

Accurate Water Simulation for Visibility in Driving

J. Marcó¹, G. Patow¹, E. Reinhard² and X. Pueyo¹

¹GGG - UdG - Spain

²Dept. of Computer Science - University of Bristol - UK

Abstract

In this paper we will present a system for the real-time simulation of rain falling on a windscreen. Our model incorporates external forces like gravity and wind, and also simulates how the rain gets removed from the windshield by the windscreen wipers. The algorithm is based on the Lattice-Boltzmann Method (LBM), which consists of a regular lattice that represents the fluid in discrete locations, and equations to simulate its flow. We perform all the computations of the LBM on graphics processors for accelerating the calculations. We render the results of the LBM simulation using an approximate image-space approach for interactive refraction, which allows the computation of refractions of a distant environment through two interfaces.

1. Introduction

Driving simulators are frequently used to help drivers improve their skills. Even nowadays there are professional drivers who spend more hours training with a simulator than with a real car, due to its cost-effectiveness. There are several components to driving simulation, one of these being how the driver experiences the environment through the windscreen. Moreover, the driver will appreciate the environment very differently in adverse weather conditions, like heavy rain.

The simulation of rain is a problem with huge importance for road safety, which until now does not have an accurate solution. Clearly such a simulator must operate in real time, and achieve a very high degree of physical and perceptual realism.

1.1. Objectives

The main objective of this work is to produce accurate and real-time simulations of rain, and simulate its behaviour on a windscreen. This includes the simulation of droplets and of heavy rain in general, the modeling of external forces, like gravity or wind, and a model for how rain gets wiped off by the windscreen wipers.

1.2. Overview

To achieve these objectives we will use techniques based on the Lattice-Boltzmann Method (LBM), which consists of a

regular lattice that represents the fluid in discrete locations, and simulates how the fluid flows on it. At high spatial resolutions, it can be demonstrated that these methods converge into the continuous equations of Navier-Stokes, which are the most important to describe the macroscopic behavior of a fluid.

We use a 2D grid, which allows us to represent the fluid as a height field with respect to the surface of the windscreen. We incorporate external forces, like gravity and wind to make the simulation as realistic as possible. Also, we add windscreen wipers to the simulation. To accelerate all calculations we implement our system on current graphic hardware.

2. Previous Work

Fluid simulations tend to be based on one of two main approaches, namely empiric models and accurate simulations. The first ones are usually based on a simplified model that treats water as particles under newtonian laws. These particles flow through a grid placed over the object's surface. This way, particles are moved by wind or gravity, but collisions are avoided by accumulating water in the grid cells and using the resulting water quantity for further calculations. The results, although not accurate, produce believable animations [EHJGP08,KKY93], and have recently been augmented with the simulation of rain streaks and splashes [GN06,GKN07].

Full-blown physical simulations include molecular mechanics, direct Navier-Stokes simulations and the Lattice-Boltzmann Method. Molecular simulations constitute a family of computer simulation techniques that involve the exact solution of statistical mechanics problems [Sad99]. In these approaches, which include molecular dynamics (MD) and Monte Carlo (MC) techniques, the properties of the studied systems are determined solely by the evaluation of inter-molecular forces and energies.

Molecular dynamics utilises Newton's equations of motion, while the Monte Carlo techniques use a statistical mechanics approach, using the concept of configuration space [BS01]. Unfortunately, these techniques can only be used to simulate a fluid up to a few millimeters scale.

The equations of motion of an incompressible, Newtonian fluid (usually called Navier-Stokes equations [Bri08]) have been traditionally used for accurate simulations, but their complexity makes them infeasible for real-time simulations of large volumes or surfaces [WMT05].

3. Lattice Boltzmann

The lattice Boltzmann method derives from the Boltzmann equation [Wag08, LFWK05], which describes the behavior of gas on a microscopic level using kinetic theory. First proposed by [MZ88], the lattice Boltzmann method replaced the boolean particle number in a lattice direction with the density distribution function to reduce statistical noise. The practical viability of simulating in three dimensions came with [HJ89], while [QdL92] suggest enhanced collisions for the LBM that allow simulations with low viscous fluids. They eliminate collisions from the LBM so that only the consequence of collisions matters.

