2. REQUIREMENTS

We discarded the last step of our project due to our access has inhibited to the computer laboratory in the university because of the COVID-19. We discussed with our project advisor and agreed on discarding the implementation of Marching Cubes algorithm requirement.
The final requirements of the project are provided in Appendix A: Requirements Specifications Document, revision 3.0 (RSD 3.0). All requirements are on the table below.

Requirement Type	**Requirement Name**
Functional Requirement	Retrieve Particle Data
Functional Requirement	Divide into Cells
Functional Requirement	Surface Recognition
Functional Requirement	Marching Cubes
Non-Functional Requirement	Efficiency
Non-Functional Requirement	Performance
Non-Functional Requirement	Usability
Non-Functional Requirement	Testability

Table 3: Requirements list.

3. DESIGN

This section describes about design of the POF system. High level and detailed designs are explained. Restrictions and conditions mentioned in this section.

3.1. High Level Design

High level design is explained in detail and a sequence diagram is included in DSD 2.0.

3.1.1. Class Diagram

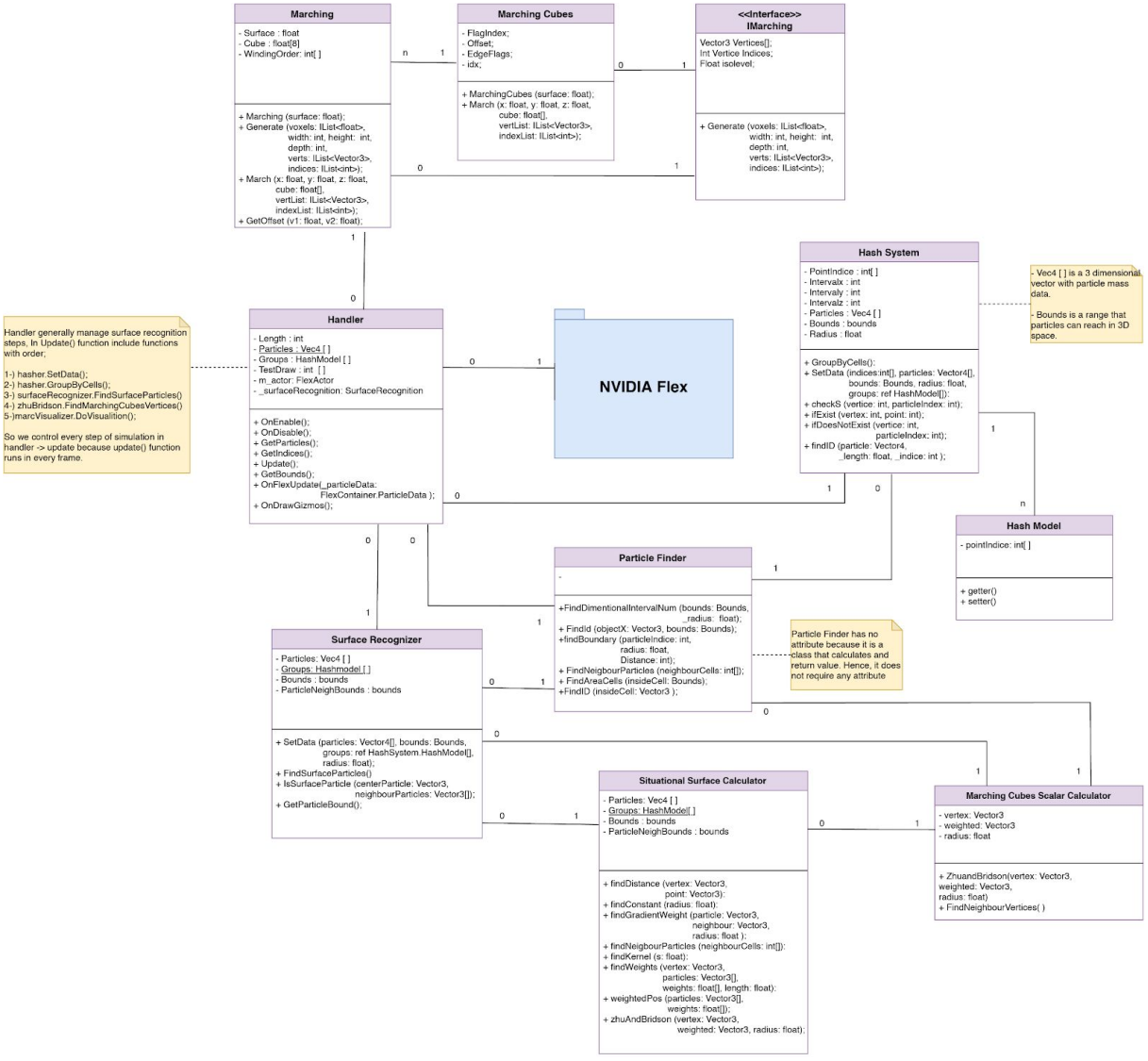

Fig 1: Class Diagram

The class diagram above explains classes and their attributes along with relations. NVIDIA flex is obligatory and an external source asset. Handler class is the main controller of the system. Handler communicates with classes transmit data to relevant classes.

