Fluid Simulation Report

Chu-I Chao, Neng Qian

March 24, 2019

Abstract

The report presents the techniques for fluid simulation, Besides the classic approaches which were taught in the lecture, we will also introduce our own creations, mainly on boundary generation and implementation of marching cube. Our work on boundary generation helps us create more interesting shapes of boundary and moving boundary as well. Also in the section about our marching cube implementation, we will prove our method is more efficient and robust.

Contents

1 INTRODUCTION

Fluid simulation techniques have been widely used in animation and games nowadays. In this lab, we've already gone through some classic fluid simulation approaches, including Weakly Compressible SPH (WCSPH) and Position Based Fluid (PBF). Built upon the methods we've learned, our own creations are made and will be further described. Our first work is involved in the boundary generation, which allows us to create spherical boundary and also moving boundary. Another creation is the dynamic bounding box of marching cube algorithm. It will be proved to be more efficient and robust than the fixed bounding box.

The report is organized as follows. The section 2 presents our software organization in both low- and high-level views. In the section 3, we will introduce the modules we used in our implementation and the corresponding unit tests. The section also includes the comparison between WCSPH and PBF, regarding the quality of simulation results and time performance. In the section 4, we will look into one of our own creations, the boundary generation. The section 5 introduces another creations of ours, the improved implementation of marching cube algorithm. In section 6 two final simulation scenarios and their configuration are shown.

2 SOFTWARE ORGANIZATION

2.1 High level workflow

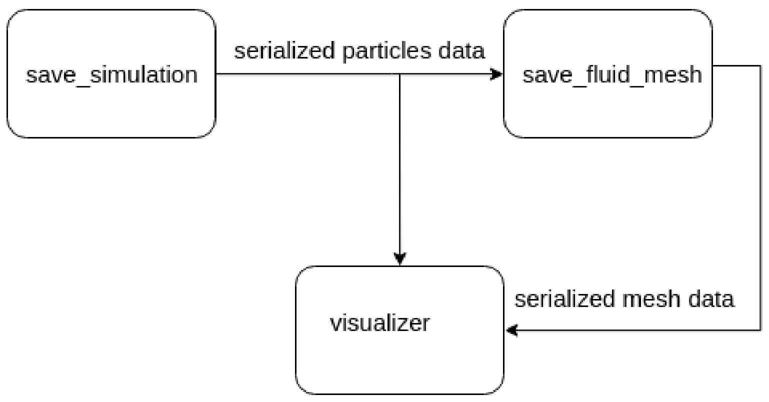


Figure 1: The workflow of our simulation software.

Basically, our software consists of three separate main executable programs. They are:

- ***save_simulation***
  This is the main simulation program. We at first set the simulation's configuration, like particles' number, total simulation time, the solvers and so on, by command arguments. Then this program runs the simulation. After finishing the simulation, it serializes the particle's data and outputs to the hard disk. In addition, we can also terminate this program in advance by typing CTRL+C, and the program will automatically store all data which have been computed so far and then close.
- ***save_fluid_mesh***
  This is the program which is responsible to reconstruct fluid surface mesh. It takes the serialized particles data as input and applies marching cube to generate the fluid surface.
- ***visualizer***
  This is the program which visualizes our simulation result. It has two modes. The first mode only visualizes particles data and only requires particles data as input. The second mode visualizes both particles and corresponding fluid meshes and thus requires both these two data. It also supports to visualizes multiple simulation scenarios simultaneously, which is helpful when we want to compare different configurations' performance.

The workflow is shown in Figure 1. First of all, save_simulation generates the particles data. Then we use visualizer to validate these particles data. If everything looks correct, then this particles data will be sent into the save_fluid_mesh and the fluid surface will be generated. Finally the visualizer is used again to show the mesh and particles.

2.2 Low level class family

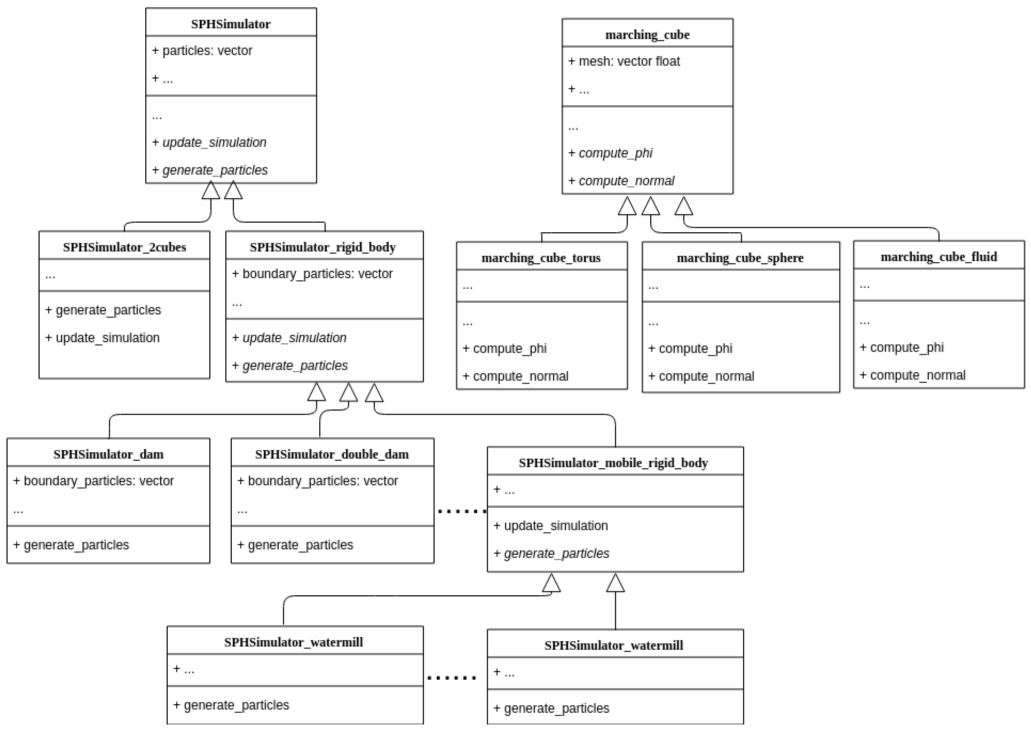


Figure 2: The inheriting tree of our simulation software.