A final improvement to the collision operator is known as the Bhatnagar-Gross-Krook approximation [QdL92, CCM92]. This version of the LBM is known as the lattice-BGK model (LBGK) and provides a single time relaxation. The LBGK is the most popular LBM used today due to its simplicity and efficiency.

The LBM is inherently compressible. Consequently, it models the compressible Navier-Stokes equation. Fluid compressibility is a main feature of the LBM and is what gives it a performance advantage over other methods. However, [HL97] recognized there is also a need for incompressible fluids and thus introduced an incompressible variant of the lattice Boltzmann model. One limitation that comes with the incompressible LBM is that the fluid speed must be kept low in order to minimize the compressibility effect. In our case, as we are interested in modeling water, we will make use of this approximation.

3.1. LBM details

The lattice Boltzmann methods work on a lattice. Several variations of the LBM exist, and are named $DXQY$, where X

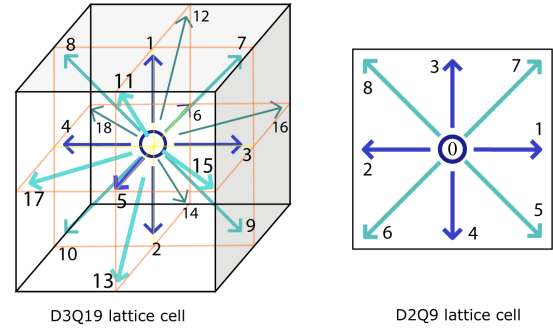


Figure 1: Left: the D3Q19 model. Right: D2Q9 Model

is the dimension and Y is the number of lattice velocities or vectors, as shown in Figure 1. All cells are updated at each time step by simple rules, taking into account the state of the surrounding cells. In our project we use the D2Q9 model.

A lattice vector is referred to as e_i where i is the lattice vector number. In our model the lattice vectors are $e_0..e_8$. At each lattice site $\vec{x}$ and time t , fluid particles moving at arbitrary velocities are modelled by particle distribution functions $f_i(\vec{x}, t)$. Each $f_i(\vec{x}, t)$ is the expected number of particles moving along a lattice direction vector e_i . Note that the particles are allowed to move only along the lattice velocity vectors. The magnitude of the velocity vectors e_1 through e_4 is 1 lattice unit per time step. The magnitude of velocity vectors e_5 through e_8 is $\sqrt{2}$ lattice units per time step. Also, the vector with index 0 has zero length, and indicates particles that are not moving anywhere in the next time step, although some of them may be accelerated due to collisions with other particles. Thus, the number of resting particles may change at each time step.

From the particle distribution functions two important physical values can be calculated. By summing up all 9 distribution functions, the density for the volume of this cell can be calculated, assuming that all particles have the same mass of 1. As the distribution functions contain the number of particles moving in a certain direction for each cell, the sum of all particles in a single cell is its density:

$$\rho = \sum_{i=0}^8 f_i \quad (1)$$

Another important attribute for each cell is the velocity and overall direction in which the particles of one cell move:

$$\vec{v} = \frac{1}{\rho_0} \sum_{i=0}^8 f_i \vec{e}_i \quad (2)$$

A simulation consists of two steps, namely *streaming* and *collision*, that are repeated for each time step. The streaming step is the simplest one, and consists of moving particles from one cell to another.

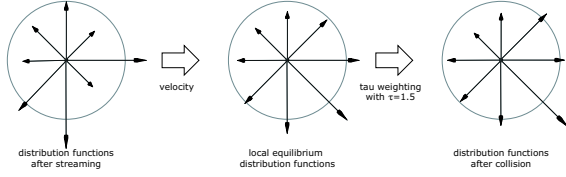


Figure 2: Collision step

In the collision step, particles arrive at a cell and collide with other particles, as shown in Figure 2. The collision step does not change the density or velocity of a cell, but only changes the cell's particle distribution.

To model this behaviour, the equilibrium distribution function, f_i^{eq} and new distribution functions must be calculated. For incompressible flows the following equilibrium distribution function is a good choice [HL97]:

$$f_i^{eq}(\rho, \vec{v}) = w_i \left[\rho + \rho_0 \left(\frac{3}{c^2} (\vec{e}_i \cdot \vec{v}) + \frac{9}{2c^4} (\vec{e}_i \cdot \vec{v})^2 - \frac{3}{2c^2} (\vec{v} \cdot \vec{v}) \right) \right] \quad (3)$$

The weights w_i depend on the length of the velocity vector.

