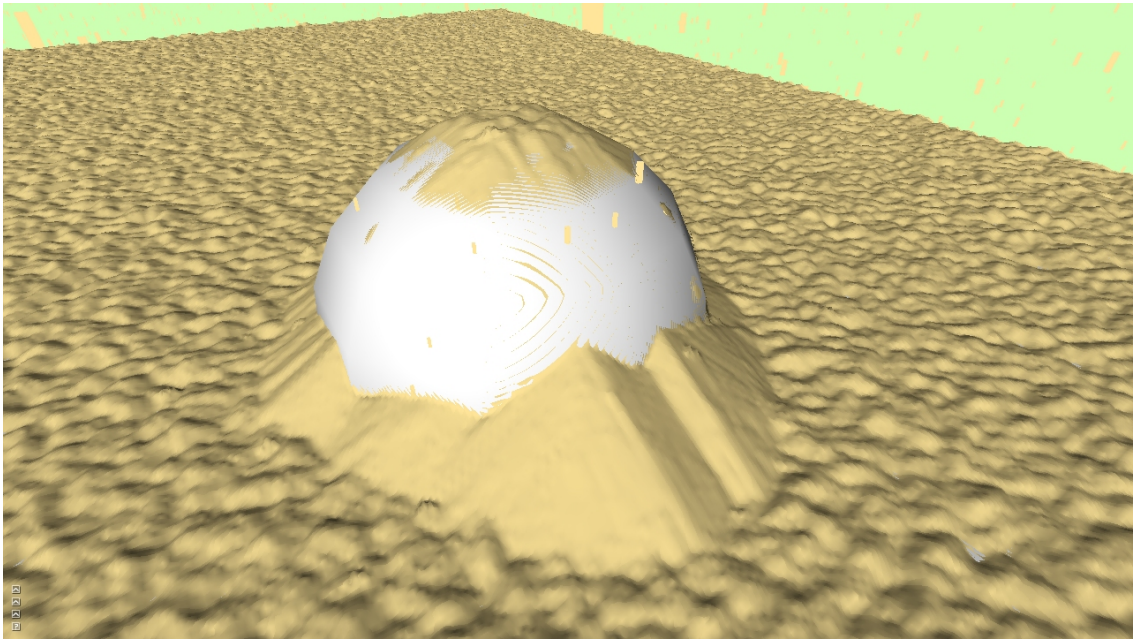
Universidade do Minho

Particle Deposition



Bruno Ferreira PG41065, José Fernandes A82467, Vasco Figueiredo PG41102

22 de Julho 2020

Índice

1. Introduction

Particle deposition is the spontaneous attachment of particles to surfaces. This phenomenon can be observed everywhere, from dust particles settling on a surface, to colloidal particles interacting with a substance, or even inside the human body, inside the respiratory system. The deposition may occur in a normal surface (planar, curved) or it can happen on other particles, which can lead to thicker particle deposits (for example, dunes).

The aim of this project was to develop a realistic, real-time particle deposition simulation, able to simulate the deposition of particles in any given environment. The goal of the simulation was to not only be able to simulate the deposition of particles like sand or snow, but to also try to simulate the deposition of fluids with different viscosities, like water or honey (in any given scene). While a simulation of liquids is more complex than a simulation of particle deposition, the simulation developed was designed to use the concepts and ideas behind particle deposition (with some additions) to accurately simulate liquids.

The report begins with the introduction to the different possible implementations, what implementation was chosen and why, the various steps (and step backs) during the development, and the results achieved by the simulation.

2. Implementation

2.1. Physics vs Height Map

The two main implementations for a particle deposition simulation are: implementing particles with physics, or using height maps. With the first implementation, particles would need to be susceptible to the laws of physics (be affected by forces) and have colliders. This means that the simulation would have to calculate all the collisions between them in real time. Each scene (and the objects present) would also need colliders. Since the particles would follow real life physics, this implementation would create a pretty realistic simulation, at the cost of performance. The fact that every object in the scene (and the scene itself) also required a collider would complicate even further the process of using a new scene (although this problem could be worked around with clever tricks).

In the second implementation, the particles don't actually "exist". The interactions and movement of the particles are calculated on a height map. Each coordinate holds the number of particles in that position. Then, with given parameters, the way the transmission from one position to another occurs can be changed. For example, if the transmission is defined to never happen, particles that fall on the same position, stay in that position. So if the opposite is true (transmission always happens), every time a particles falls on a position and that position has more particles than the neighboring positions, the particle would be sent to another position (presumably the neighbor with less particles on it). So in this implementation, we simulate the physics with transmission parameters on a height map. In terms of the scene, the only thing we need is the height of the scene (and every object present). The main drawback of this implementation is that creating a realistic simulation is not as easy as the physics implementation, and requires a lot more tuning of the transmission rules.

For this project, the second implementation (height maps) was chosen, for a couple of reasons. Firstly, this implementation is less complicated, since it doesn't require a physics system, and it performs better with a large number of particles, since it doesn't have as many calculations per particle. Secondly since one of our goals was to be able to use any given scene, the way the physics implementation operates would complicate this goal a lot more than by using height maps.

2.2. The Implementation

Simulating a large number of objects, or analyzing every pixel of a large image are tasks not suitable for a normal CPU, since a CPU would not be able to this efficiently. A GPU on the other hand, is much more

capable of processing large numbers of particles and/or processing large images. So, without a doubt, this project was designed to take advantage of the GPU with the use of shaders. To utilize shaders, the project used the engine Nau3D, which mainly uses GLSL (OpenGL Shading Language) for the shaders, and XML setup everything else (for example, the scenes and objects, cameras, lights, textures, etc).

The simulation has 2 main systems working at the same time: a particle system, to simulate the particles falling (creating a more realistic scene) and a system that processes the height map (that actually simulates the particle deposition). While these are the "main" systems, there are a lot of other shaders working behind the scenes to help producing the final result (for example, shaders that analyze the scene and generate the scene height map, or a shader responsible for drawing the particles deposited). The next part is separated into 3 parts: the particle system, the height map system and the rendering. In each part, the intricacies of each system will the explained in-depth.

2.2.1. The Particle System

The particle system is composed of two main elements: a compute shader, that processes each particle; and a geometry shader that draws the particles. The particles (as an object) are created by Nau3D as a buffer of positions. Nau3D is also used to create a separate buffers for the directions.

2.2.2. The Compute Shader

The compute shader is very simple and straightforward. Each thread first checks if the particle is not active (the w component of the position is 0).

If it's not, it randomly gives a position do that particle. The constraints for the random position can either be the whole scene, or in a modifiable range form a spawning position (that can also be changed in real time). The height (or range of heights) that the particle is created at can also be changed in real time. Aside from generating the position, it also generates a random direction, which will then be saved in the direction buffer. This way, each particle can have a different direction each time.

If the particle is already active, the compute shader only sums the forces (gravity, wind) to the particle, and updates the position accordingly. If the particle moves out-of-bounds, it's teleported to the other side other the scene (as to prevent piles of particles on the edges of the map). When it detects that the particle fell on the ground (by calculating the height of the scene height map on the position plus the height map the particles, also on that position), the computer shader deactivates the particle, and increases the particle height map by a modifiable amount.

2.2.3. Geometry Shader

Although the compute shader controls the positions of the particles, they don't have any form, since they're not actually an object. So a geometry shader is used to generate the "body" of the particle. To simplify, each particle takes a form of a cube. By generating the position of each vertex of the six faces, and then calculating the normals, a simple cube, that is also reacts to the scene light is created. While a cube maybe isn't the best form for every type of particle (for example snow), its enough for the goals of the project.

To further improve the particles, they can also be rotated towards their direction. By creating a rotation quaternion (number system that extends the complex numbers and can define rotations and orientations, and, unlike euler angles, avoid the gimbal lock problem), we can easily rotate the vector. This creates a more realistic look to the particles.

Depending on the selected particle spawning option (either the whole scene or in a certain range), the geometry shader also creates a form (either a square or a circle) that visually shows the range of the spawner. Like the compute shader, a number of the aspects of a particle can be changed, like height and width, or even the color of each particle.