Hash System class is responsible for the classification and finding particles. Particle Finder class returns the particle data such as position, particle id or particle's cell. Surface recognizer marks the surface particles when called.
Three classes on the top are related with Marching Cubes algorithm which is responsible for visualizing the particles in a different way. We wrote the Marching Cubes algorithm as a C# script in our project. However, we could not implement the Marching Cubes method into the POF system for obvious reasons (Please read warning page!). You can visit the design specifications document revision 3.0 (DSD 3.0) [2] for detailed explanations of class diagrams.

3.2. Detailed Design

The detailed design part is mentioned in the DSD 3.0. In DSD 3.0 [2], detailed design is explained with activity diagrams of the POF system.

We explained those substructures as four parts of activity diagrams. First activity diagram is Marching Cubes Scalar Calculator which calculates the scalar value for the marching cubes algorithm. Second activity diagram is Surface Recognizer which determines the surface particles. Third activity diagram is Marching Cubes which analyses the surface vertices and decides which configurations of triangles will be sent to draw. Last activity diagram is Hash System which finds cell of a particle, cell id, particle id etc. Hash system is helping to find data briefly.

3.3. Realistic Restrictions and Conditions in the Design

We neglected some aspects of the project on purpose to finish the project on time. The security problems are ignored because the project aims to help everybody who has interested fluid simulations and contribute to science. Normal computers are not capable of running the simulation fluently. User can not increase particle number without sacrificing the performance with normal computers. Simulation computer is required for this job. We assumed that users of the POF system have the necessary equipment and software and know-how to use them.

4. IMPLEMENTATION and TESTS

This section describes the tests and implementation stages of the POF system. Test results are discussed in the performance tests and results title. In the implementation of the POF system title, we shared important code snippets and explained the logic behind it.

4.1. Implementation of the POF System

4.1.1. Problems and Solutions

Firstly, we had to recognize elements of NVIDIA flex, and then we discussed how to classify the particles and wrote the hash system. After, we research methods of identifying surface particles and discussed how to apply it. Last part we wrote a prototype code of the Marching Cubes algorithm to implement into POF system.
We explained how implementations made and code explanations in the product manual 1.0.

Main Problem	**Solution of Problem**
Searching particle data linearly due to 3D space positions and vector3 to integer translation.	Spatial hashing algorithm provides reaching particles by put them into cell data.
Too many particles appear in simulations and handling all of them occurs performance problems.	Do not put into calculations inactive and unnecessary particle on visualization (surface particle finding algorithm).
Duplication of functions and similar methods.	We created a situational surface calculator class and particle finder classes to remove code duplication and increase the functionality of our code.
Increasing inter-community and data transmission complexity.	We constructed the Handler class which is responsible for all data transmission and user controls interface.

Table 4: Problems and Solutions

4.1.2 Implementation of Hash Algorithm

We use a spatial hash algorithm to access the particle position easily. Hash algorithm simplifies the three dimensions of float particle positions to integer id numbers in a specific order.

Cube numbering starts from top left and from left to right and then top to bottom. Then it implies the same operation for the third dimension. On x dimension, we found cell id by subtracting minimum boundary area x from particle x, so it means that numbering increases from left to right. Same logic implies on y dimension, cell id number

increase from top to bottom and similarly cell id number increases from backwards to forward in the z dimension.

We subtracted one from the xid, yid and zid when we are assigning an integer number because we initialize id numbering from zero. Cubex represents how many particles can be fit into the cell. We apply these calculations for all dimensions. We can reach cell id by position without storing it.

```
int xId = (int)Math.Ceiling((partcle.x - _bounds.min.x) / _length) - 1;
int yId = (int)Math.Ceiling((_bounds.max.y - particle.y) / _length) - 1;
int zId = (int)Math.Ceiling((particle.z - _bounds.min.z) / _length) - 1;
float cubeX = (particle.x - _bounds.min.x) % _radius ;
float cubeY = (_bounds.max.y - particle.y) % _radius;
float cubeZ = (particle.z - _bounds.min.z) % _radius;
```

Fig 2: Cell ID Numbering

Vertex index struct represents cells by creating an array on this struct. We keep each vertex as 3d vector struct. We hold index point data in integer list.

```
public struct vertexIndex
{
    public Vector3 vertexs;
    public List<int> pointIndice;
}
```

Fig 3: Group Struct

```
public void FindDimentionalIntervalNum(Bounds bounds, float radius)
{
    this._intervalx = (int)Math.Ceiling((_bounds.max.x - _bounds.min.x) / _ra-
dius); // Calculate the small cube number in x-axis.
    this._intervaly = (int)Math.Ceiling((_bounds.max.y - _bounds.min.y) / _ra-
dius); // Calculate the small cube number in y-axis.
    this._intervalz = (int)Math.Ceiling((_bounds.max.z - _bounds.min.z) / _ra-
dius);
    // Calculate the small cube number in z-axis.
}
```

Fig 4: Hash Size

We created cell ids as indices. By finding how many grids in each dimension we find our hash table size represented as figure 4. We find particles by looking at cells.