At the very beginning of the course, we only have one simulator class and we put everything into this class. However, with the development of the lab course, we get more and more tasks and then his class became full of commented codes, switch and if branches, and many duplicated codes for different tasks. Our codes became messy. To solve this issue, we then use class inheritance techniques to extend our simulators and marching cube. Figure 2 shows the inheriting tree. We at first define a base class called *SPHSimulator*, and put every common method to this class. We also declare two virtual methods, *update_simulation* and *generate_particles* in this class. Then everytime when we want to extend new scenarios or different solvers, we only need to inherit a new class from the inheriting tree and override these two methods. This effectively helps us to avoid duplication of code and also makes our code easy to maintain and extend. For the marching cube, we apply the same technique. All these derived class are stored in the *derived_class* folder. We here should note, there do are some derived classes which do not obey the 'only override virtual functions' rule we mentioned above, and this is caused the limited time and our two coders' imperfect communication.

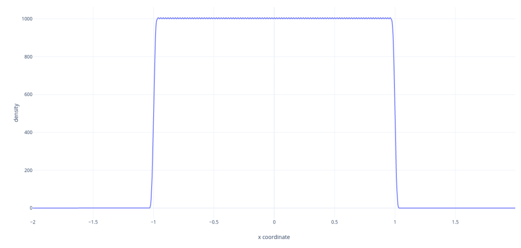
3 FLUID SIMULATION

3.1 Kernel functions

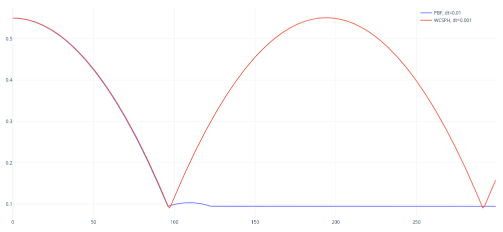
Before we go into the solvers for fluid simulation, let's look at the kernel functions first. The kernel functions for Smoothed Particle Hydrodynamics(SPH) define the behavior of the simulation, such as stability, viscosity and compressibility([Pri12]). In this course we've implemented three commonly used kernels: M4 cubic spline, M5 quartic and M6 quintic. Since kernel functions is the foundation of all our following work, we spent lots of time building unit tests to make sure our implementation is correct. Below we will introduce the unit tests we've done for this.

Unit tests

We created three family of unit tests and all are passed. The upper bound of roundoff error we use, says ϵ, is 10^{-6}.



(a) Density estimation test: density over the sample points



(b) Dropping single fluid particle on the boundary: height of the particle through the time

Figure 3: Unit test results

1. Unit tests for kernels: Are our results the same as manually-computed results?
   We pick one point per branch of the functions and do the tests.

2. Unit tests for kernels: Is the integration of kernel functions one?
   Instead of the analytical integration, we use Riemann Sum approach to approximate the integration. In detail, the smoothing length is divided into 100 equal sub-intervals. Therefore, there will be one million uniformly distributed sample points in the three dimensional domain of kernel functions.

3. Unit tests for gradient of kernels: Are the analytical results nearly the same as the approximate results?
   Here we compare the results using a relative criterion, says $\|\nabla_x W - \nabla_x^\epsilon W\| \leq \tau \|\nabla_x W\|$, where $\nabla_x W$ is our analytical solution and $\nabla_x^\epsilon W$ is the approximation. And we let $\tau = \epsilon$.
   To elaborate on our unit tests, we fix one point x and uniformly sample one hundred points along the x-axis within x's smoothing length. Then we compare the analytical and approximate results of gradients, given x and each sample point.

Issues we faced

1. Floating point precision
   At first we used FLOAT to do all the calculation, and the errors we got from unit tests above are around 0.01. Afterwards, we decided to change to DOUBLE and everything works fine since then.

2. Gradient function undefined at certain points
   If you look deeper into the gradient of kernels, you might find that $\nabla W(x, h) = \nabla W(x_i - x_j, h)$ is undefined at point $x = 0$. But, we took a while to notice this. To avoid numerical issue, we simply set $\nabla W(x, h) = 0, \text{if } \|x\| \leq \epsilon$. And here again, $\epsilon = 10^{-6}$.

3.2 Density estimation

The density estimation is the 2nd most important module in our fluid simulation, just after the kernel functions. Since it's built upon our kernel functions which is verified above, here we only have one test case.

Unit tests

The only unit test we've done is, we put a $2 * 2 * 2$ cube of water at the origin and sample the density along with x-axis from $(-2, 0, 0)$ to $(2, 0, 0)$ using 1000 sample points. The result is shown in Figure 3a. Besides, all parameters are the default values.

3.3 WCSPH and PBF

In this section, we will introduce the verification of our implementation of WCSPH([BT07]) and PBF([MM13]). To do so, we apply two scenarios on both solvers, shown as follows.

Test problems

Since the scenarios are so simple that it's hard to see something from just one or two screenshots, we decide not to put any screenshot here. Instead, we suggest to repeat both test scenarios using our program if you are interested. Please find more details in the README of the repo.