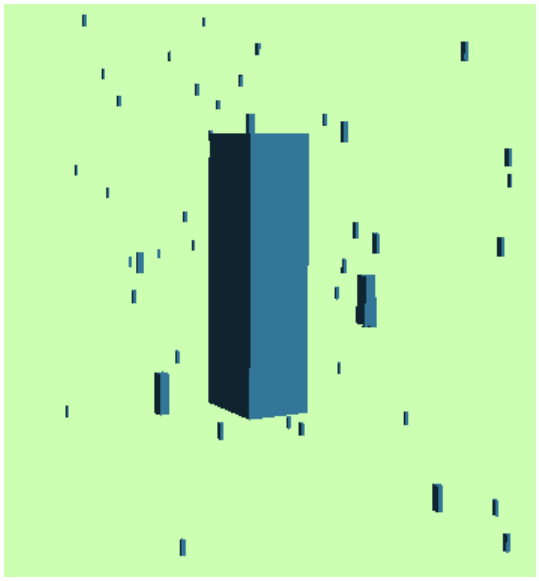


Figure 1: Simple cube generated by the geometry shader

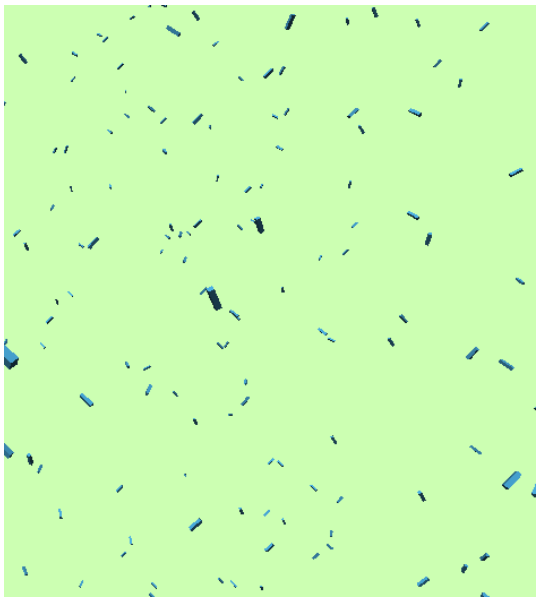


Figure 2: Particles pointing towards their direction

As stated before, during the execution of the simulation, the user can change a number of variables that affect the particle system.

Resuming the options:

- Particles Active: activates or deactivates the particles;
- Whole Map: defined the spawning option of the particles (either the whole scene or just in a circle;
- Spawn Position: center position of the spawner;
- Spawn Radius: radius of the spawner (only affects if the option wholeMap is deactivated).

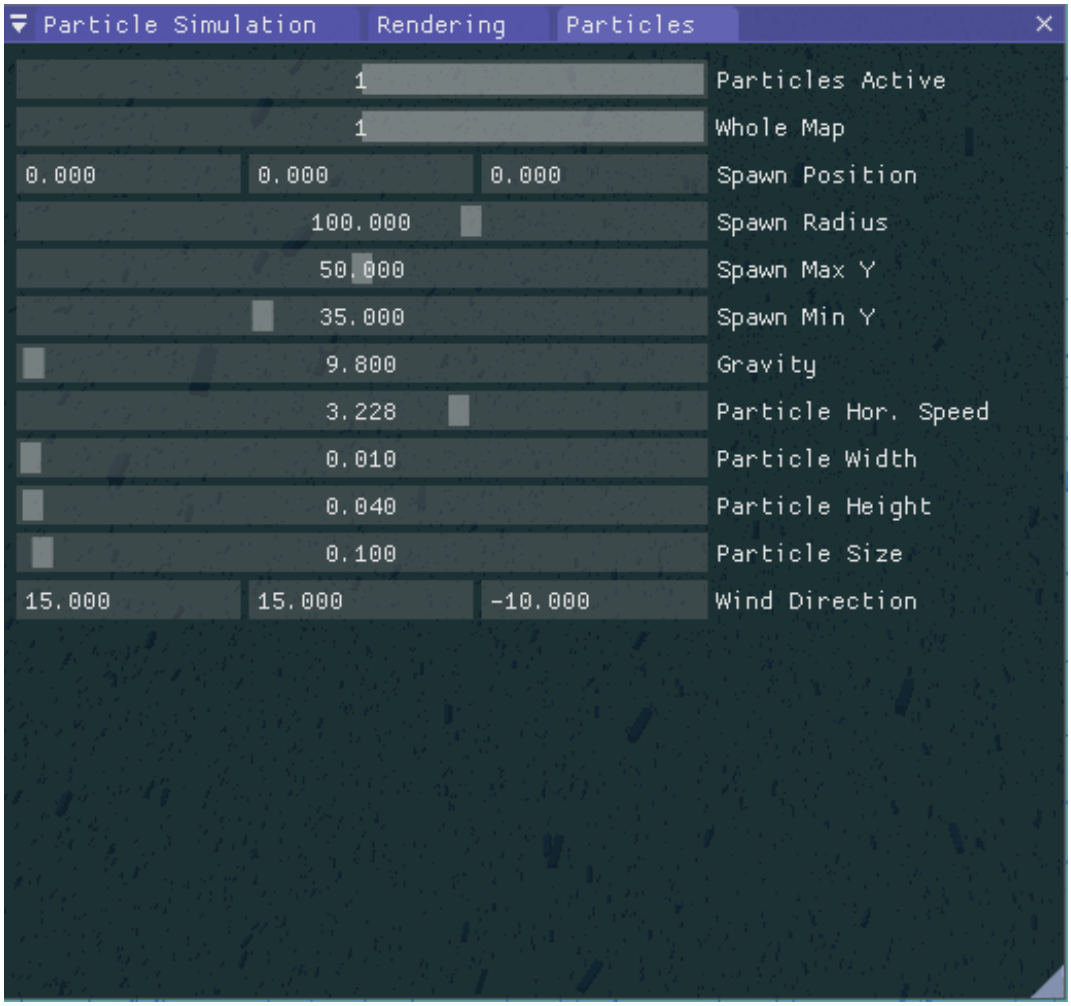


Figure 3: Particle system variables

- Spawn Max Y: maximum height position that a particle can be created at.
- Spawn Min Y: minimum height position that a particle can be created at.
- Gravity: Downwards force applied to the particles.
- Particle Hor. Speed: the horizontal speed of each particle (doesn't change the horizontal direction).
- Particle Width: the width of each particle.
- Particle Height: the height of each particle.
- Particle Size: the amount incremented to the particle height map when the particle collides with the terrain (bigger amounts will create bigger piles).
- Wind direction: Horizontal force applied to the particles (changes the direction of the particles).

In a simulation, the particle system looks like this.

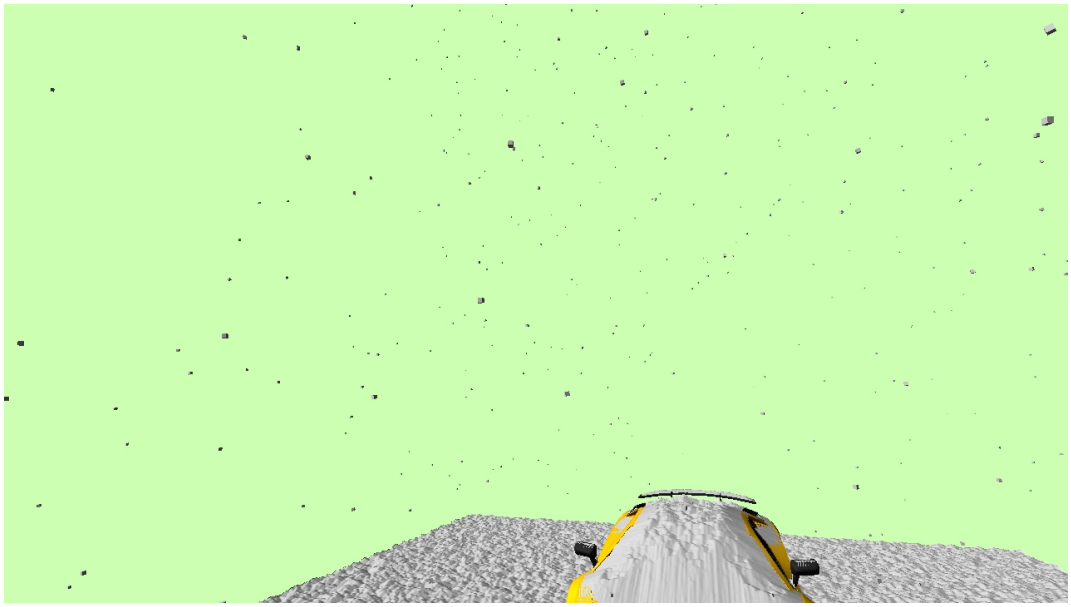


Figure 4: The particle system

3. Particle Deposition/Dissipation

3.1. Implementation

The algorithm was implemented using a compute shader. This algorithm was inspired by other dissipation algorithms such as the *heat* equation and was adapted in order to work with height-maps and ceiling-maps, height-maps that denote the maximum value a height can reach in a certain position.