We have realized a problem occurs in this section during the implementation. If particle centre intersects with cell boundaries, we must decide to assign which particle will be in which cell. We solve this problem by logically assigning particle closest previous cell. If you imagine many neighbour cubes such as small cubes constructing a bigger cube such as 64 cubes, in the centre intersection point there are 8 possible cubic cells you can assign the intersecting particle. These if cells are looking for all intersecting possibilities for all dimensions. There are 2 to the power of 3 cases because of the spatial space we are dealing with. We did not share the whole code but give a snippet to make more understandable. If there is an intersection, we assign a previous cell. To give an example, let us assume a particle intersects with 2 cells. These cells are called cell id 1 and cell id 2. We assign particle to smaller cell id which is cell id 1.

```
// Five errors in here for fill to fix
        if (cubeX == 0 && cubeY == 0 && cubeZ == 0)
        {
            if (xId > 0 && yId > 0 && zId > 0)
            {
                cubeID = (xId--) + (this._intervalx * (yId--)) + (this._inter-
valx * this._intervaly * (zId--));
                checkS(cubeID, _indice);
            }
            if (xId > 0 && yId > 0)
            {
                cubeID = (xId--) + (this._intervalx * (yId--)) + (this._inter-
valx * this._intervaly * (zId));
                checkS(cubeID, _indice);
            }
```

Fig 5: Intersection Boundaries Check.

For example, in fig. 5. we found that particle on intersection boundaries and we found its second cell on y dimension

So that way we can find the particles by looking in cells instead of a linear search of the particles. The ceiling is implemented to derive integer numbers because particles are float and they are derived according to the AABB size.

4.1.3 Particle Neighbour Algorithm

We must find the effect of the particles in a range on a specific particle and calculate it for each particle. To think mathematically consistent and coherent with the particles, the shape of the volume that we are checking should be spherical. However, the required time and finish the project according to the given time interval has compelled

project team to search cubical volumes. We must look at the hashed cell ids in the volume that we are searching to find neighbour particles.
We need four corner cells to solve the problem of finding neighbour particles.

```
int topLeftBackward  = FindID(new Vector3(insideCell.min.x, insideCell.max.y,
insideCell.min.z));
int topLeftForward = FindID(new Vector3(insideCell.min.x, insideCell.max.y,
insideCell.max.z));
int topRightBackward = FindID(new Vector3(insideCell.max.x, insideCell.max.y,
insideCell.min.z));
int bottomLeftBackward  = FindID(new Vector3(insideCell.min.x, insideCell.min.y,
insideCell.min.z));
```

Fig 6: Finding Corner Cells.

These three cell borders are called as tx, ty and tz.

```
int tx = (topRightBackward - topLeftBackward),
    ty = ((bottomLeftBackward - topLeftBackward) / _intervalx),
    tz = ((topLeftForward - topLeftBackward) / (_intervalx * _intervaly));
```

Fig 7: Finding Dimensional Cell Count

We find grid intervals that cells are going towards in every dimension. So that way we can find all cells in this volume just by looking at one cell that we have grouped neighbour particles in that cell.

```
int[] areaNums = Enumerable.Repeat(-1, ( (ty+1) * (tx+1) * (tz+1)) ).ToAr-
ray();
int i = 0, tempNum = topLeftBackward;

for (int k = 0; k < ty; k++)
{
    for (int j = 0; j < tz; j++)
    {
        for (int m = 0; m < tx; m++)
        {
            areaNums[i] = tempNum + m;
            i++;
        }
        tempNum += (_intervalx * _intervaly);
    }
    tempNum = areaNums[0] + (_intervalx * (k + 1));
}
return areaNums;
```

Fig 8: Finding Cell Numbers in Specific area.

4.2 Performance Tests and Results

These performance tests are made in the Unity platform to test our POF system and the performance of its subsystems. We have created a corridor with two obstacles (a capsule and a sphere objects) and released the particles (4096 particles) above a ramp. During the particles are moving in the test scene, we recorded surface recognition performance execution time frame by frame. We have recorded 100 frames and the particles go into a stable position rest of the frames in the scene. That is why we did not record after the particles are in halt position because it would not be correct measuring. The time per frame values could be wrong.

In the test scene, we aim to change the position complexity of the particles. So that why we could create a different combination of position data cases between particles and calculation parameters in the surface recognition system.

We divided the test scene into five phases. The initial position of the test scene is called perfect cube phase as you can see in the figure below. When the user pressed the play button, the particle set will be released, and particles will scatter. Inner particles are painted to blue.