The streaming and collision models are usually combined into one formula, which is known as the Lattice-Boltzmann equation:

$$f_i(\vec{x} + \vec{e}_i, t + 1) - f_i(\vec{x}, t) = \omega(f_i^{eq}(\rho, \vec{v}) - f_i(\vec{x}, t)) \quad (4)$$

where $\omega = 2/(6\nu + 1)$. The left hand side of this equation accounts for the stream step. The right hand side is a combination of the current distribution function and the local equilibrium.

The simulation of the LBM then proceeds by repeating the following steps: First, the density distribution is computed according to Equation 1. Then, velocities are computed using Equation 2, following by the equilibrium distribution given by Equation 3. Finally, the distribution is updated using Equation 4.

4. Accurate Water Simulation

The algorithms we employ in our simulations account for rainfall on a windscreen, the effect of wind, and the removal of water from the windscreen by windscreen wipers. Each of these components is discussed in the following sections.

4.1. Adapting the LBM equations to gathering operations

The LBM equations are normally written as *scattering* equations, i.e. they write to multiple locations. On the other hand, fragment shaders only offer *gather* operations, which means that data can be read from anywhere in memory, but writing is limited to a fixed location.

To adapt the LBM equations for implementation using

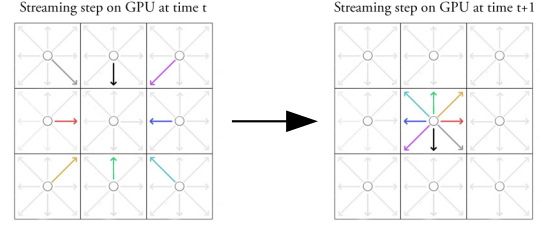


Figure 3: The streaming step on the GPU, where we read the particles distribution from the cell neighbors.

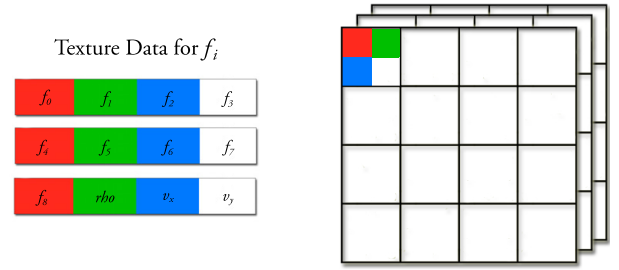


Figure 4: Particle Distribution Function, velocity and density storage in 3 textures

fragment shaders, we change the equations so that every value written depends on the values read from neighbouring cells, as shown in Figure 3. We can rewrite the equations by replacing $\vec{x}$ with $\vec{x} - \vec{e}_j$. This leads to a new set of equations that model density, velocity, and collision and streaming, respectively:

$$\rho^* = \sum_{i=0}^8 f_i(\vec{x} + \vec{e}_i, t) \quad (5)$$

$$\vec{v}^* = \frac{1}{\rho_0} \sum_{i=0}^8 f_i(\vec{x} + \vec{e}_i, t) \vec{e}_i \quad (6)$$

$$f_i(\vec{x}, t + 1) - f_i(\vec{x} - \vec{e}_j, t) = \omega(f_i^{eq}(\rho^*, \vec{v}^*) - f_i(\vec{x} - \vec{e}_j, t)) \quad (7)$$

4.2. Implementation of the LBM

As discussed in Section 3, the lattice-Boltzmann method requires the program to keep track of 9 particle distribution functions f_i , as well as density ρ , and velocity $\vec{v}$ for each cell. We therefore need to store 12 floats per cell and per time step. This amount of data can be accommodated by 3 RGBA floating point textures, as shown in Figure 4.

