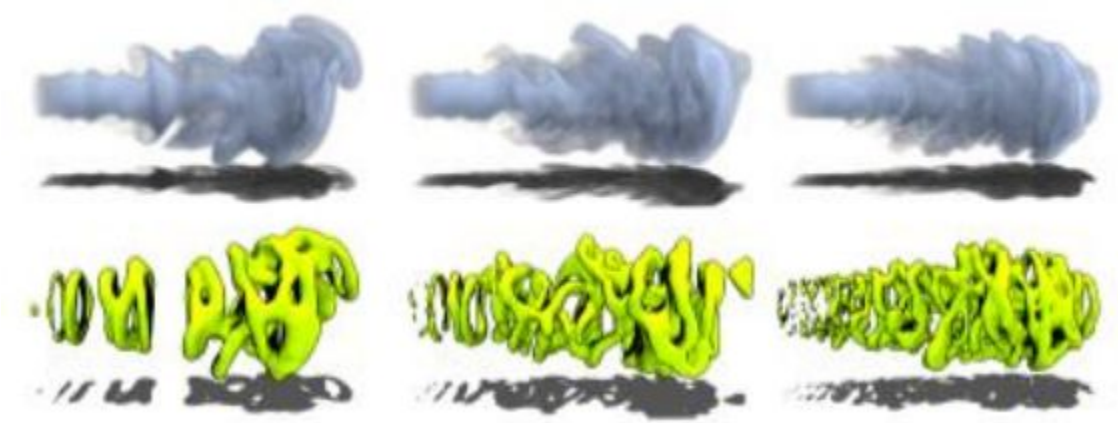
Parallel computing technologies and rendering optimization in the problem of fluid simulation by the example of the Incompressible Schrodinger Flow method.

Incompressible Schrödinger Flow (ISF)

- Two-component wave function of complex variables changing with time according to the Schrodinger equation
- Euler method - a fluid is represented as an area that is divided into cells, which forms a grid



Navier-Stokes equations

$$\frac{\partial \vec{v}}{\partial t} = -(\vec{v} \cdot \nabla)\vec{v} + \nu \Delta \vec{v} - \frac{1}{\rho}\nabla \rho + \vec{f} \tag{1}$$

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{v}) = 0 \tag{2}$$

$$\nabla \cdot \vec{v} = 0 \tag{3}$$

$$\vec{v}|_{\partial \Omega} = 0$$
$$\vec{v}|_{t=0} = \vec{v}_0 \tag{4}$$

Incompressible Schrödinger Flow (ISF)

$$\psi = (\psi_1, \psi_2)^T : M \to \mathbb{C}^2 \tag{1}$$

$$\rho = |\psi|^2 = \langle \psi, \psi \rangle_{\mathbb{R}}$$

$$\rho v_\alpha = \hbar \langle \frac{\partial \psi}{\partial x_\alpha}, i\psi \rangle_{\mathbb{R}} \quad \alpha = 1,2,3 \tag{2}$$

$$i\hbar\dot{\psi} = -\frac{\hbar^2}{2}\Delta\psi + p\psi, \qquad \frac{\partial \psi}{\partial N}|_{\partial M} = 0 \tag{3}$$