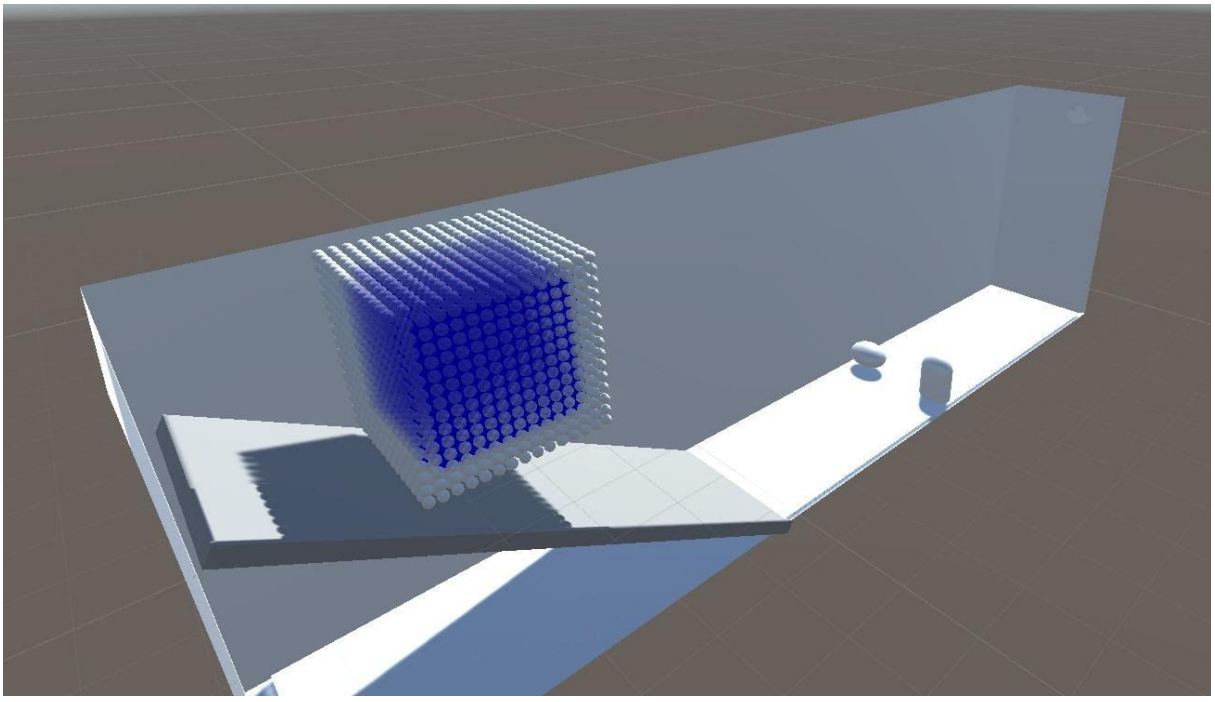

Fig 9: Particles in Perfect Cube Form.

The particles released (4096 particles) and moving downwards through the ramp and complexity of particles is increasing. The particles have approached one another and particle number in a cell is increased. That means more calculation when calculating the weight of a particle and it increases the execution time of the surface recognition.