An LBM implementation consists of four operations: streaming, boundary detection, velocity and density computation, and collision handling (Figure 5). All these operations can be implemented in one fragment shader, combined with



Figure 5: Operations for computing the LBM

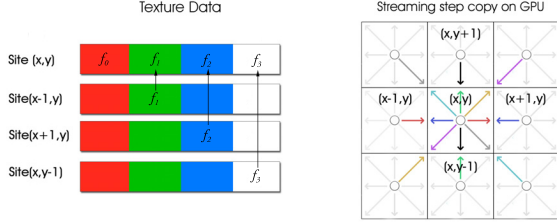


Figure 6: Streaming step on the GPU

the "ping pong" technique and Multiple Render Targets, as follows:

- **Streaming** The streaming step for non-boundary cells is accomplished by swapping channels in a texture. This operation is straightforward to carry out in a fragment shader: we just need to read the data stored in the correct neighbour. For example, to stream f_1 and f_2 (the red and purple arrows in Figure 6), the following Cg code is used:


```
fNew1 = tex2D(texture1,
               coords + float2(-1.0,0.0)).g
fNew2 = tex2D(texture1,
               coords + float2(1.0,0.0)).b
For f0, we simply use: fNew0 =
tex2D(texture1,coords.r).
```
- **Boundary detection** In our case, we need to implement a boundary condition that allows the water to leave at the boundaries. Figure 7 shows that all the particles in a boundary cell will be removed from the simulation in the next step, effectively implementing water flowing off the screen.
- **Velocity and density computations** After streaming all the particle distribution functions, we calculate the new velocity and density using a direct implementation of equations 1 and 2.
- **Collisions** The final step in the fluid simulation is calculating the new distribution functions after collision han-

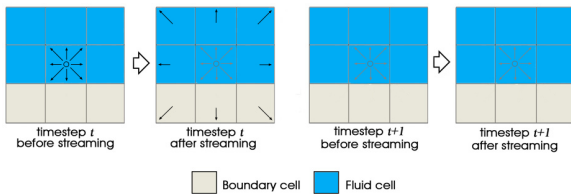


Figure 7: Boundary conditions implemented

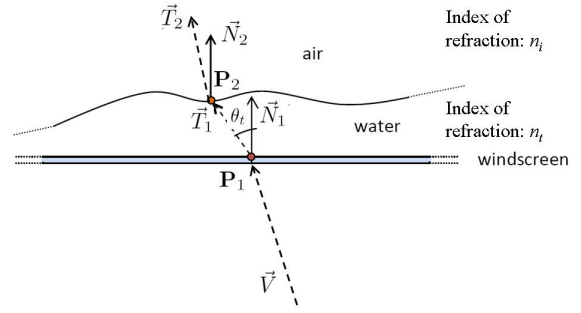


Figure 8: The driver is inside the car, so vector $\vec{V}$ hits the surface at P_1 and refracts in direction $\vec{T}_1$ based upon the incident angle θ_i with the normal $\vec{N}_1$. Physically accurate computations lead to a further refraction at P_2 .

dling. The shader uses the incompressible variant of the LBM, given by Equation 3. First we calculate the equilibrium distribution function with equation 4 to find the new distribution functions. Then the shader saves the new f_i into the texture for use in the next time step.

4.2.1. Negative densities

During an LBM simulation, some cells may end up with zero density. As a result, and after applying the equilibrium distribution function (Equation 3), there may be cells with negative densities. It is possible to add a parameter α to the advection term to reduce advection in areas with low densities [CT05], and thereby avoid negative densities. The new equilibrium distribution function is then given by:

$$f_i^{eq}(\rho, \vec{v}) = w_i \left[\rho + \alpha \rho_0 \left(\frac{3}{c^2} (\vec{e}_i \cdot \vec{v}) + \frac{9}{2c^4} (\vec{e}_i \cdot \vec{v})^2 - \frac{3}{2c^2} (\vec{v} \cdot \vec{v}) \right) \right] \quad (8)$$

The variable α is defined by the well known *Smoothstep*(0, λ , ρ), where λ is a user specified value: *Smoothstep*($\cdot$) will set α to 0 in cells with no water, causing no advection to occur. Otherwise α will be greater than zero. We have found that $\lambda \in [0.1, 0.6]$ to work well.

4.3. Rendering

The output of the LBM simulations are subsequently rendered. The combination of windscreen and a thin layer of water which can vary in thickness dependent on the amount of water in each cell, requires rendering with multiple refractions (Figure 8). We have found Wyman's approach to be suitable [Wym05], so that our rendering pipeline is as outlined in Figure 9.

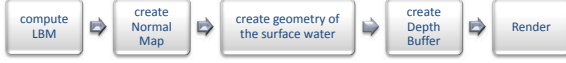


Figure 9: The main operations for rendering

4.3.1. Modelling the windscreen

Our windscreen model has a curvature and an angle of inclination α with respect to the X/Z plane. To take curvature into account, we store the windshield surface normals in a separate texture. These normals are defined in object space.