$$\langle \Delta\psi, i\psi \rangle_R = 0 \tag{4}$$

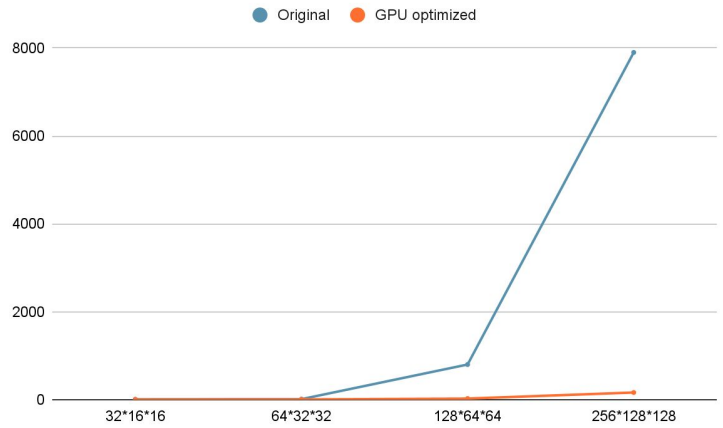
$$|\psi|^2 = 1$$

$$div(v) = 0 \tag{5}$$

$$\rho = 1$$

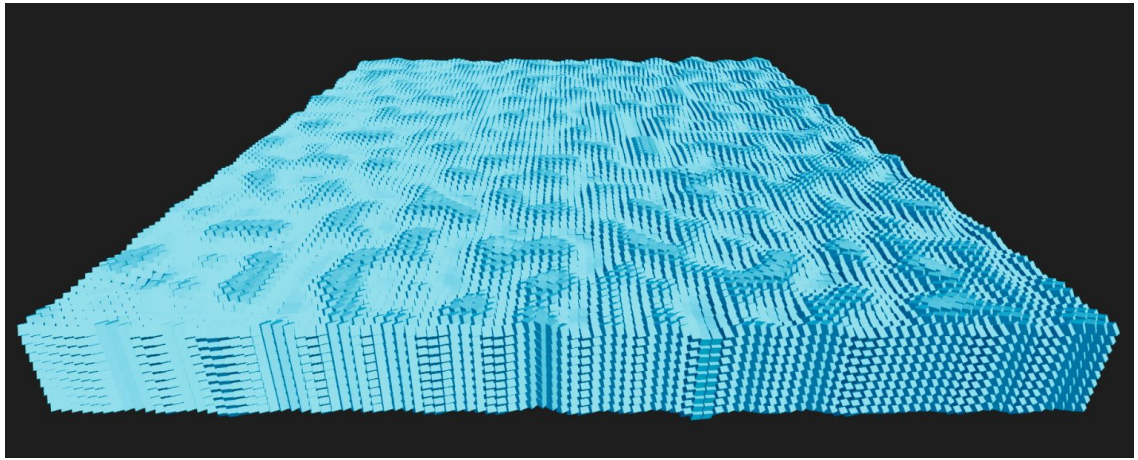
GPGPU (CUDA)

The managedCuda library was used, as it has support for the c# language, the ability to write your own CUDA kernels, it has a built-in implementation of FFT3D and it is actively supported.



Rendering optimization

In order to apply the best methods and technologies, their consideration was previously given outside the problem of fluid modeling. We took a set of a large number of elements moving through Perlin noise and considered solutions for the most productive rendering.



Rendering methods

1. [Unity Game Objects] The first basic method is based on using standard Unity objects. It has the widest possibilities, because it does not contain restrictions in the possibilities of interaction with objects. But it has very low performance, due to the large amount of unnecessary information and functionality contained in the Unity Game Object. Suitable for game development, but not for simulations with a large number of objects.

2. [Avoid external calls] This point is more a useful practice in programming large systems than a separate method. Its essence is to minimize external calls, in this case, instead of getting the position of the object, in each step we save the resulting position to an array and use it in the next step. In general, it is a very useful practice and should be used wherever possible.

3. [Particle System] The third method is to use the Particle System built into Unity. In it we can set the number of particles, their position, speed, material and some other parameters. Using meshes (three-dimensional grid) as an element (particle), it does not show the best results. But it has very good performance when using billboard display (always an image directed to the screen) ([Particle System Billboard] 3.1 in the table). But, accordingly, it has many limitations, such as the inability to set the rotation, the direction of the particle, etc. For many tasks, where displaying the positions of particles will be enough is one of the best ways in terms of the speed of execution and the labor spent.

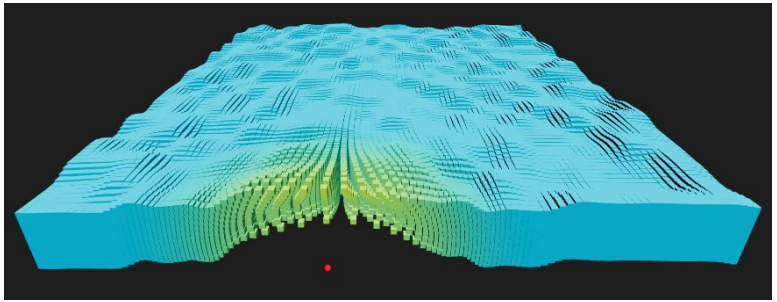
4. [GPU Instancing] This method uses GPU Instancing technology. GPU instantiation is a method of optimizing rendering calls that displays multiple copies of a grid with the same material in a single rendering call, where each copy of the grid is called an instance. This is useful for drawing objects that appear in the scene multiple times. Creating a GPU instance displays identical meshes in the same rendering call. To add variability and reduce the appearance of repetition, each instance may have different properties, such as color or scale. The following described methods also use this technology in combination with other features.