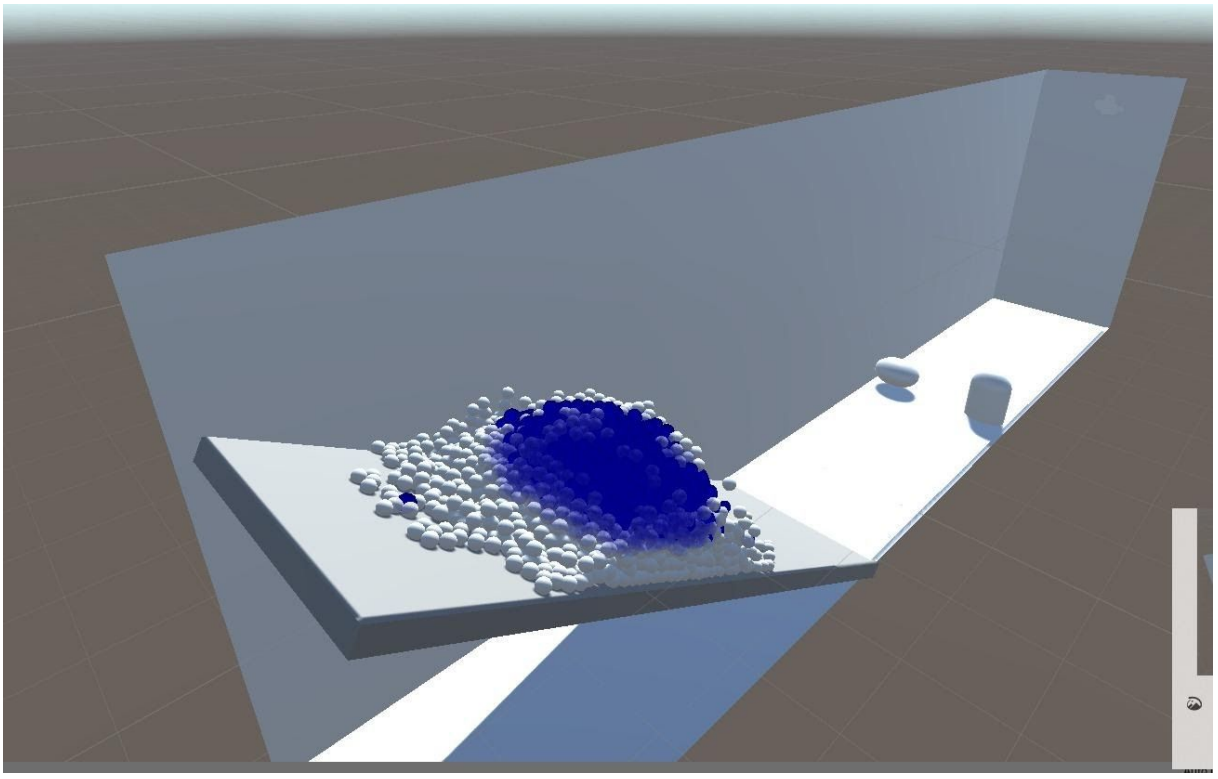

Fig 10: Particles on Ramp.

Particle complexity is decreasing between the ramp and obstacles because particles are just moving forward. When the particles crashed to the obstacles, complexity is increasing. Especially the particles that accumulate in front of the obstacles. These particles are approaching one another but the rest of the particles are going from the middle of the corridor floor.

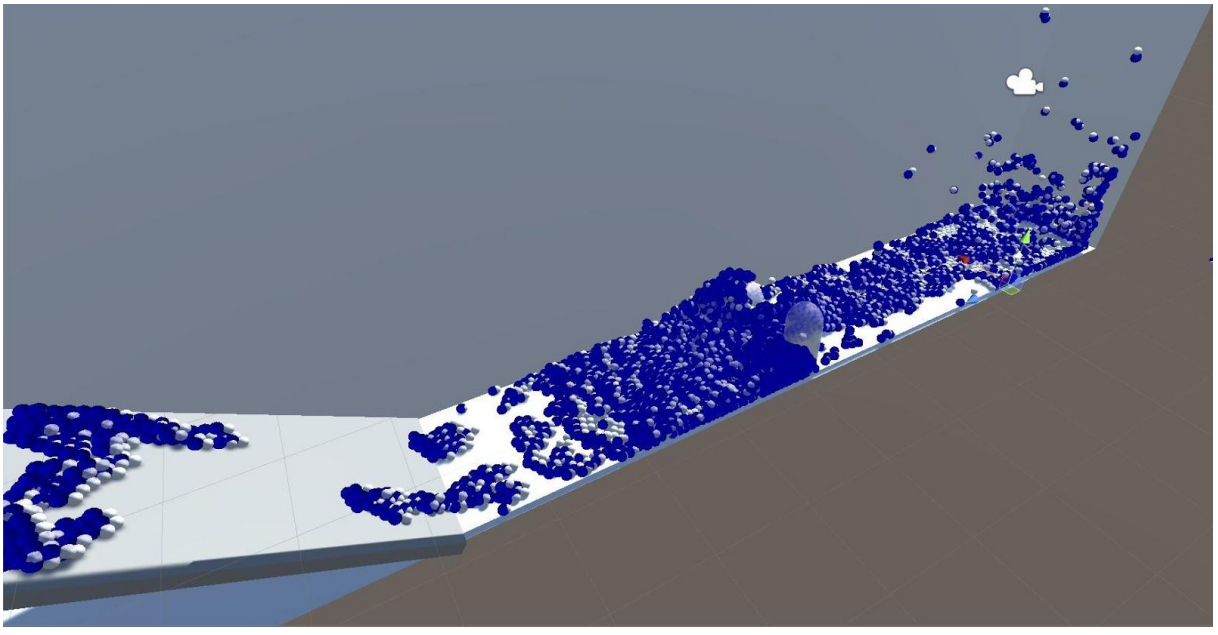

Fig 11: Particles are Crashing to Obstacles.

After the crashing of the particles to the sphere obstacles, particle speed is decreasing, and the other particles position is getting stabilized. That means complexity is decreasing again and surface execution takes less time because of the less computational load.

We deduced the first hundred frame of the scene that we explained, and this is the chart of the surface recognition time per frame chart. We divided a hundred frames of test scene into five phases. As you can see from the chart, the initial execution time is about 700 milliseconds and increasing when moving on the ramp. When the particles pass the ramp and moves through the sphere and capsule obstacles, complexity is decreasing exponentially. At the moment of particles collide with obstacles, particle complexity is increasing, and execution take more time. After the particles pass obstacles in the last phase, particle complexity is decreasing, and surface execution is getting faster.

