat06].

Lighting may be improved by a glow effect. Through the use of high dynamic range (HDR) better reflections could be achieved [Deb02]. Additionally the height map can be used as shadow map so that drops cast shadows on objects behind them [Tat06].

8 Acknowledgements

The authors thank Michael Haller for coaching and hardware. Models of dragon, Buddha, bunny and Max Planck were provided by Stanford University Computer Graphics Laboratory.

References

[AMH02] Tomas Akenine-Möller and Eric Heines. *Real-Time Rendering*. A K Peters, 2. edition, 2002.

[CdVLHM97] Jim X. Chen, Niels da Vitoria Lobo, Charles E. Hughes, and J. Michael Moshell. Real-time fluid simulation in a dynamic virtual environment. *IEEE Computer Graphics and Applications*, 17(3):52–61, 1997.

[Deb02] Paul Debevec. Image-based lighting. *IEEE Computer Graphics and Applications*, 22:26–34, March–April 2002.

[Dem03] Wolfgang Demtröder. *Experimentalphysik 1: Mechanik und Wrme*. Springer-Verlag, 3. edition, 2003.

[Eve03] Cass Everitt. Reflective bump mapping. Technical report, NVIDIA Corporation, Santa Clara, California, USA, 2003.

[Fer04] Randima Fernando. *GPU Gems*. Addison-Wesley, 2004.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial*. Addison-Wesley, 2003.

[KZYN96] Kazufumi Kaneda, Yasuhiko Zuyama, Hideo Yamashita, and Tomoyuki Nishita. Animation of water droplet flow on curved surfaces. In *Procedings PACIFIC GRAPHICS '96*, pages 50–65, 1996.

[Lat04] Lutz Latta. Building a million particle system. In *Game Developers Conference 2004*, August 2004.

[Pha05] Matt Pharr. *GPU Gems 2*. Addison-Wesley, 2005.

[Rat02] Leonhard Rathner. Landschaftserstellung und -erosion in der computergrafik. Master's thesis, Fachhochschule Hagenberg, Medientechnik und -design, Hagenberg, Austria, September 2002.

[Tat06] Natalya Tatarchuk. Artist-directable real-time rain rendering in city environments. Technical report, ATI Research Inc., 3D Application Research Group, Marlborough, Massachusetts, USA, 2006.

[WMT05] Huamin Wang, Peter J. Mucha, and Greg Turk. Water drops on surfaces. *ACM Trans. Graph.*, 24(3):921–929, 2005.

[YZZ04] Yonggao Yang, Changqian Zhu, and Hua Zhang. Real-time simulation: Water droplets on glass windows. *Computing in Science and Engineering*, 06(4):69–73, 2004.

[ZDA04] Stefan Zerbst, Oliver Düvel, and Eike Anderson. *3D-Spiele-Programmierung*. Markt + Technik, Deutschland, 2004.