Unity DOTS

1. [Job + Burst] New approach called Unity DOTS - Data-Oriented Technology Stack. DOTS is a combination of technologies and packages that provides a data-centric approach to development in Unity. Applying data-centric design to the project architecture allows you to scale processing with high performance. At this point, technologies such as the C# Job System and Burst Compiler are used. Job System allows you to write simple and secure multithreaded code so that the application can use all available processor cores to execute code. Burst is a compiler that can be used with Unity Job System to create code that improves the performance of your application. It translates the code from the IL/ byte code.NET into optimized native processor code using the LLVM compiler.
2. [ECS + Job + Burst] This item adds to the previous use of the ECS - Entity Component System. It is a data-oriented framework compatible with objects in Unity. This is the best approach without using GPU computing, it allows you to get maximum performance while maintaining full control over objects, for example by adding the possibility of their interaction. Uses the advantages of the CPU, such as, for example, the size of the cache memory. Entities Graphics is used for rendering. Entities Graphics provides systems and components for rendering ECS entities. Entities Graphics is not a render pipeline: it is a system that collects the data necessary for rendering ECS entities, and sends this data to Unity's existing rendering architecture.

Compute Shaders

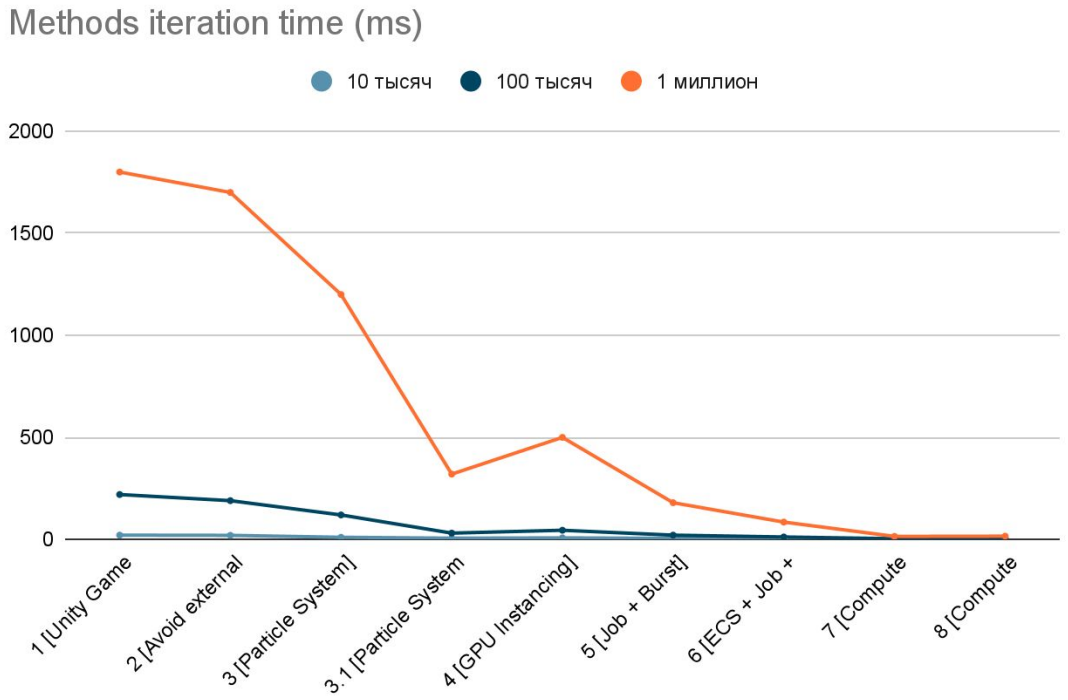
1. [Compute Shaders] This point consists in using shaders written in the HLSL language to calculate positions directly on the video card and then display them. This way we avoid transferring a large amount of data every frame (we only send their initial location and the necessary data at the start) to the video card and use the huge computing capabilities of the GPU.
2. [Compute Shaders with Interaction] This item adds the use of Compute Shaders technology, i.e. computational shaders. Computational shaders are shader programs that run on the GPU outside of the normal rendering pipeline. They can be used for massively parallel GPGPU algorithms, as well as for some rendering stages. The system requires support for computational shaders, which satisfies most modern devices, including mobile ones.



Rendering time comparison

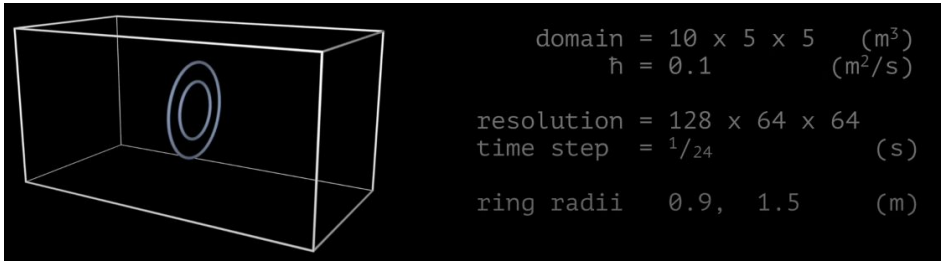
	10 тысяч	100 тысяч	1 миллион
1 [Unity Game Objects]	21	220	1800
2 [Avoid external calls]	20	190	1700
3 [Particle System]	10,5	120	1200
3.1 [Particle System Billboard]	6,2	31	320
4 [GPU Instancing]	7,8	45	500
5 [Job + Burst]	5,2	21	180
6 [ECS + Job + Burst]	4,7	12	85
7 [Compute Shaders]	3,5	3,6	15
8 [Compute Shaders with Interaction]	3,6	3,8	16