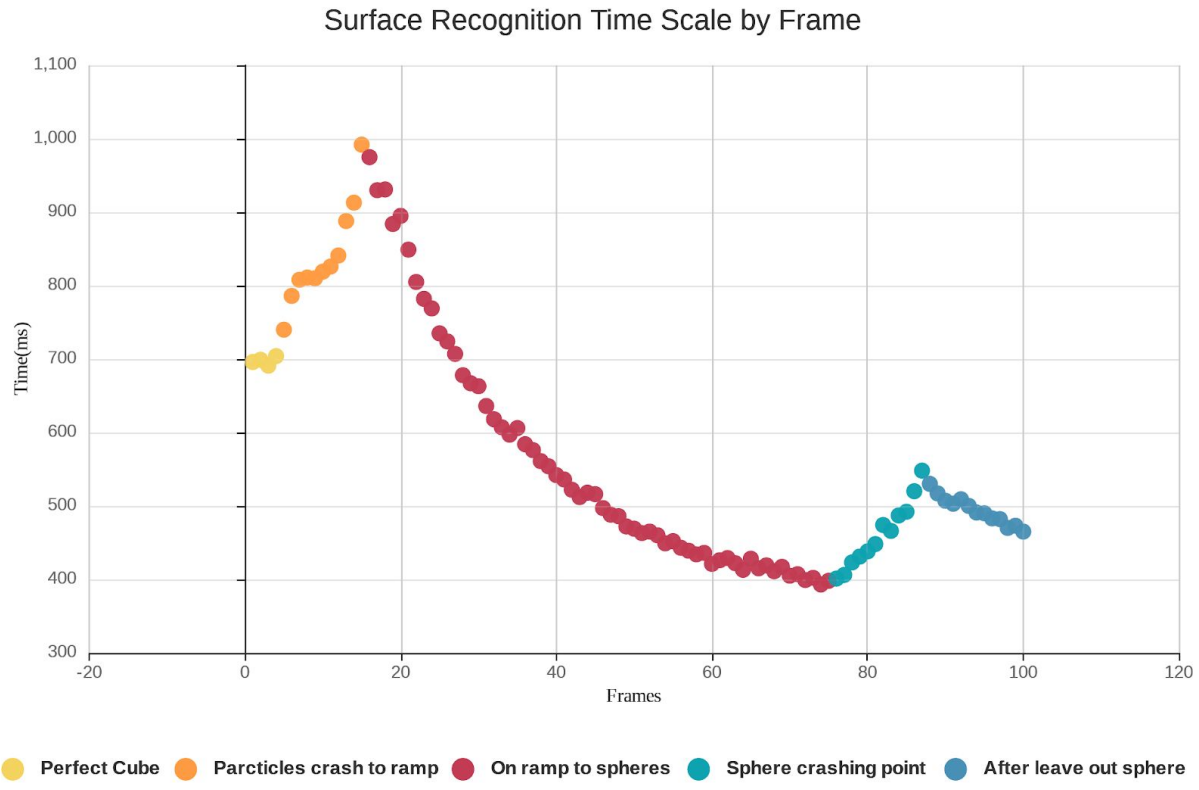

Fig 12: Surface Recognition Time Scale by Frame Chart

We prepared another test measuring scale as particle number. We could not increase the particle numbers more than four thousand due to lack of simulation computer. So, we decided four test cases: One thousand, two thousand, three thousand and four thousand and ninety-six particles.

We decided that measuring the first sixty frames of every particle cases is enough for the testing. We did not change the test scene so we can expect the same results of the previous chart with 4096 particles. The other lines of the different particle numbers are very similar to the yellow line that represents 4096 particles. However, in this test scene we measured 60 frame and that means we did not include particles are colliding to the obstacles phase so that little spike at the right of the chart is not exist compared with the previous chart.