1. 2-cube collision
   First, we start our testing with a scenario without boundary, so that we can verify our implementation more easily. And we use the simplest scenario we can think of, two colliding cubes of water. Then we visually check the result and see if it acts as what we expect. And it does, indeed. The parameters we use are again the default values. Gravity and boundary are excluded. And cases with and without artificial and XSPH viscosity are tested.

2. Dropping single fluid particle on the boundary
   After passing the first test, now we apply the boundary. Again, we use the simplest scenario we can think of, dropping single fluid particle on the boundary. Before the test, we ensure the boundary is symmetric from the fluid particle's point of view, so that the fluid particle will only move along the z-axis. Here we visualize the z-coordinate of the particle over time in Figure 3b. From the result, we can see WCSPH and PBF involve totally differently mechanisms. WCSPH acts more like a spring, always restoring the particle to the original height. In the other hand, PBF applies constraint on the density of the particle and forces particles to stick to the place which makes its density to be rest density. In our case, it's on the boundary.

3.4 WCSPH versus PBF

Now we have WCSPH and PBF these two different solvers for fluid simulation. And because we are seeking a way to give us a more fluid-like simulation, we are curious about which solver generates a better quality of simulation result.
But, here comes a question: How to evaluate if a simulation is more realistic? And **Compression Rate** will be the answer.

Definition 1. *Compression rate is defined by*

$$\frac{\rho}{\rho_0}$$

Before we explain why it works, let's recall one of the properties of water: incompressibility. Incompressibility means that even if we put a force to a chunk of water, the volume of water will keep unchanged. So, in order to get a more fluid-like result, the property should be maintained as possible as we can.

Experiment result

And now let's get back to the compression rate. Here the rest density ρ_0 is a constant and the density ρ can be interpreted as the extent the area around the particle being compressed. Therefore, high compression rates imply the incompressibility is violated, which further implies less fluid-like simulation.

In this part we will talk about the experiments we've done to evaluate simulation results from both WCSPH and PBF, based on compression rate and time performance. In our experiment, gravity, artificial viscosity and XSPH viscosity are applied. Besides, the parameters we use are the default values mentioned in the exercise sheets, except the timesteps and stiffness, which will be specified later.

Our experiment setup is simple: we drop a chunk of water from two different heights inside a pillar and see the max compression rate we get at each iteration. The preview of the experiment is

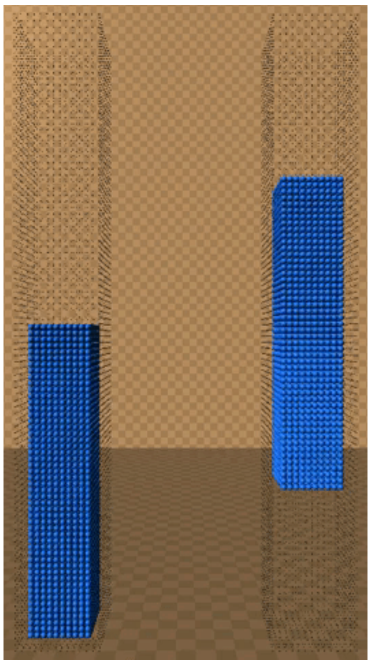
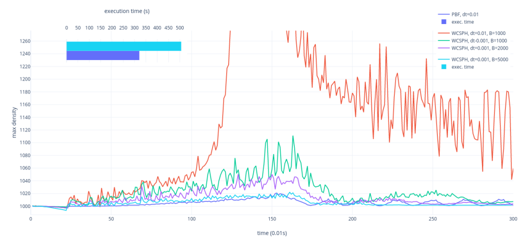
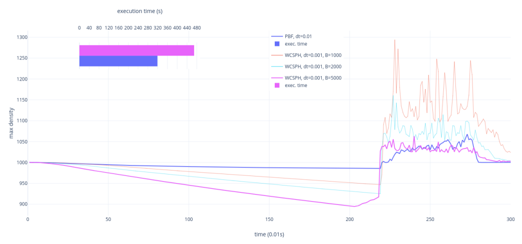


Figure 4: Experiment screenshot



(a) result from the experiment with the lower height



(b) result from the experiment with the higher height

Figure 5: Results: max compression rate and time performance

shown in Figure 4. From this experiment, we expect to see how the max compression rate is affected by different solvers and parameters, and also how much time both solvers need to get the same max compression rate. One thing to note is that we will interchange *max compression rate* and *max density* in the following content.

Now let's look at the case with the lower height, whose result is shown in Figure 5a. The first thing can be told from the result is that PBF works fine with $\Delta t = 0.01$, but WCSPH explodes. However, WCSPH becomes stable when the timestep drops to 0.001, which further allows us to increase the stiffness. And when the stiffness reaches 5000, the max densities from WCSPH and PBF are nearly the same (1022.0 and 1020.8, respectively).
So far, we've realized WCSPH with stiffness 5000 and timestep 0.001 leads to similar max compression rate which PBF with timestep 0.01 gives. And given the similar results, an evaluation of time performance becomes possible, and we find that WCSPH takes 57% longer than PBF.

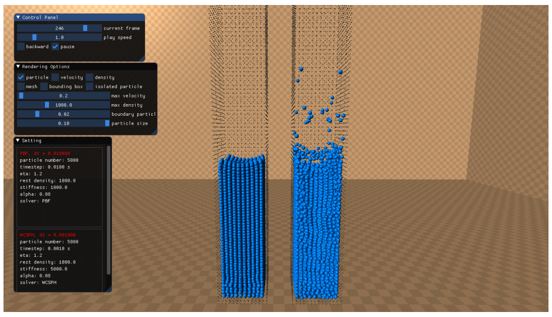
In the case with the higher height, we got very similar result to the above one, and it's shown in Figure 5b. The max compression rates from WCSPH with stiffness 5000 and timestep 0.001 and PBF with timestep 0.01 are still almost the same at this time. On the other hand, the execution time of WCSPH is 47% longer than that of PBF. Here we skipped WCSPH with timestep 0.01 since we'd already learned that it would explode.