Each thread of the compute shader is attributed one position of the particle map. It then checks, for each neighboring element whether it is able to receive or if it needs to transfer particles to it. This is done by subtracting or adding the difference between it's height and the neighbors.

A particle is deemed illegible for transfer if it in a higher position than it's neighbors as we can see in Figure 5. In Figure 6 we can see an example of how the dissipation occurs.

The world interaction was done by adding the height of the world, determined by a height map obtained from an orthographic camera overlooking the world, and by then clamping the value in order to preserve the number of particles in the scene.

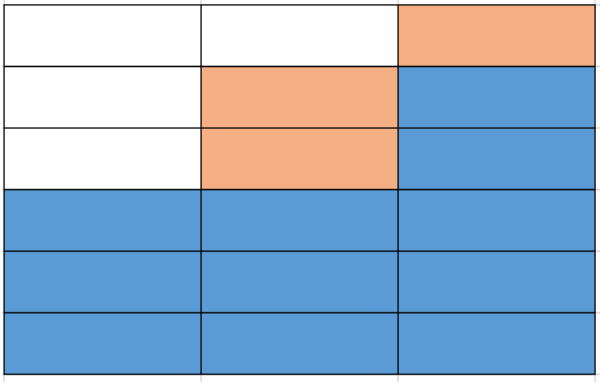


Figure 5: Transferable particles (Orange), Acumulated particles (Blue)

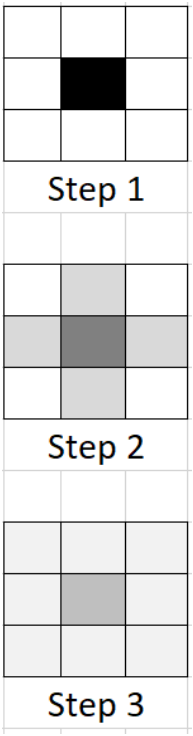


Figure 6: Dissipation Algorithm

3.2. Limitations and Solutions

The current implementation, which uses height-maps for the world height, cannot accumulate particles under models as we can see in figure 7.

The solution for this would be to slice the scene in multiple regions, each with their own height-maps and ceiling-maps as we can see in figures 7 and 8. It would involve, in each iteration to run multiple simulations and then transferring particles between each layer in a separate step.

Unfortunately were were not capable of finishing the implementation of this solution due to problems with the code and difficulties with the configuration of the various cameras and textures.

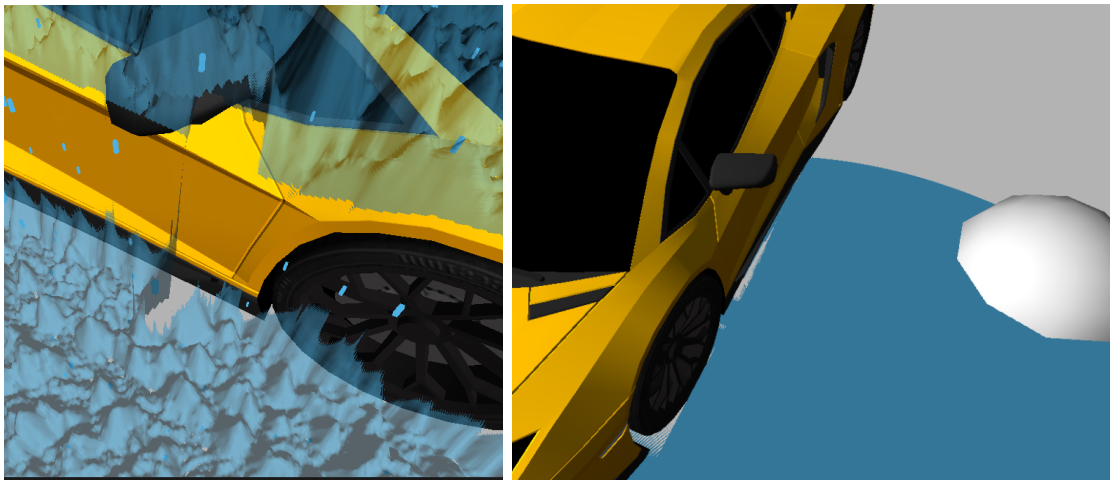


Figure 7: Left Without using Slices, Right with Slices

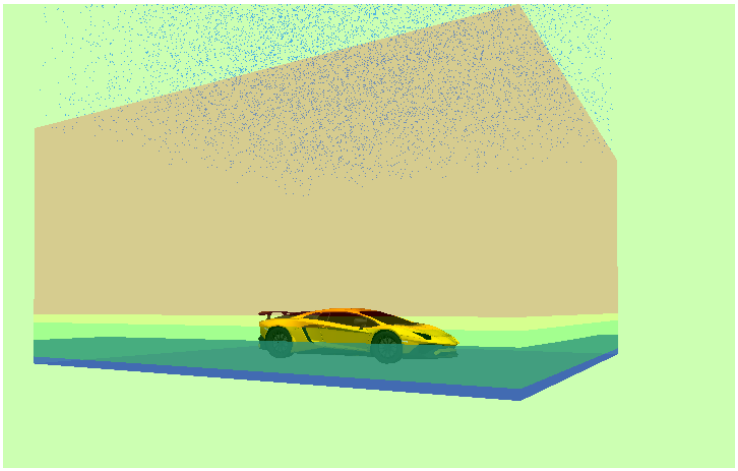


Figure 8: Example of scene divided into 4 slices

We did create a script that positioned the cameras in the scene and created a compute shader which, given a ceiling-map and a height-map determined the place where the camera should be split-ed. The split was calculated by determining the points in which the ceiling map had a lower value than the height-map and then selecting the place where this difference was the highest.

3.3. Simulation Parameters

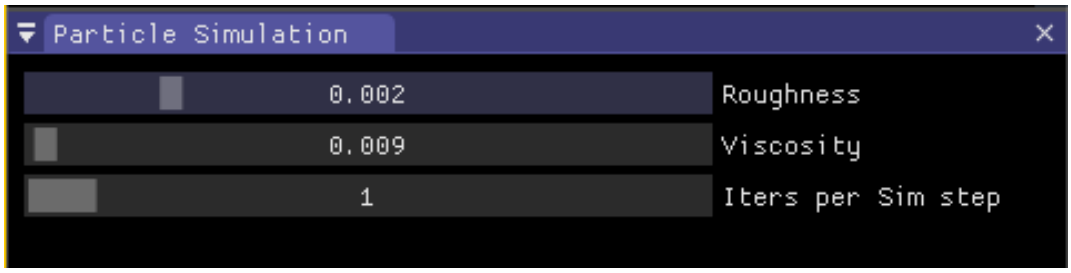


Figure 9: Simulation Parameters Menu

In the figure above we can see the parameters that can be changed in order to affect the way the dissipation of the particles occurs.

These parameters are:

- Roughness
  - The value of the roughness grants roughness to the deposition height-map
  - If the amount to be transferred is lower than the roughness the transfer doesn't occurs which creates bumps and diminishes the smoothness of the model.
- Viscosity
  - The viscosity parameter is a percentage value which helps control the amount of particles transferred in each iteration.
  - If the value is 0 the particles will be transferred at the maximum possible speed.
  - If the viscosity is exactly 1 no transference will occur and the simulation will be static.
- Iters per Sim step
  - This parameter is used to control the number of iterations that occur in every frame of the simulation.
  - It can be used to increase the speed of the simulation. However higher values tend to become more demanding, thus reducing the frame-rate.
  - This parameter becomes more useful when the simulation isn't being run in every frame.

3.4. Physics behaviour

As mentioned above, we have mainly two parameters for the behaviour of the deposited particles. Roughness and viscosity. Viscosity intends to simulate how slowly a particle moves around. Roughness, on the other hand, is a name we chose due to the appearance it creates and not exactly what it controls due to the lack of a better name. This changes how likely it is for a particle to start moving once it has been deposited.