4.3.2. Creating a smoothed normal map of the water surface

To create a normal map of the water surface we need its geometry. In fact, LBM gives the number of particles of water in each cell. We take this quantity to be proportional to the height of water in each cell. Thus, the LBM method effectively produces a height field on the windscreen, from which we compute a normal map using a *Sobel Filter* [Zim07].

4.3.3. Building the geometry of the water surface

At this point we want to create the geometry of the water surface. To do that we need three things: the height map from the LBM, a mesh of triangles which lie on the windscreen and the smoothed normal map of the water on the windscreen. We create a mesh with two triangles for each cell, which is stored in a vertex buffer object (Figure 10).

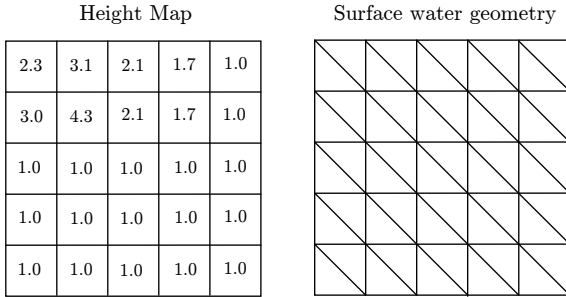


Figure 10: Inputs for the geometry of the water surface

For changing the geometry of the mesh we will use a vertex shader. For each vertex we know from the previous steps the height of the corresponding texel in the simulation and its normal, so the shader will change the coordinates of the input vertex adding to the height times the normal for this texel, plus its base 3D coordinates on its plane.

4.3.4. Creating the depth buffer from the surface water

After modifying the geometry of the water surface we are able to create the depth-buffer of the surface, needed for

Wyman's algorithm [Wym05]. This is done simply by storing the depth component when rendering the geometry generated for the water surface.

4.3.5. Refraction algorithm

To render refraction of a distant environment through two interfaces, the rendering algorithm requires two passes on a GPU. In our case the two interfaces will be the water and air. We ignore the glass of the windscreen, as its contribution to visual experience is minimal in comparison to the water surface.

Computing refractions through a single interface is relatively straightforward given basic information about the surface at the hit point P_1 (Figure 8). The behaviour of refracted rays follows Snell's Law, given by:

$$n_i \sin \theta_i = n_t \sin \theta_t \quad (9)$$

When ray tracing, refracting through complex objects is trivial, as refracted rays are independently intersected with the geometry, with subsequent applications of Snell's Law. Since GPU ray tracing techniques are relatively slow, multiple bounce refractions for complex polygonal objects are not interactive.

Instead, Wyman's algorithm proposes to approximate the necessary information to refract through two interfaces with values easily computable via rasterization [Wym05]. Consider the information known after rasterization. For each pixel, we can easily find the incident direction $\vec{V}$, the hit point P_1 , and the surface normal N_1 at P_1 .

Using this information, the transmitted direction $\vec{T}_1$ is computed using Snell's Law. To compute $\vec{T}_2$, only $\vec{T}_1$, the point P_2 , and the normal N_2 are necessary. Since finding $\vec{T}_1$ is straightforward, we have to know the P_2 and N_2 . Once again, we use an approximate point P_2 and normal N_2 since accurately determining them requires per-pixel ray tracing. After finding T_2 , we assume we can index into an infinite environment map to find the refracted color.

The main components of this method are the approximation of P_2 , and the computation of N_2 . Consider the parameterization of a ray $P_{origin} + t\vec{V}_{direction}$. We can write this as: $P_2 = P_1 + d\vec{T}_1$, where d is the distance $\|P_2 - P_1\|$. Knowing P_1 and $\vec{T}_1$, approximating location P_2 simply requires finding an approximate distance d , such that:

$$\tilde{P}_2 = P_1 + d\vec{T}_1 \approx P_1 + d\vec{T}_1 \quad (10)$$

The easiest approximation $\tilde{d}$ is the non-refracted distance $d_{\vec{V}}$ between front and back facing geometry. This can be easily computed by rendering the refractive geometry with the depth test reversed, storing the z-buffer in a texture, rendering, and computing the distance using the z values from the two z-buffers. We precompute $d_{\vec{N}}$ for every vertex, and interpolate using:

$$\tilde{d} = \frac{\theta_t}{\theta_i} d_{\vec{V}} + \left(1 - \frac{\theta_t}{\theta_i}\right) d_{\vec{N}} \quad (11)$$