Experiment conclusion

From the experiment results, we learned that PBF is always comfortable with timestep 0.01, but WCSPH isn't. However, if the timestep of WCSPH drops to 0.001, we are able to get the similar result PBF gives by increasing the stiffness of WCSPH to 5000. And because smaller timesteps imply more iterations are demanded to simulate same amount of time, that's why WCSPH needs to run 10 times more iterations than PBF does in our cases and therefore, takes more time.
But since we didn't test every possible parameters, we believe it can be possible that WCSPH actually allows timesteps to be somewhere between 0.01 and 0.001. If it's the case, then we would have a chance to use a rather small stiffness value (<5000) to get the same result we had now, but with shorter execution time.
To sum up, PBF is always comfortable with larger timesteps. Regarding the execution time, we are unable to tell if WCSPH is possible to beat PBF or not. Even if the answer is yes, the parameter tuning involves the timestep and the stiffness these two variables, which makes the finding of optimal settings much tougher.

Experiment discussion

The discussion is divided into two parts. First, we just want to show that even the max compression rates are similar. The results from PBF and WCSPH still look very different, as shown in Figure 6a. Second,



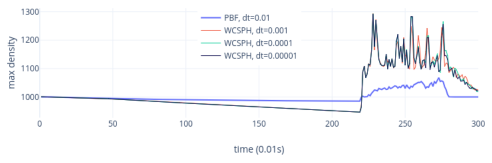
```
246   current frame
1.0   play speed
☐ backward ☑ pause

☑ particle ☐ velocity ☐ density
☐ mesh ☐ bounding box ☐ isolated particle
0.2      max velocity
1000.0   max density
0.02     boundary particl
0.10     particle size

PBF, dt = 0.010000
particle number: 5000
timestep: 0.0100 s
eta: 1.2
rest density: 1000.0
stiffness: 1000.0
alpha: 0.08
solver: PBF

WCSPH, dt = 0.001000
particle number: 5000
timestep: 0.0010 s
eta: 1.2
rest density: 1000.0
stiffness: 5000.0
alpha: 0.08
solver: WCSPH
```

(a) visual results. **Left:** from PBF. **Right:** from WCSPH



(b) the result from only changing timesteps

Figure 6: Experiment discussion

you may find that this experiment is conducted in a little bit different way, compared with the one we showed in the final presentation. It's because we had not understood the relation between timestep and stiffness until one of your colleague told us. Before that, we just simply reduced the timesteps of WCSPH and got results shown in Figure 6b.
So now let's have an insight into these two parameters. First, the timestep is responsible for the simulation stability the most since it directly affects the velocity and position of particles. That's why minimizing the timestep is the most effective way to avoid the simulation from explosion. But once we have a rather stable simulation, continuing decreasing the timestep won't produce a better result, as shown in Figure 6b. It's because the extent of restoration of particles from the state of being compressed becomes constant in the same amount of time. What to do instead is to enlarge the stiffness so that the restoration becomes faster, which further makes compression rates drop, as shown in Figure 5.

4 BOUNDARY GENERATION

4.1 Introduction of rigid boundary

We apply the method from Akinci, et, al[Aki+12] to calculate the rigid boundary' force on our fluid particles. Basically, this approach treats the boundary also as particles. Thus it is easy to integrate the force from boundary particles into the whole particles force equation. Besides, particles are easy to be set at arbitrary positions to form boundary different shapes. Furthermore, comparing with other particle-based boundary methods, this approach derives a new equation where each boundary particle has a different relative contribution to the force. This relative contribution depends on the sampling density of boundary particles. This approach also allows boundary particles not uniformly distributed.

4.2 Our generating boundary method

We use implicit surface representation to generate our boundary. Implicit surface representation represents the surface by:

$$F(x, y, z) = 0$$

For example, a unit radius sphere can be represented by:

$$F(x, y, z) = x^2 + y^2 + z^2 - 1 = 0$$

Algorithm 1 shows the procedure we generate the sphere boundary particles. Note here it does not only build the surface but also fill particle insides sphere. Since the method from [Aki+12] compensates the boundary particles' sampling density, then solid boundary actually has the same impact as the hollow boundary. Our boundary generating algorithm is not efficient. Given the radius r, the time complexity could be $O((r/h)^3)$. However, since usually we generate boundary particles only one time per simulation, the time complexity is not the key factor. Except this, it also has two advantages.
First, this is a general method to generate boundary surface particles. Other surfaces can be simply generated through this procedure by replacing the $x*x+y*y+z*z-r*r < 0$ in line 6 with these surfaces'

Algorithm 1 Generate sphere boundary particles

Input: Sphere radius r, sampling interval h
Output: a vector of boundary particles

```
 1: load particles from the simulation data
 2: $boundaryParticles \Leftarrow [\,]$
 3: for z = -r; z <= r; z = z + h do
 4:   for y = -r; y <= r; y = y + h do
 5:     for x = -r; x <= r; x = x + h do
 6:       if x*x+y*y+z*z-r*r < 0 then
 7:         $boundaryParticles.push\_back(p(x, y, z))$ // this x,y,z is inside the sphere
 8:       end if
 9:     end for
10:   end for
11: end for
12: return boundaryParticles =0
```