Although initially we planned to only use a viscosity parameter, we feel as though it is necessary to have both to distinguish between the two states of matter: solids and liquids. It makes sense for liquids to always eventually converge into a steady state of height no matter how viscous they are and if roughness is set to zero, this is what happens. However for solid particles, viscous or not in the sense that they "glue" to each other, due to gravity and built pressure from stacking particles, they will collapse under their own pressure eventually and form sorts of mounds.

With these two parameters we are able to replicate the following behaviours.

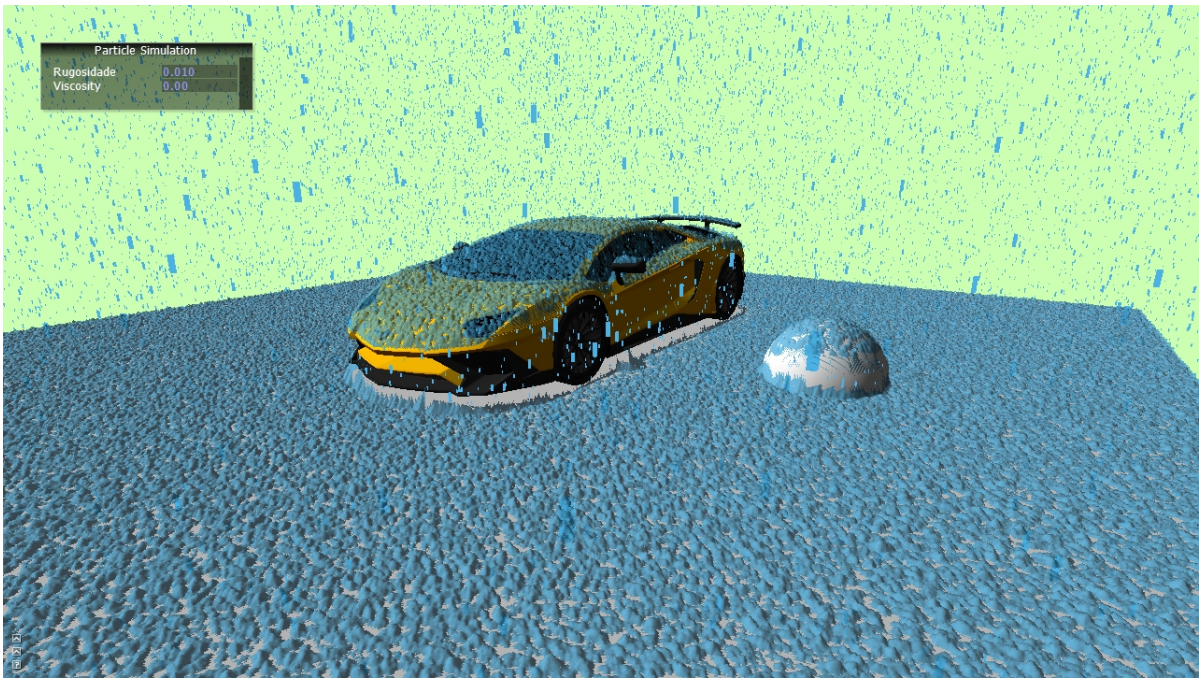


Figure 10: Low viscosity but high roughness (Solid)

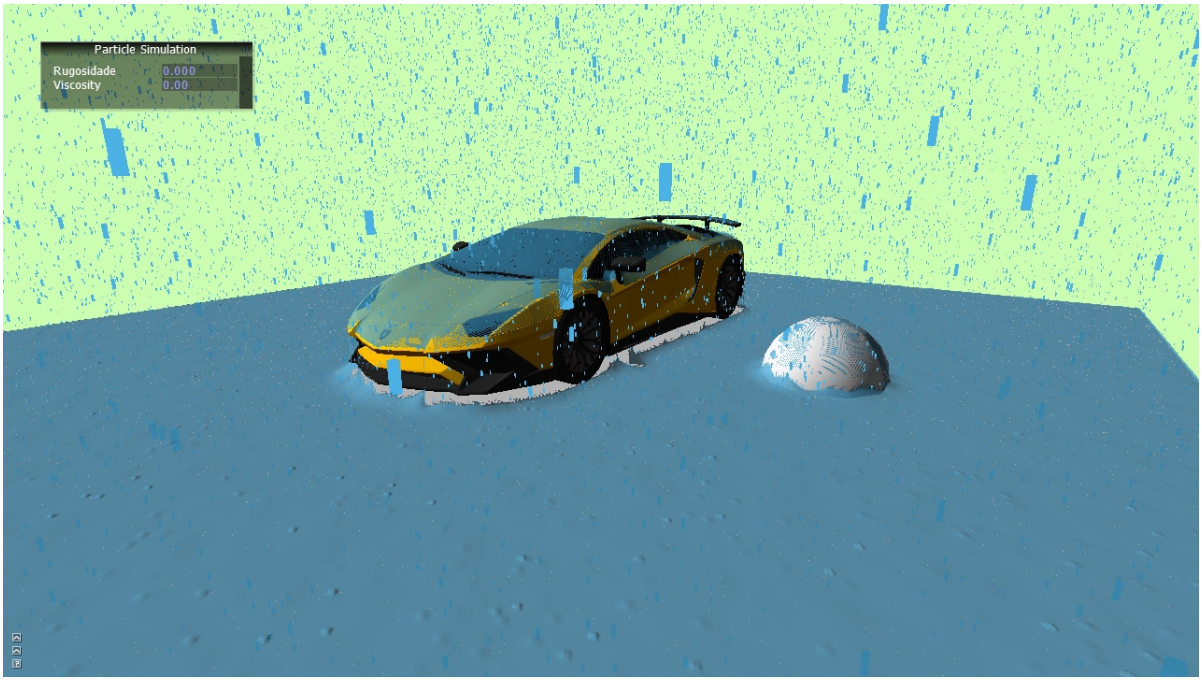


Figure 11: Low viscosity and low roughness (Liquid)

In these two images, despite having the same low viscosity, we can tell that each behaviour represents a different state of matter, one being a solid volume of particles and the other one being a liquid volume.

With different sets of values with higher viscosity we also got the following set of behaviours:

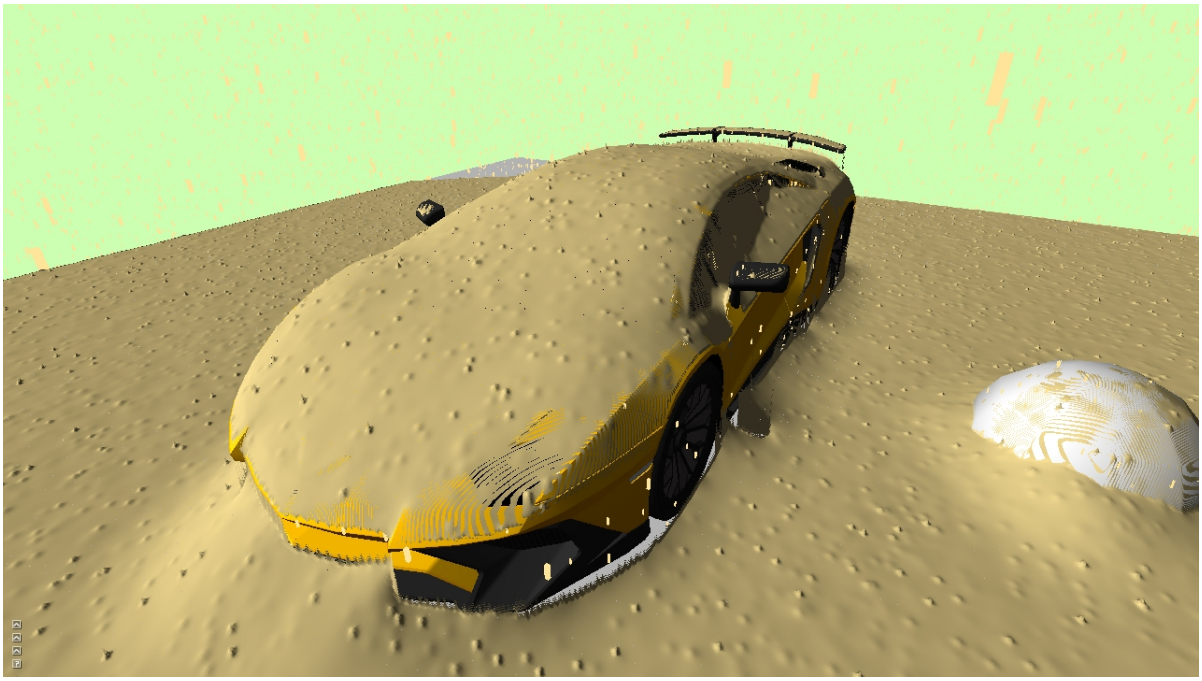


Figure 12: Thick liquid like behaviour

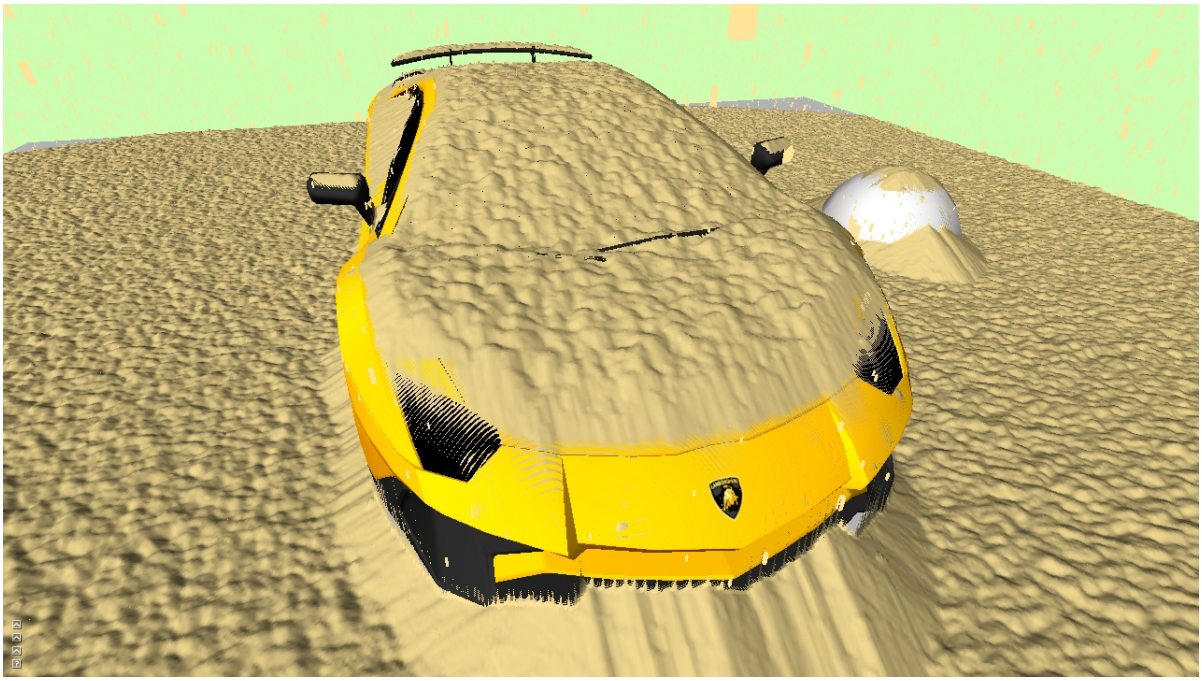


Figure 13: Sand like behaviour

Although the behaviour in figure 12 is still a liquid, due to its high viscosity it takes a lot longer for it to reach the steady state height a liquid would reach, the high points take very long to lower and even out with everything else.

On the other hand, we can see a different behaviour in figure 14 that represents what we described as sand and creates interesting realistic volumes like underneath the front of the car but also on top of the hood where the nose tips down and every particle slid down of it. A difference can also be noted on the sphere to the side.

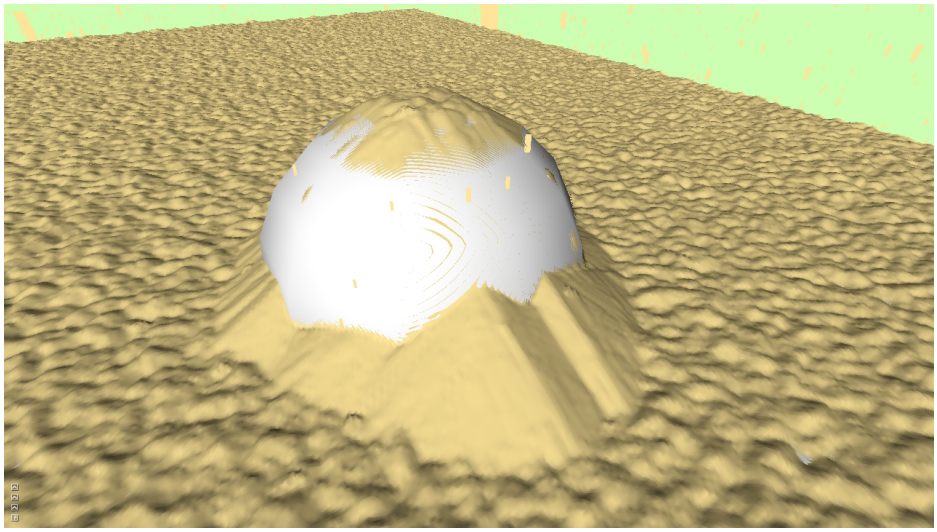


Figure 14: Sand accumulated on top of a sphere

4. Rendering

Having all the physics of the volumes of particles being simulated on textures isn't enough as we also want to see them on screen. For this we considered two ways of doing it: either having a mesh that we manipulate its vertices to match the height of the particles represented in the textures or have a system of rays that we go along of and check when they have intersected the height of these particles.

We opted for the second one as manipulating a mesh sounds simple enough for continuous scenes, in a scene with objects it would get complicated to break the mesh and separate it along discontinuous heights, whereas the second method we would not need to worry about this problem.

The idea is to use a type of ray tracing called ray marching. This is usually done by sampling along the ray and collecting data about the environment but here we simply want to check if we have intersected the volume of our particles and then stop processing if we have.

This method, however, can be very costly performance wise but we did a few tricks to minimize this.

4.1. Performance optimizations

One problem we could have is that, since a stoppage condition for marching the rays is necessary, for example the amount of samples, if the camera is too far away these rays will never reach our scene.

A way around this was to use a technique for intersecting rays with axis aligned bounding boxes (or ray-AABB intersection for short) which allows us to know exactly where the scene is if it is described by a bounding box, if a ray will ever intersect it or even where exactly it will intersect. Since we already have bounds that describe the scene due to the cameras that generate height maps, we can use these as a bounding box without limiting our program any more than it is.

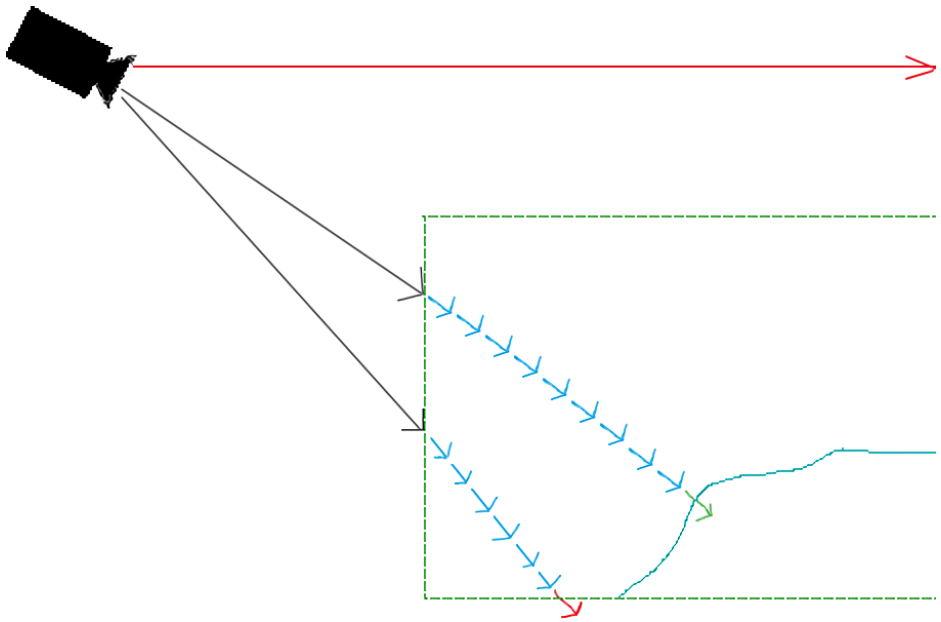


Figure 15: Sketch of bounding box optimization

In this sketch that describes our rendering algorithm we can see three rays. One of them, which is red, is immediately determined by the ray-AABB intersection as a ray that will never hit our scene as it misses the green dotted line that represents its bounding box. This means that the correspondent pixel can stop and simply output the original render color saving us a lot of processing.