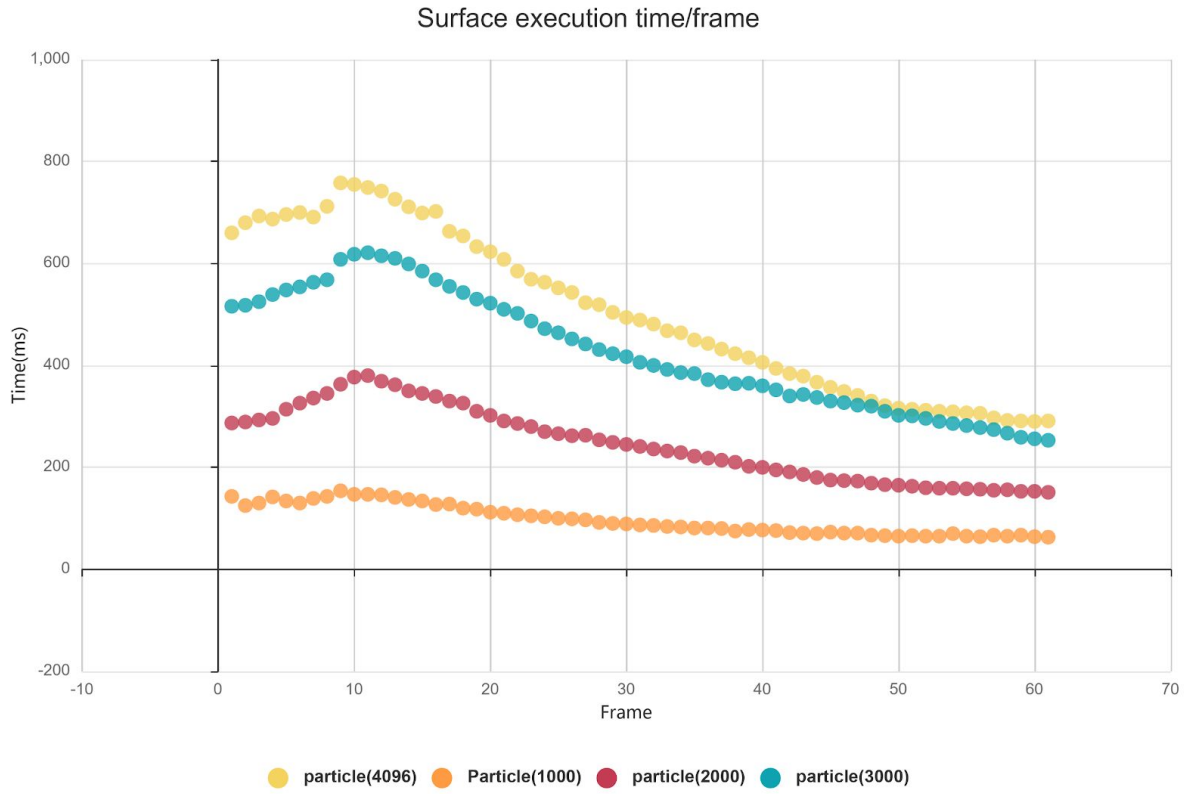

Fig 13: Surface Recognition by particle number comparison chart.

4.3. Environment Testing

In this section, we explained why we decided to use NVIDIA Flex and compared to other particle-based fluid simulations.

4.3.1 Availability of the Necessary Environment

We have tested three particle-based fluid simulation before deciding to use NVIDIA Flex and we disqualified other assets. We stated the reasons for the table below.

Name	**Description**
uFlex	We had small bugs and errors in the code, even though we have fixed minor bugs, the particles were not recognizing the collider of the objects. Collider of the simple primitive objects was not recognized by the Uflex and particles were penetrating the objects. The only plane object was being recognized by the uFlex. The problem could not be solved, and we have changed the fluid simulation.
Obifluid	We eliminated the Obifluid because of performance problems. The expected result was not satisfied by the Obifluid compared to other fluid simulations. Our expectation was reaching 30fps with a hundred thousand particles, but we obtained 3 fps in a scene with three thousand particles.

Screen Space Fluids Pro	We encountered bugs and errors in the code like uFlex. Although we fixed bugs, the performance was very low on higher particle count compared to NVIDIA Flex.

Table 5: Environment Availability.

To give an overview of the test results of the product manual, we showed with images that components of the POF systems operates correctly. Hash System and Surface Recognizer components proved that operating without any problems. In DSD 3.0, we mentioned the working prototype of Marching Cubes code however it will not be implemented into our project. Other test results are included in the product manual revision 1.0 (PM 1.0).

5. CONCLUSIONS

In this section, we summarize our project by explaining the project goals and how we achieved it. We give cost analysis of workers, software, and hardware. We described the benefits of the POF system and fields of usage. Lastly, we reviewed future works as further developments on the project.

5.1. Summary

We stated our requirements in RSD 3.0 [1]. Generally, our requirements were increasing performance and efficiency. Functional requirements except for the last part which is the implementation of Marching Cubes algorithm. The excuse is explained in the warning title page 8. The non-functional requirements are not completely met because we could not finish the project. If we could have finished the last step. There would be tangible proof that all requirements fully met. The marching cubes algorithm would vastly decrease the workload on a GPU. Because drawing and rendering millions of sphere particles instead of drawing triangles to covering the surface vertices take so much time.

5.2. Cost Analysis

In this section we separated project schedule as first and second semester.

5.2.1 Cost Analysis of First Semester

First-semester cost analysis involves the cost of workers, software, and hardware. Besides, we shared the ideal hardware components of the simulation computer after the table of the system that we used.

5.2.1.1 Cost of Workers

Members	Day/Hour	Week/Hour	Semester/Hour	Salary/Hour	Salary/Monthly	TOTAL
Member	8	40	560	30 TL	4800 TL	16800 TL

Table 6: 1. Semester Cost of Workers.

As shown in the cost analysis table, three people works in the project. Every people work equally as workload. Therefore, only one member is represented on the cost table.
Every member works 8 hours a day and 5 days a week. A semester consists of 14 weeks and salary is 30 Turkish Lira per hour. Each member costs 4800 TL per month and costs 16800 in a semester. The salary costs of all three members are 50400 TL per semester. The equivalent of 16800 TL is $2894, 67. Currency translation has made from Dollar / Turkish Lira = 1 / 5.80 on 10 December 2019.