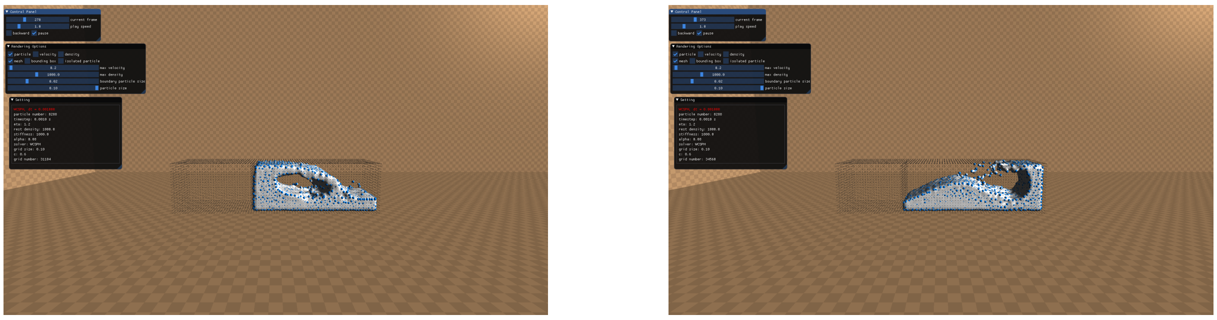


Figure 7: Screenshots of selective moving boundary demo: wave simulation

implicit representation. Implicit representations are also usually easier than explicit representation to find. For example, to generate a heart shape boundary, we can use:

$$F(x, y, z) = (x^2 + 9y^2/4 + z^2 - 1)^3 - x^2z^3 - 9y^2z^3/80 = 0.$$

Second, the sampling interval h in our generating boundary algorithm can guarantee that the maximum distance between two adjacent boundary particles is $\sqrt{3} * h$. Therefore, by setting the sampling interval h small enough, our algorithm can guarantee a minimal density of the boundary particles at the whole surface, which makes sure the fluid particles cannot easily penetrate the boundary in any place of the surface.

4.3 Moving boundary

In our final presentation, we've shown the demo of moving boundary. Here we will further explain the details of our implementation.
Recall that both the volume and the (pseudo) mass of boundary particles could be involved in the computation of boundary handling, depending on the solver type. And according to the equation as follows, these two quantities can be easily reconstructed from either one of them. Therefore, we can just store the pseudo mass in our Particle struct.

$$m = \rho_0 V$$

The only difference between handlings of static and moving boundary is that updating the pseudo mass and volume of boundary particles is required in every iteration if moving boundary is applied. As a result, we can adapt our simulator in the following way to the moving boundary.

```
class moving_rigid_body : public static_rigid_body {
    virtual void update() override {
        update_boundary_mass();
        static_rigid_body::update();
        update_boundary_movement();
    }
}
```

4.4 Outlook

Our next step is to enable the boundary particles to move according to the force from the fluid particles. [Aki+12] also mentioned the method to compute the force from fluid particles to boundary particles. Because of the symmetry of the force, the force from fluid particles to boundary particles are exactly the force from boundary particles to fluid particles in a negative direction. Knowing the force and position of each boundary particles, we then can simply get the rigid boundary's total force and torque. Thus by applying the total force and torque, we can make the rigid boundary move according to the force from fluid particles.

5 MARCHING CUBE

Marching cube is a classic algorithm to generate triangle meshes. We apply this method to reconstruct our fluid surface. The simplest version of the marching cube which is mentioned in the exercise sheet 5 has at least third issues. First of all, the simplest version's memory consumption is very high. Each vertex and edge of marching cubes should be shared with several voxel grids. However, this simplest version directly generates voxel grids, and each voxel grid contains its own instances of eight edges and twelve vertices. Memory consumption for these duplicate edges and vertices are unnecessary. Besides, it stores all voxel grids in the bounding box. However, most of the grids are not intersected with the surface. Memory for these voxel grids can also be saved. Second, the output mesh of this simplest version contains duplicate vertices. This is because the duplicate edges of voxel grids generate several copies of the same mesh vertex. To solve these two issues, we design our own special data structure and procedures. Third, the bounding box is fixed. This may cause the particles to be excluded during rendering. Also it's not robust. Users have to set up the size of bounding box every time when a new scenario comes. Last but not least, it leads to large memory consumption. In the following we first talk about this data structure and new procedure, and then we move to the dynamic bounding box.

5.1 Data structure and procedure

Data structure

```
struct mVoxelVertex{
    Vector3f position;
    double phi;
    bool is_inside;
};

struct mVoxelEdge{
    mVoxel_vertex * vertex1_ptr;
    mVoxel_vertex * vertex2_ptr;
    bool has_mesh_vertex;
    mMesh_vertex * meshVertexptr;
};

struct mVoxel{
    std::array<mVoxel_vertex *,8> vertex_ptrs;
    std::array<mVoxel_edge *, 12> voxel_edges_ptrs;
    unsigned int bitcode;
};
```

The key point in this data structures is that we now only store pointers in the voxel gird(*mVoxel*) object. Using pointers first makes sure these grids not to duplicate edges(*mVoxelEdge*) and vertices(*mVoxel*). Second pointers are cheaper than a real instance of edges or vertices, which also make the memory cost of one grid decrease.

Procedure

1. Given the bounding box and marching cube resolution, the program first initializes and distribute all vertices(*mVoxelvertex*) uniformly in this bounding box.
2. The program then initializes all edges(*mVoxelEdge*). For each edge, it assigns two pointers of the edge's corresponding two vertices.
3. Next, the program initializes voxel grids(*mVoxel*). For each voxel grid, it again assigns the pointers of edges and vertices. Furthermore, the program also checks if this voxel grid is intersected with the surface by seeing if its eight vertices are not all inside or outside and then it only stores these intersecting grids.
4. Finally, the program iterates over these stored voxel grids, computes the bitcode and generates the mesh.