After approximating $\tilde{P}_2$, the last requirement to compute $\tilde{T}_2$ is the normal $\tilde{N}_2$. Here, we rasterize the geometry in two passes: one culling front faces and one culling back faces. If we render the first pass to texture using the surface normal as color, during the second pass we can project our approximate outgoing point $\tilde{P}_2$ into texture space and index into the texture to find the normal. Now, we just use $\tilde{N}_2$ to find the reflected color using the environment map.

4.4. Gravity and wind

There are many ways to add external forces to the LBM. We modify the calculated velocity in each cell in the collision step. Thus, the local equilibrium is calculated with a modified velocity $\tilde{v}^*$ that is given by:

$$\tilde{v}^* = \tilde{v} + \tau \tilde{F}_g \quad (12)$$

We will insert this new operation between the computation of the velocity and density, and the collision step.

The force of wind is introduced into the model in the same way. We modify the velocities by adding an acceleration. For simplicity, we assume that wind has the same direction and force for all points on the windscreen. We allow the user to modify the direction and the force of the wind during the simulation using the keyboard.

The force of the wind is modeled by a 3D vector, but our simulation is in two dimensions. We therefore compute, for each texel, the tangential component of the vector $\tilde{F}_t$ over the windscreen. In each texel, we calculate the net force of the wind with $\tilde{F}_t = \tilde{F}_{wind} - \tilde{N}(\tilde{N} \cdot \tilde{F}_{wind})$, where $\tilde{F}_{wind}$ is a vector which represents the force of the windscreen, and $\tilde{N}$ is the surface normal. This tangential force is then added to the simulation:

$$\tilde{v}^{**} = \tilde{v}^* + \tilde{F}_t \quad (13)$$

4.5. Modelling rain

In each cell we know the number of particles of water, and therefore the density ρ . To add new water to the simulation, we only need to choose a set of cells and add more particles, which can be accomplished with a shader function. This shader receives as input the three textures used for LBM, and the coordinates of the cell where we will add new water. We distribute the new water among the 9 particle distribution functions f_i , according to the force of the wind and gravity. Thus, we modify the three textures of LBM adding in each f_i their respective new water.

4.6. Wipers

The windscreen wipers can be seen as providing boundaries but with movement instead of being fixed. This creates several challenges: the computation of the boundaries, and how



Figure 11: Operations for render, now with a new shader for compute the boundaries of the wipers

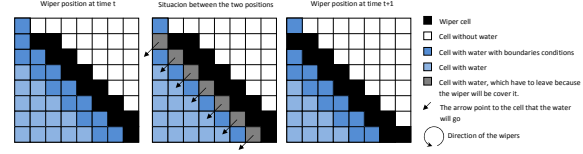


Figure 12: Wipers

we change the LBM to take them into account, as well as the final visualisation.

We define the wiper with four points (rectangle) and an angle ω representing its inclination. This angle is updated at each time step. We render the wipers into a texture which will tell us at the texel level where we have a wiper and where not.

Thus, some of the cells in the LBM simulation will be occupied by the wiper. For those cells, any remaining water is funneled into neighbouring cells according to the movement of the wiper. Once more, this can be achieved using a shader function (Figure 11). In this shader we modify the texture created before: We write the coordinates (0, 1 or -1), with respect to the center texel, of the closest neighbour that receives the water. To choose a neighbour, we use the one with smallest angle between the wiper's perpendicular vector and the e_i direction of the distribution functions f_i .

Now we have a new input for our shader, namely the texture with the location of the wipers. Three situations could occur for each cell. The cell is either occupied by the wiper, one of its neighbours is occupied by the wiper, or the in the default case, neither the cell or its neighbours is occupied by the wiper (Figure 12).

For cells occupied by the wiper, we set its new density in the next time step to zero. All of this water will move to one of its neighbours, as described above.

If a neighbour is occupied by the wiper, then we apply a *free slip boundary condition* on these texels. Afterwards, we read its 8 neighbours, and check for a wiper over them. Then, we pick up the water of a neighbour that still has water and in this time step will be covered by a wiper. In this case, we add the neighbour's water to our own cell.