The most lower ray does hit the bounding box and our ray-AABB intersection algorithm tells us

exactly the position where it intersected. With this known position, we can start marching along the direction and sampling if we are lower than the volume texture's values of the same position. This specific ray, however, never intersected any volume of particles. To save some more computation, we always check for each sample if it has exited the bounds and terminate the ray.

The ray in middle is however successful in finding a volume of particles. This ray then reads the value at its position from the texture and also the values around because we don't just need to know if it intersected something but also show it on screen. This implies correct shading which needs normals so we use these values to calculate the current pixel's shading.

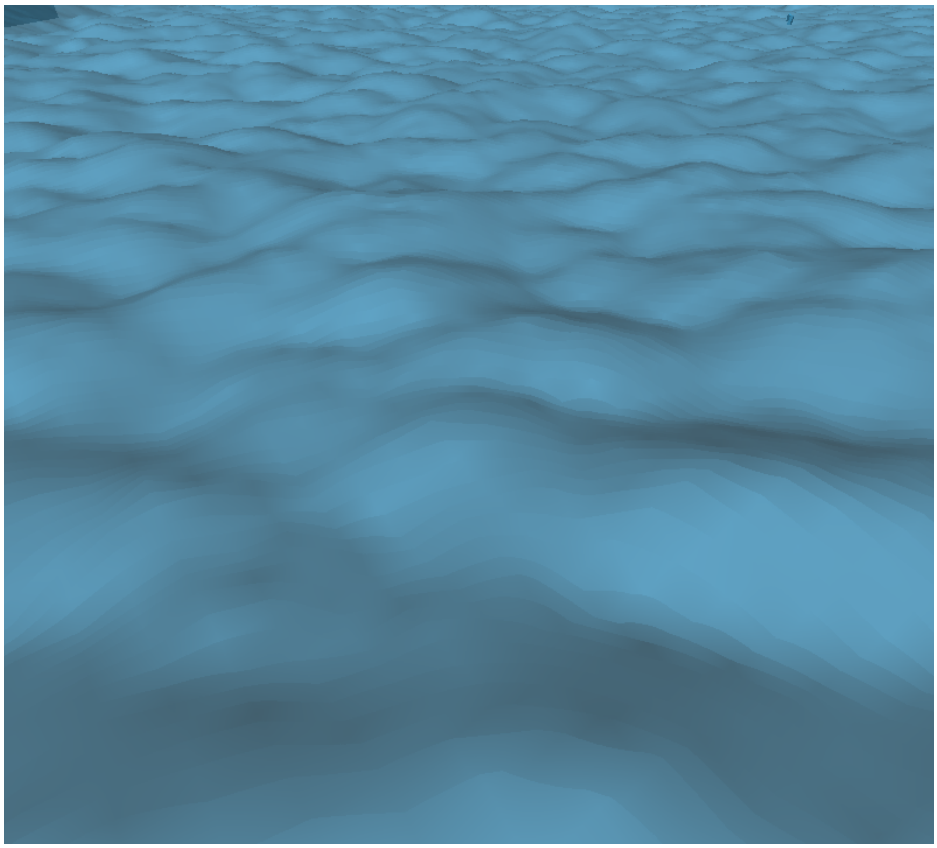


Figure 16: Render of shaded volume of particles with smooth different heights

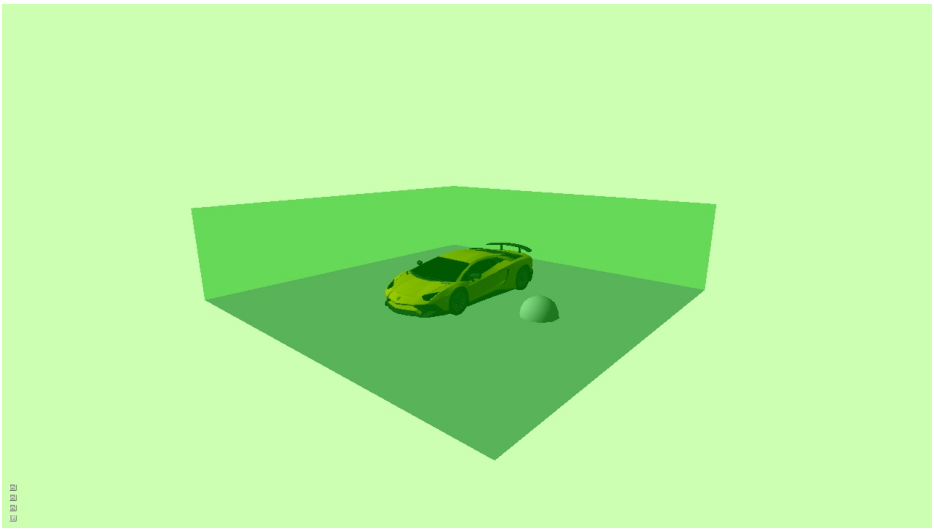


Figure 17: Render of bounding box with ray-AABB intersection

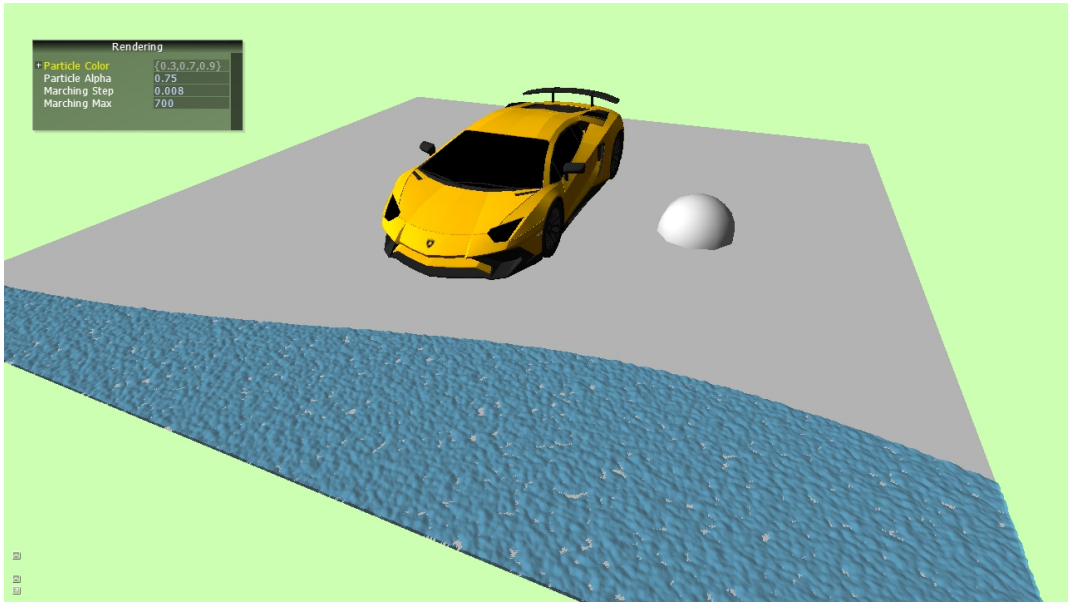
4.1.1. Parameters

Since we want our program to be adaptable to any scene, we have two important controllable parameters along others:

- **Marching Max** - how many samples the ray will take until it gives up from finding something;
- **Marching Step** - how much distance is traveled along each sample.