Result

Table 1 shows these three struct's size. As we can see, if we let the *mVoxel* not contain pointers but the instances of *mVoxelVertex* and *mVoxelEdge*, its size should be at least 648 bytes, which is nearly 4 times bigger than the current size.

We then test our algorithm by generating a sphere. The bounding box size is fixed in 3*3*3, and the resolution is 0.01. Table 2 shows the number of *mVoxel* before and after checking intersecting. We calculate the total size by multiplying the number of *mVoxel* and the unit size of *mVoxel*(168 bytes). As we can see the latter is nearly 100 times smaller than the former number. This checking intersecting is extremely effective. Table 3 shows the number of output mesh's vertices. The number in the simplest version is exactly 4 times the number in our version. This result is expected because 1 *mVoxelEdge* is exactly shared by 4 *mVoxelEdge*, and each *mVoxelEdge* exactly decides 1 mesh vertex.
Figure 8 shows different visualization of mesh with and without duplicate vertices. The most left two are the meshes with duplicate vertices and the right two are the meshes without duplicate vertices. The most left one and the middle right one use Merely3D to compute their normal vectors. The middle left one and the most right one have normal vectors which we compute according exactly to the sphere. As we can see, the visualization of two spheres which use sphere's normal vectors is exactly the same, and

	mVoxel	mVoxelVertex	mVoxelEdge
size in byte	168	32	32

Table 1: Size of three structs.

	before checking intersecting	after checking intersecting
number of mVoxel	27,000,000	188,428
total size (byte)	4.5 G	31.7 M

Table 2: The number of mVoxel, before and after checking intersecting.

	The simplest version	our version
number of output mesh's vertices	753,704	188,426

Table 3: The number of output mesh's vertices.

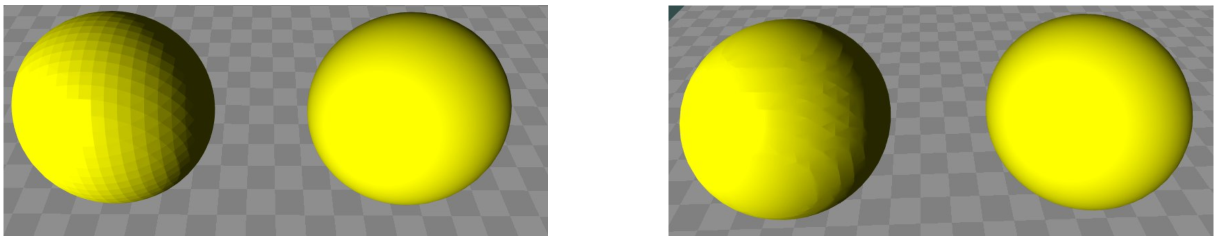


Figure 8: Visualization of mesh with duplicate and not duplicate vertices.

the visualization of two spheres which use normal vectors from Merely3D is different. The reason is, in Merely3D's computing vertex's normal vector method, every adjacent triangle of this vertex has a contribution. However, in the duplicate vertices case, triangles and triangles are not exactly connected with each other, and therefore for each vertex there exactly is only one adjacent triangle. We should note here, removing duplicate mesh vertices is not only for saving memory, but also for generating the real triangle mesh where every triangle is connected with each other. There are many mesh-based algorithms which can furtherly improve our fluid surface, and most of them require that triangles in this mesh should be connected with each other.

5.2 Dynamic bounding box

The bounding box plays an important role in the marching cube algorithm, in terms of both execution time and the render result. It affects the size of the grid, which further influences the number of times we need to compute ϕ and also the area which will be rendered. Therefore, we can expect that minimizing the bounding box could reduce the execution time but it might also cause the loss of particles being rendered. On the other hand, maximizing the bounding box makes it cover more particles but the running time also increases. In this section, we will introduce our method "Dynamic Bounding Box" and explain how it finds a happy medium.

Algorithm

Our algorithm is sketched out in Algorithm 2. In short, for each frame, we first skip isolated particles(, whose density is less than 185 by applying the cubic kernel function) when loading the serialized simulation data. Then we compute the bounding box and makes it tightly cover the rest of particles. At last, we check if the discarded particles are inside the bounding box or not. If they are, we will pick them up again, so that we don't lose too much details during rendering.

Experiment result

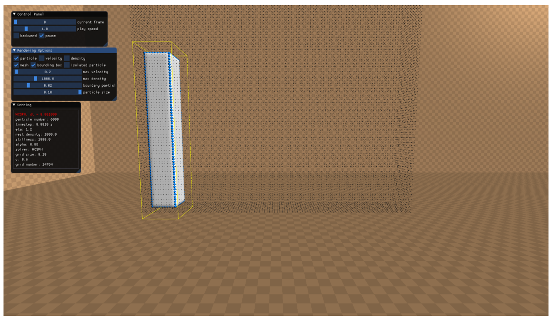
We test our dynamic bounding box by using a dam break simulation and compare it with the fixed one, based on how many voxels are saved and how many particles are lost.

We emphasize better render result more than the shorter execution time. Therefore, our experiment is designed as follows: we fix the number of lost particles in every frame and see how many voxels do both dynamic and static bounding boxes need in order to cover the same amount of particles. The parameters we use are the default values mentioned in the exercise sheets, except PBF's $\Delta t = 0.01$ and WCSPH's $\Delta t = 0.001$. And the number of particles is 6000. The screenshot of the experiment setup can be found in Figure 9a.