5. Results

Our simulations are executed on a NVIDIA PCI-E GeForce 8600 GT. As shown in Table 1, the frame-rate decreases for increasingly large textures. This is a typical result of

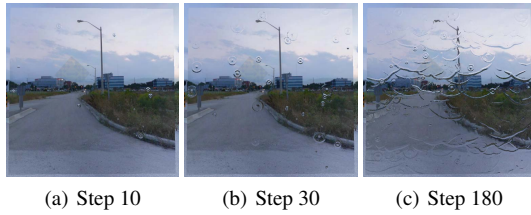


Figure 13: A simulation of rain, with gravity and the boundary conditions which permit to leave the water at the sides. Texture size: 512x512

Texture-size	16	32	128	512	1024
FPS	240	180	120	31	9.5

Table 1: The evolution of the frame-rate (FPS) depending on texture size.

GPGPU simulations. However, observe that for textures under 512x512 more than 30 frames per second are achieved.

Now we will show some results of the simulation in different situations and conditions. In these results, the windscreen is forming a right angle with the ground plane (90°) because, as in this position we can better appreciate the water effects.

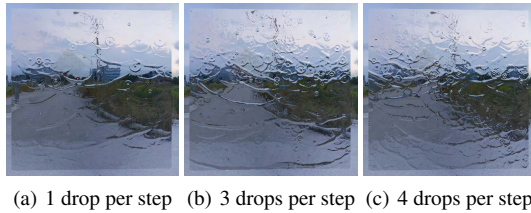


Figure 14: Differences between 1 (a), 3 (b) and 4 drops (c)

- A simulation of rain with gravity and boundary conditions, Figure 13. Simulation is performed in a 512x512 texture. As we can see in Figure 13, we can have a thick layer of water on the windscreen.
- Differences between rain intensities, Figure 14.
- A simulation with wind, Figure 15. We start from the situation when there is wind in the same direction as gravity (a) to one showing what would happen when we are driving fast and the net wind force comes from the lower part of the windshield (d).
- With the wipers turned on, Figure 16. This simulation shows the effect of the wipers and how they take off the water. The effect of the wipers is not as real as can be because of the lack of a surface tension implementation: when the wipers take off the water, and another drop which falls in a location where we supposedly do not have

water anymore, the drop is still doing the same effect ripple as if there would be water.

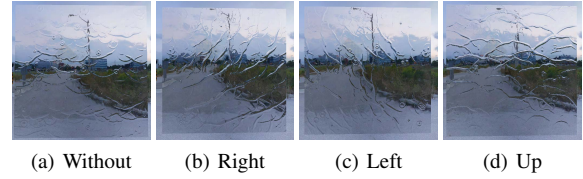


Figure 15: Wind simulation: without (a), from right (b), from left (c) and upwards (d)

6. Conclusions

In this paper we have presented a real-time, accurate water simulation on a car windscreen using the Lattice-Boltzmann Method. We have successfully implemented external forces like gravity and wind. Also, we implemented boundary conditions to simulate a realistic car windscreen. Moreover, we implemented an easy solution for the wipers to take off the water on the windscreen. About the results on the render, the results are comparable to a real windscreen, but further comparisons with real photographs should be made.

7. Future Work

The main objective of this project was to implement a realistic windscreen simulation. For this, we implemented a real-time version of the LBM method, adding forces like gravity and wind, and the wipers, plus a visualization stage. So the main future works in this project will be to focus in the simulation:

- Implementation surface tension. It is important for the situation when we do not have water and a drop falls on the windscreen. For instance, this happens when the wiper has just passed.
- Explore the usage of other shader units, like the Geometry Shaders, to decrease the number of passes on the GPU.
- Improve the implementation of the wipers. In our current implementation, all the water which is in a cell is moved to only one neighbor, the closest one. We think we should implement a movement of water in more than one direction at the same time.
- Improve the rendering of the water. Computing an accurate Fresnel term in a pre-processing stage would improve both rendering speed and visualization quality.
- Improvement of the FPS of the simulation. Different implementation aspects could be considered in order to speed it up.

8. Acknowledgements

This work was done with grant TIN2007-67120 from the Spanish Government.