5.2.1.2 Cost of Software

Title of Software	Cost
uFlex	$30
Obi Fluid	$30
SSF	$7
Total Cost	**$67**

Table 7: Cost of software

5.2.1.3 Cost of Hardware

5.2.1.3.1 PC components that used in Project

Total cost = Total employee cost + Total software cost + Total Hardware cost (PC1)

PC 1 components that used in Project	Description
Operating System	Windows 10 (64-bit)
Processor	Intel Core i7-4700 HQ CPU
Memory	16 GB RAM – DDR3L-1600 MHz
GPU	NVIDIA GeForce GTX850M 4GB DDR3
Cost of PC 1 per user	$1693, 21
Total cost (for 1 worker)	**$4684, 88**
Total cost (for 3 workers)	**$14054, 64**

Table 8: Cost of PC1

5.2.1.3.2 Optimal Simulation Computer (PC 2)

Total cost = Total employee cost + Total software cost + Total Hardware cost (PC2)

Optimal Simulation Computer (PC 2)	**Description**
Operating System	Windows 10(64-bit Pro)
Processor	8-core Intel i7 5.1 GHz
Memory	32 GB RAM- DDR4- 2666MHz
GPU	NVIDIA Quadro P2200 5GB
Cost of PC 2 per user:	$5017
Total cost (for 1 worker)	**$8008,67**
Total cost (for 3 workers)	**$24026,01**

Table 9: Component costs of PC2

5.2.2 Cost Analysis of Second Semester

In this section cost of workers and hardware tables are given. There is no cost of software and hardware table because we did not spend money on software and hardware on the contrary of the first semester.

5.2.2.1 Cost of workers

In this semester, we could not continue to project code implementations since the university is closed in the middle of March. We spend our working time by completing document work and preparing other materials. Therefore, the cost of workers in April and May is lower than general because we spent fewer hours for obvious reasons. Calculations are made for each month and each member.

Month	**Day/Hour**	**Workday/Month**	**Month/Hour**	**Salary/Hour**	**Salary/Monthly**
February	8	20	160	30 TL	4.800 TL
March	8	22	176	30 TL	5.280 TL
April	4	21	84	30 TL	2.520 TL
May	4	14	56	30 TL	1.680 TL
				Total Salary (for 1 member) = 14.280 TL	
				Total Salary = 42.840 TL	

Table 10: 2. Semester Cost of Workers

5.3. Benefits of the Project

In this section, we listed the areas that we can think about. It could be more areas that our project can benefit.

5.3.1 ***Animations and Movies:*** The POF system can be used in any movies, animations that used fluids.

5.3.2 ***Scientific work:*** Our project benefits scientific areas the most because the project is heavily research and development based on the research papers about the particle-based fluid simulations. Scientist and researchers can use the POF system for their scientific research.

5.3.3 ***Games:*** Games can necessitate a fluid simulation system to make more realistic games. The POF system can be used to obtain more realistic games. For instance, a sailing simulator game is a viable option for our system.

5.3.4 ***Construction:*** The construction and Architecture areas can benefit from our system because the simulation is physics-based which means the POF system is almost realistic. The POF system neglects some imperceptible elastic deformations. For instance, a civil engineer can build a barrage and want to test endurance, on the computer simulation. Therefore, our system can be used for construction and architecture testing.

5.4. Future Work

The first thing to do is implementing the Marching Cubes algorithm and later research algorithms to make the visual output more fluid-like. More research papers can be implemented to the POF system and discuss the results for future work.

References

1. Requirement Specifications Document revision 3.0 (RSD 3.0).

2. Design Specifications Document revision 3.0 (DSD 3.0)**.**

3. Product Manual revision 1.0 (PM 1.0).

4. **[WH87]** William E. Lorensen and Harvey E. Cline. (1987). Marching cubes: A high resolution 3D surface construction algorithm. ACM SIGGRAPH Computer Graphics. 21, 163-169.

5. **[ZB05]** Zhu, Y., & Bridson, R. (2005). Animating sand as a fluid. (New York, NY, USA, 2005) *ACM Trans. Graph., 24*, 965-972.

6. **[KBSS01]** Kobbelt, Leif & Botsch, Mario & Schwanecke, Ulrich & Seidel, Hans-Peter. (2001). Feature sensitive surface extraction from volume data. Proceedings of the ACM SIGGRAPH Conference on Computer Graphics. 10.1145/383259.383265.

7. **[AIAT81]** G. Akinci, M. Ihmsen, N. Akinci, and M. Teschner. 2012. Parallel Surface Reconstruction for Particle-Based Fluids. Comput. Graph. Forum 31, 6 (September 2012), 1797–1809. DOI:https://doi.org/10.1111/j.1467-8659.2012.02096.x

APPENDICES

APPENDIX A: REQUIREMENTS SPECIFICATIONS DOCUMENT