First let's look at the results from PBF, shown in Figure 10a. In this experiment, the number of lost particles is always zero through the whole process. And the picture shows that to achieve the same render result, fixed bounding box always needs more voxels than dynamic one. Especially, it needs 10 times more when the simulation get mild in this scenario. This fact implies that dynamic bounding box is more robust to deal with different scenarios.

Algorithm 2 Dynamic Bounding Box

```
Input: simulation data of a given frame
Output: the bounding box and particles which are going to be rendered
 1: load particles from the simulation data
 2: $renderParticle \Leftarrow [\,]$
 3: $discardParticle \Leftarrow [\,]$
 4: for p in particles do
 5:     if p is not isolated then
 6:         $renderParticle.push\_back(p)$
 7:     else
 8:         $discardParticle.push\_back(p)$
 9:     end if
10: end for
11: build bounding box according to the $renderParticle$
12: for dp in $discardParticle$ do
13:     if dp is inside the bounding box then
14:         $renderParticle.push\_back(dp)$
15:     end if
16: end for
17: return $bounding\_box, renderParticle$ =0
```


(a) experiment setup. **Yellow box:** our dynamic bounding box

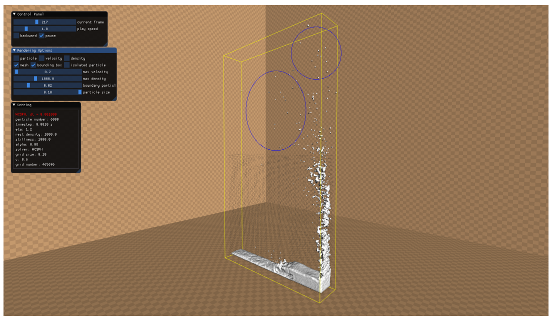
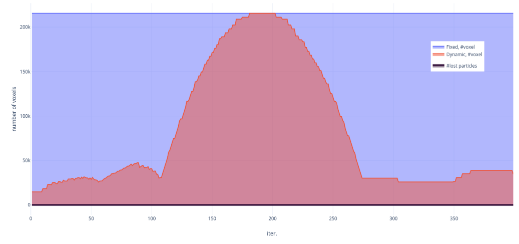
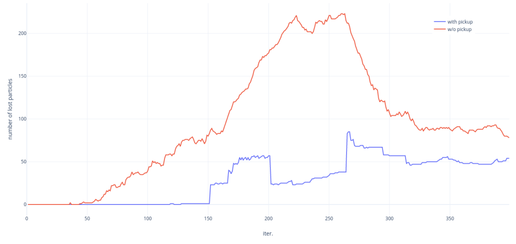

(b) render result. **Yellow box:** our dynamic bounding box. **Blue Ellipse:** the mesh credited to the re-pick mechanism

Figure 9: Screenshots of bounding box experiment



(a) compare #voxels between dynamic and fixed bounding box using PBF

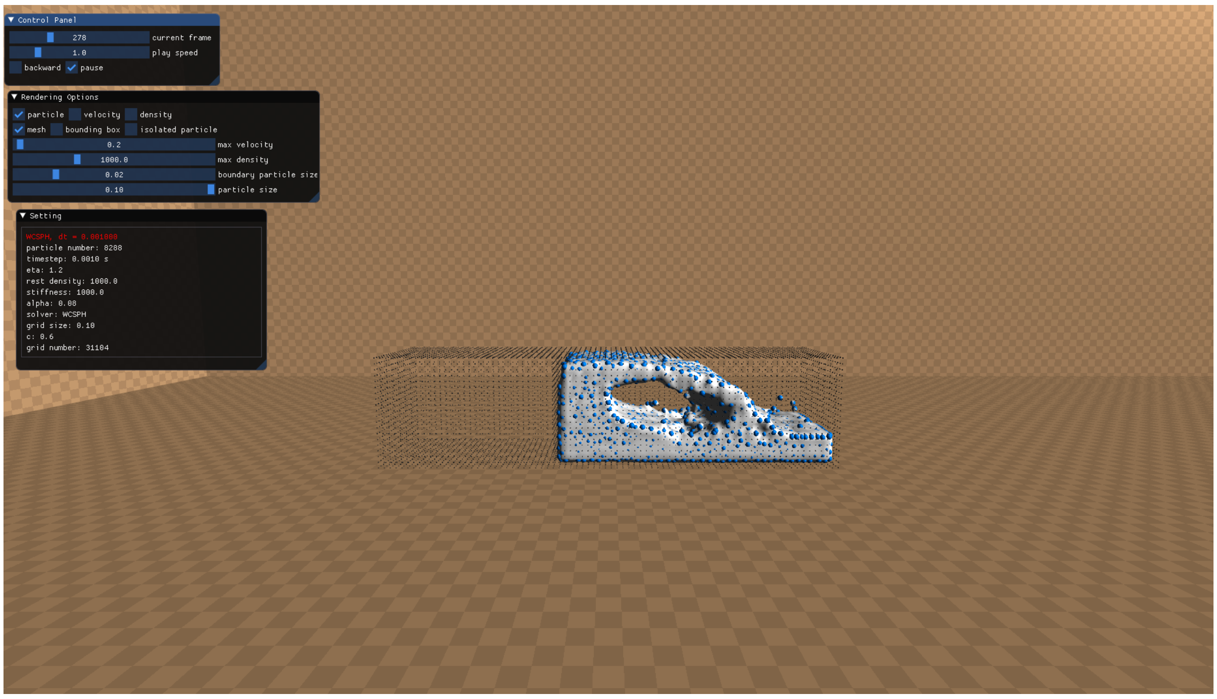


(b) compare #lost_particles between dynamic bounding box with and without pickup mechanism using WCSPH