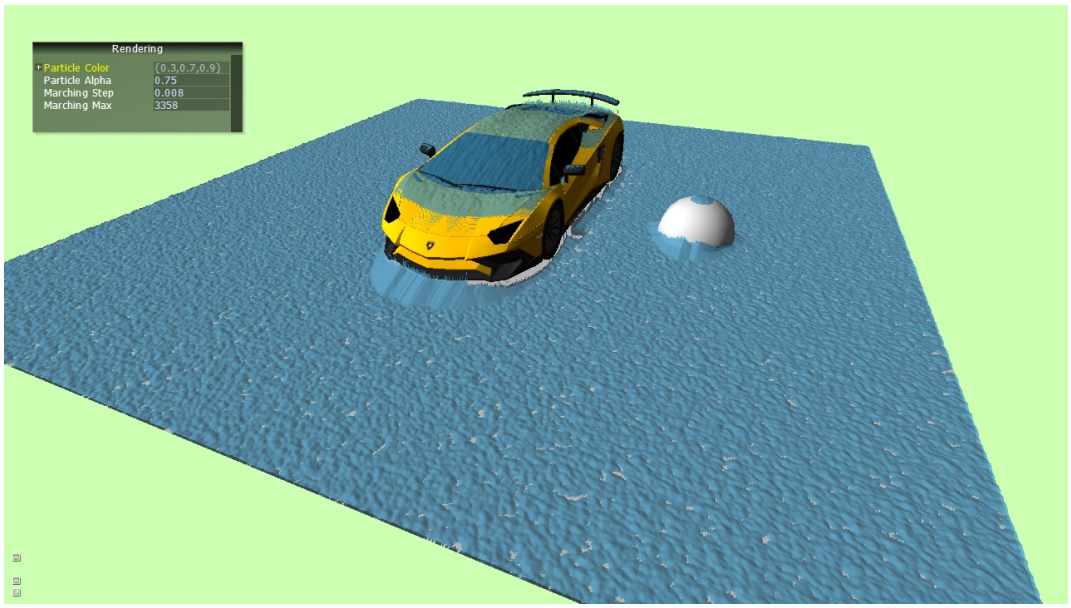
These are controllable to be fined tuned by the user as they impact performance heavily and have a big impact on the look of the rendering. As one can imagine, the **max** parameter is essentially our render distance and the **step** parameter is our level of detail. In the next four figures, we will see examples of what some combinations of values could be and their uses.

4.1.2. Low marching max and low marching step



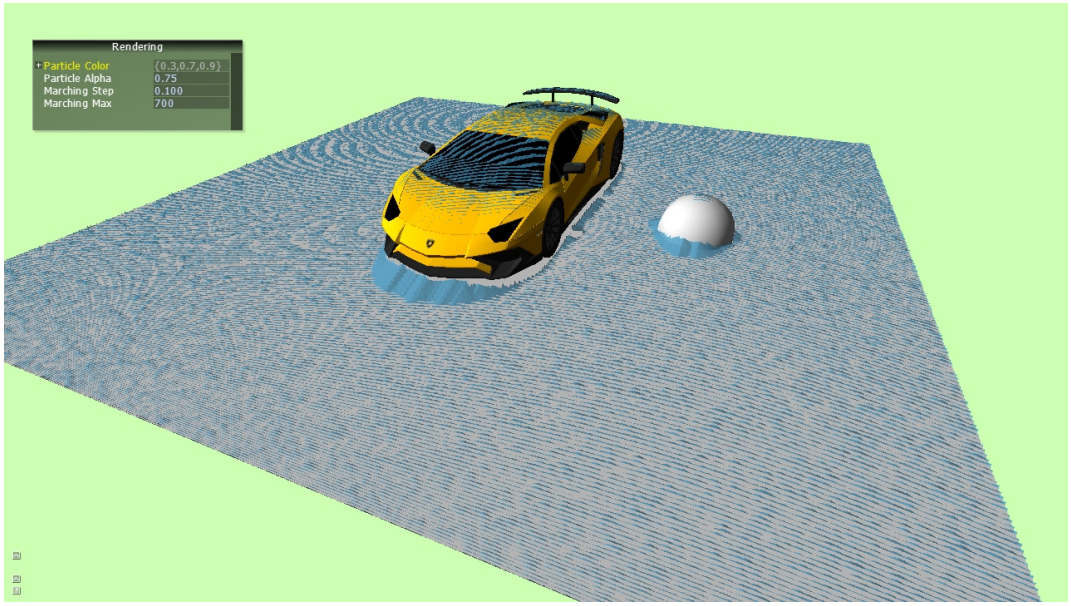
Here we can see that the **marching step** for our current camera position seems to be enough as it gives us good fine detail but rays that have to travel more distance to reach the floor where the particles sit are too shortly stopped due to the low **marching max** and do not render anything at all. This does not mean you should never have values similar to this as this would make sense if the camera was closer looking at very up close details without having as much performance cost.

4.1.3. High marching max and low marching step



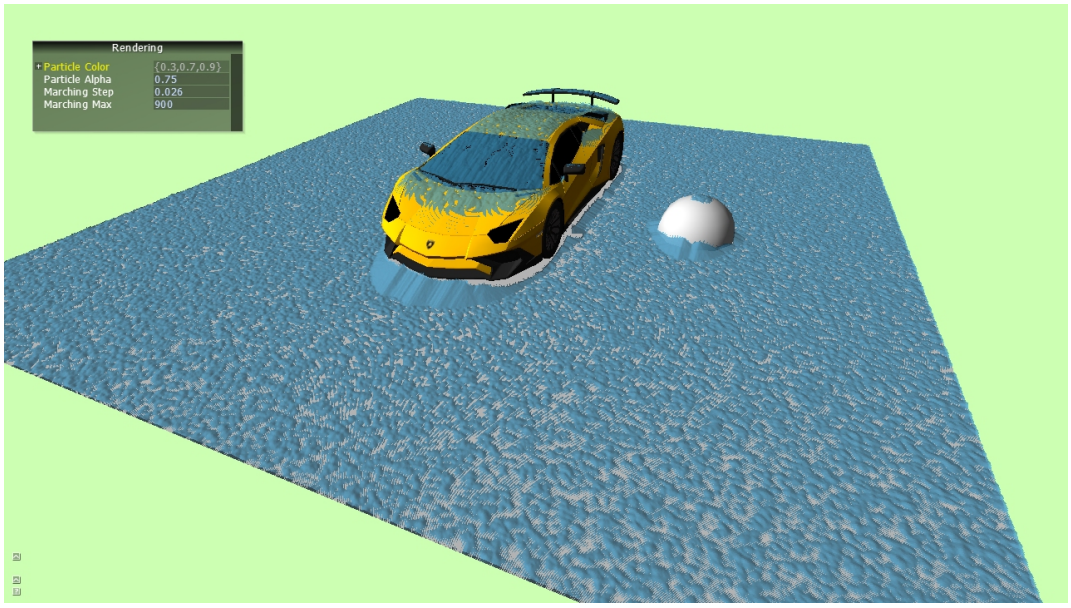
This one is the ideal choice as we lack less and less visual quality as we lower the **marching step** and we are able to see the whole scene in a single shot due to our big **marching max**, however, we could not run this at more than 60 frames per second, which is obviously dependent on our machine so this option could be viable for other hardware.

4.1.4. High marching max and high marching step



Here we can see exactly how much the **marching step** affects visual quality. What is happening here is that at certain spots of the image the step is longer than the width of the particles' volume meaning it skipped it entirely and think it didn't intersect anything in the scene, therefore not showing any color. This could, however, be enough if we were very far away from the scene.

4.1.5. Medium marching max and medium marching step



This one is a mix between visual quality and computation costs as we have enough visual information of what's happening and we are able to see it at over 60 fps.

4.1.6. Automation

The manipulation of these values can be a bit tiresome every time you a open a new project or even one with a different scene. For this we created a script that reads information about the scene and chooses an adequate **marching step** and **marching max**. This script also takes care of setting up all the sizes and orthographic cameras so that the textures created by it include the loaded scene such as the vertical depth map. It just needs to be opened and change the string value correspondent to the scene that encloses most of the volume we want to render due to a limitation of the engine used.

We also thought of changing these values in real time as the user travels through the scene, because these parameters can be changed with the depth information we have the camera's position. If the user was really close to an object we could increase the detail but lower the rendering distance and vice versa for when the user is further away. We did not implement this but it's a good future feature to implement.

4.2. Depth Testing

One of the problems this idea is the fact that the geometry created doesn't take into account the objects in the scene automatically. So the shader needed a way of knowing where the objects are. This could be achieved by using a depth buffer. With a depth buffer, a render image would be created with the distances from the objects to the camera. The problem is, OpenGL doesn't give us an image with the actual distances to the camera. And we can't find those distance in this shader since it is working on a render image. To overcome this issue, a custom depth buffer was created.