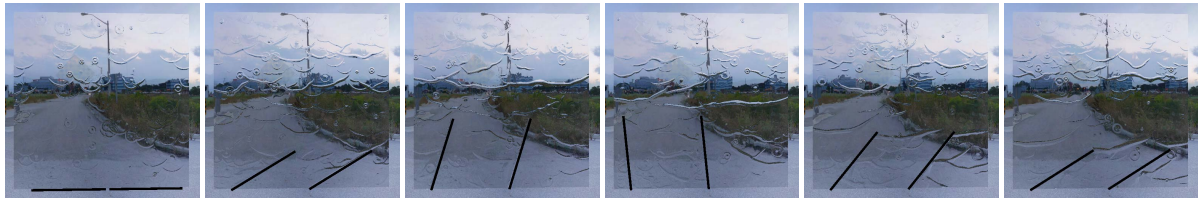


Figure 16: A simulation of the wipers. In a 512x512 texture.

References

- [Bri08] BRIDSON R.: *Fluid Simulation for Computer Graphics*. AK Peters, 2008.
- [BS01] BUKOWSKY R., SZALEWICZ K.: Monte carlo simulations for gaseous and liquid argon with complete ab initio nonadditive potential. *Journal of Chemical Physics* (2001).
- [CCM92] CHEN H., CHEN S., MATTHAEUS W. H.: Recovery of the navier-stokes equations using a lattice-gas boltzmann method. *Physics Review A* 45, 8 (1992), R5339–R5342.
- [CT05] CHU N. S.-H., TAI C.-L.: MoXi: real-time ink dispersion in absorbent paper. *ACM Transactions on Graphics* 24, 3 (2005), 504–511.
- [EHJP08] EL HAJJAR J.-F., JOLIVET V., GHAZANFARPOUR D., PUEYO X.: A model for real-time on-surface flows. *The Visual Computer* (2008).
- [GKN07] GARG K., KRISHNAN G., NAYAR S. K.: Material based splashing of water drops. In *Proceedings of Eurographics Symposium on Rendering* (June 2007).
- [GN06] GARG K., NAYAR S. K.: Photorealistic rendering of rain streaks. *ACM Trans. on Graphics* 25, 3 (2006).
- [HJ89] HIGUERA F., JIMENEZ J.: Boltzmann approach to lattice gas simulations. *Europhysics Letters* 9, 663 (1989).
- [HL97] HE X., LUO L.-S.: Lattice boltzmann model for the incompressible navier-stokes equation. *Journal of Statistical Physics* 88, 3-4 (1997), 927–944.
- [KKY93] KANEDA K., KAGAWA T., YAMASHITA H.: Animation of water droplets on a glass plate. In *Proceedings Computer Animation* (1993), pp. 177–189.
- [LFWK05] LI W., FAN Z., WEI X., KAUFMAN A.: Flow simulation with complex boundaries. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Pharr M., Fernando R., (Eds.). Addison-Wesley Professional, 2005.
- [Mar08] MARCÓ J.: *Accurate Water Simulation for Visibility in Driving*. Master's thesis, Universitat de Girona, Spain, September 2008.
- [MZ88] MCNAMARA G. R., ZANETTI G.: Use of the boltzmann equation to simulate lattice-gas automata. *Physics Review Letters* 61, 20 (1988), 2332–2335.
- [QdL92] QUIAN Y. H., D'HUMIÈRES D., LALLEMAND P.: Lattice bgk models for navier-stokes equation. *Europhysics Letters* 17, 6 (1992).
- [Sad99] SADUS R. J.: *Molecular Simulation of Fluids: Theory, Algorithms and Object-Oriented*. Elsevier Science Inc., New York, NY, USA, 1999.
- [Wag08] WAGNER A. J.: *A Practical Introduction to the Lattice Boltzmann Method*. Department of Physics North Dakota State University, 2008.
- [WMT05] WANG H., MUCHA P. J., TURK G.: Water drops on surfaces. *ACM Transactions on Graphics* 24, 3 (2005), 921–929.
- [Wym05] WYMAN C.: An approximate image-space approach for interactive refraction. *ACM Transactions on Graphics* 24, 3 (2005), 1050–1053.
- [Zim07] ZIMA C.: Converting displacement maps into normal maps in hlsl. <http://catalinzima.spaces.live.com/blog/cns!3d9ecae1f2dc56ef!223.entry>, September 2007.