Figure 10: Results of bounding box experiment over dam break scene

In the above experiment, we've seen how dynamic bounding box improves the efficiency. And here we will further evaluate the re-pickup mechanism, regarding the number of lost particles. However, from the last experiment we found that PBF didn't give us exploding particles, so here we use WCSPH instead to get some exploding particles to evaluate. The inspiration of this mechanism comes from the observation of large amount of missing particles (and therefore, missing meshed) after we visualized them. And what's more is, a big portion of them are actually inside the bounding box we built. That's the reason why we decide to add the re-pickup mechanism. Figure 10b shows the reduction of lost particles the re-pick mechanism gives us. The max loss rate (defined by #lost_particle/#total_ particles) drops from 3.7% to 1.4%, which implies the re-pickup mechanism produces a more detailed rendering output based on the same number of voxels being considered. The screenshot of this experiment result is shown in Figure 9b.

6 DEMO SCENARIOS



```
278        current frame
1.0        play speed
[ ] backward  [✓] pause

[✓] particle  [ ] velocity  [ ] density
[✓] mesh  [ ] bounding box  [ ] isolated particle
0.2        max velocity
1000.0     max density
0.02       boundary particle size
0.10       particle size

WCSPH, dt = 0.001000
particle number: 8288
timestep: 0.0010 s
eta: 1.2
rest density: 1000.0
stiffness: 1000.0
alpha: 0.08
solver: WCSPH
grid size: 0.10
c: 0.6
grid number: 31104
```

Figure 11: Wave simulation.

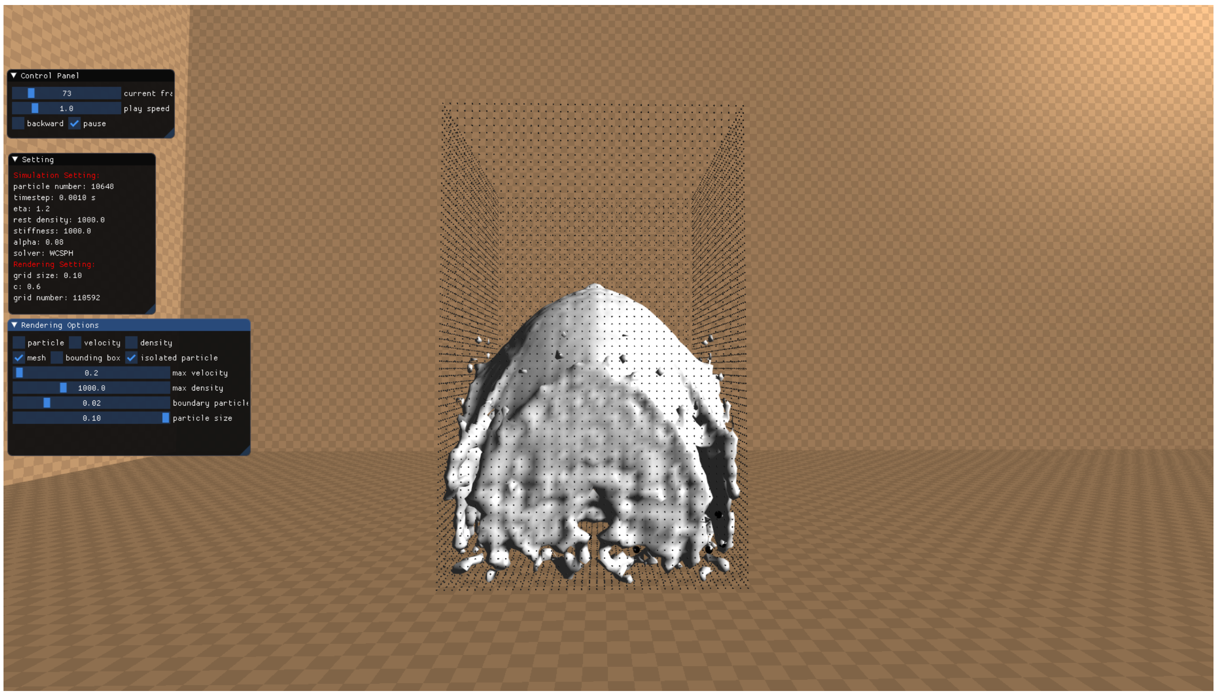
References

[Aki+12] Nadir Akinci et al. "Versatile rigid-fluid coupling for incompressible SPH". In: *ACM Transactions on Graphics (TOG)* 31.4 (2012), p. 62.

[BT07] Markus Becker and Matthias Teschner. "Weakly compressible SPH for free surface flows". In: *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation.* Eurographics Association. 2007, pp. 209–217.

[MM13] Miles Macklin and Matthias Müller. "Position based fluids". In: *ACM Transactions on Graphics (TOG)* 32.4 (2013), p. 104.

[Pri12] Daniel J Price. "Smoothed particle hydrodynamics and magnetohydrodynamics". In: *Journal of Computational Physics* 231.3 (2012), pp. 759–794.



```
73 current fra
1.0 play speed
backward ✓ pause

Simulation Setting:
particle number: 10648
timestep: 0.0010 s
eta: 1.2
rest density: 1000.0
stiffness: 1000.0
alpha: 0.08
solver: WCSPH
Rendering Setting:
grid size: 0.10
c: 0.6
grid number: 110592

particle   velocity   density
✓ mesh   bounding box   ✓ isolated particle
0.2 max velocity
1000.0 max density
0.02 boundary particle
0.10 particle size
```

Figure 12: Sphere boundary.