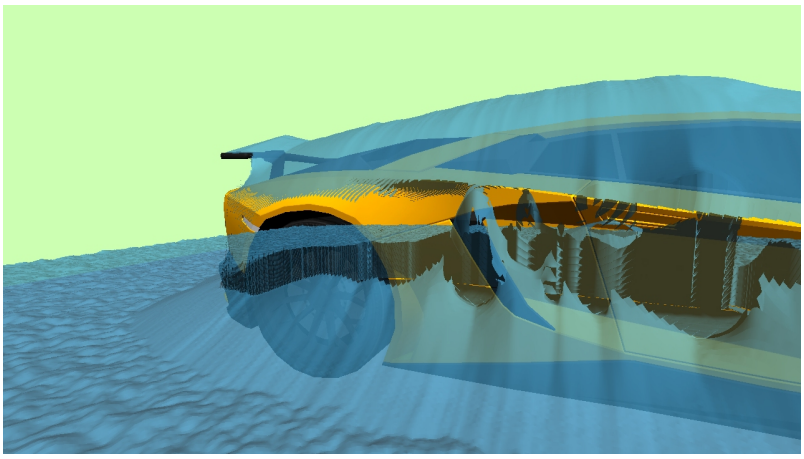


Figure 18: Rendering without the depth buffer

This custom depth buffer is created by a vertex and a fragment shader. The main idea here is using the fragment shader to calculate the distance of the camera to the pixel, and then outputting the real distance between them. A simple and effective way we can do this, is to use the position of the vertex (sent by the vertex shader) and the camera position, both eye space, and calculate the distance between them. My moving both to eye space, we can accurately calculate the distance between them. The result from this shader is then rendered to a texture

The rendering shader than uses the uv coordinates to the max distance in the current pixel. Each step of the marching accumulates the traveled distance. If the traveled distance exceeds the max distance, the function stops and returns zero. This way, particles behind objects won't be drawn.

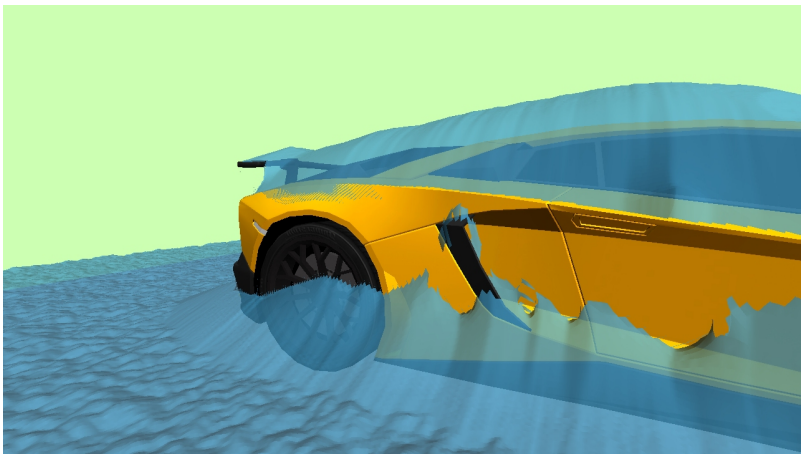


Figure 19: Rendering with the depth buffer

5. Conclusions and Future Work

The goal of the project presented in this report was to realistically simulate particle and fluid deposition. To achieve this, a height map based implementation was used. With these height maps, the project can simulate the physics of this phenomenon without actually having to calculate any physics. With the use of shaders, we can use the GPU to improve the performance of the calculations, and since processing textures in one of the biggest strengths of shaders/GPU, the simulation can simulate a larger number of particles, for example, 1 million particles, without much trouble.

The way we achieve a realistic result is by adding parameters that try to simulate the laws of physics without actually trying to calculate them. For example, changing the speed at which the height map transfers the particles to its neighbors or even the amount transferred can greatly change the simulation in order to produce a result similar to real life.

Aside from the particle deposition system, the simulation also has a particle system that creates the falling particles, and a render shader, that utilizes ray marching to create the deposited particles volume.

This simulation produces very satisfying results, accurately simulating snow, sand and even liquids (to some degree). However, the implementation has some problems.

The biggest problem is the height mapping. Using one height map might be enough for some scenes, but if the scene has roofs, a problem appears. The simulation needs to know that, while there's in fact an object at a certain height, below that object, there's empty space. Due to the complicated nature of this problem, trying to resolve it by implementing a generalized algorithm proved to be very tough. Although many options were tried (using compute shaders or even lua scripts) no option was able to produce great results.

This simulation, while great at simulating particle deposition, is not the best at simulating liquids. It does produce a very good result, but some properties present in an actual simulation of liquids are not present here, for example particle pressure or velocity. The convergence is also not as fast as it would be with, for example, water. Essentially, the simulation is great at simulating dissipation and deposition, but not as good at simulating other phenomenons.

The ray marching used also has some set backs, mainly in the cost vs visual quality. Ray marching can prove to be very expensive performance wise if it tries to give best possible result.

For future work, resolving the height mapping problem is the biggest target, either by finding a different solution to this problem or trying correct the current implementation. The height maps could be aggregated with the particle height maps in order to reduce the number of textures needed thus leading to better performance. To further improve the ray marching, changing certain values in real time as the user travels through the scene, for example, increasing the detail but lowering the render distance if the user is really close to an object and vice versa for when the user is further away.

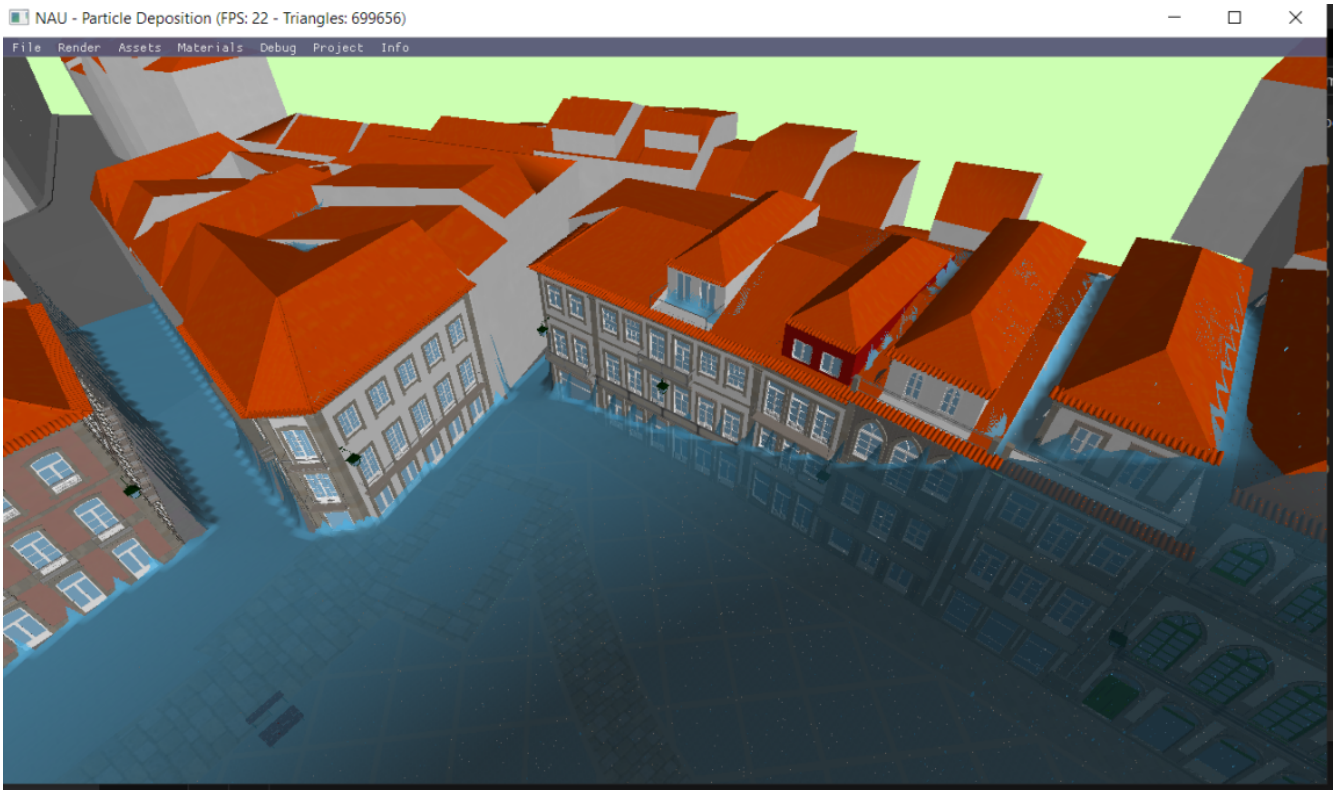


Figure 20: The Great Flood