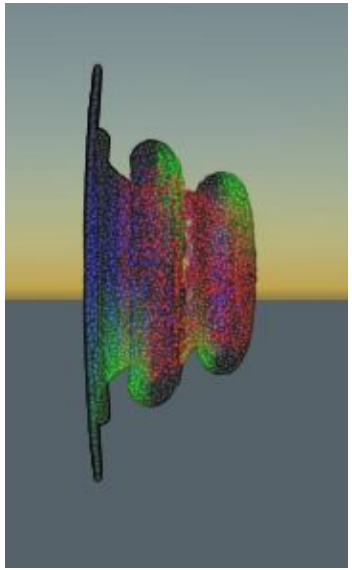
Rendering time comparison

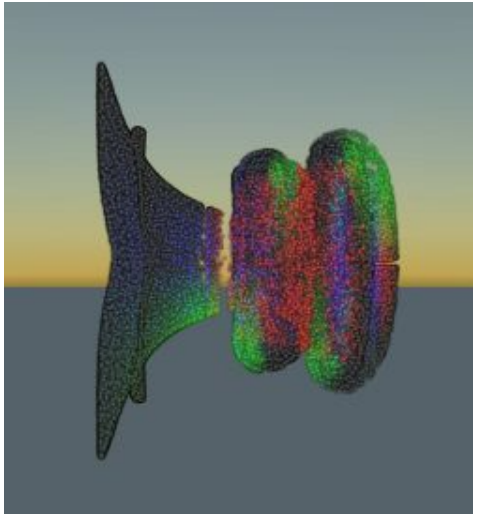


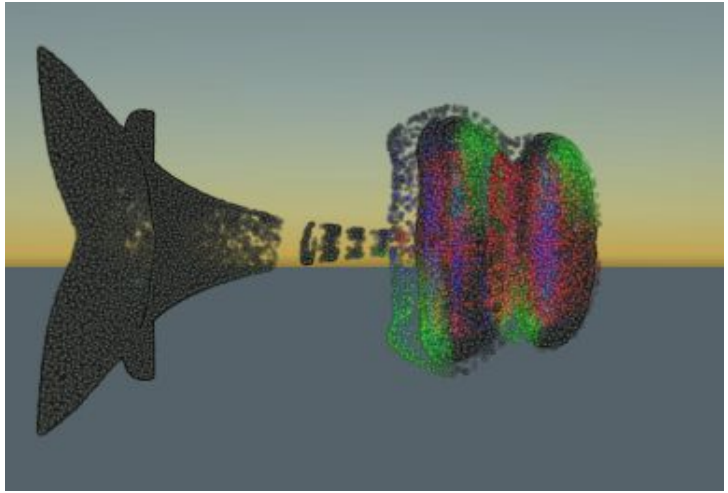
Simulation rendering

Leapfrogging vortex rings



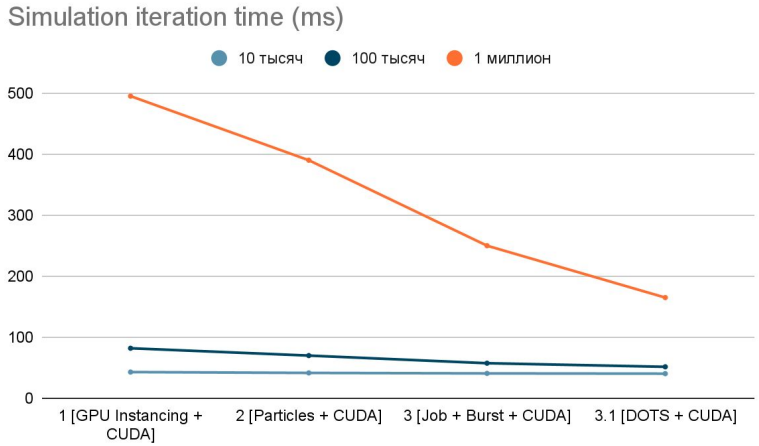




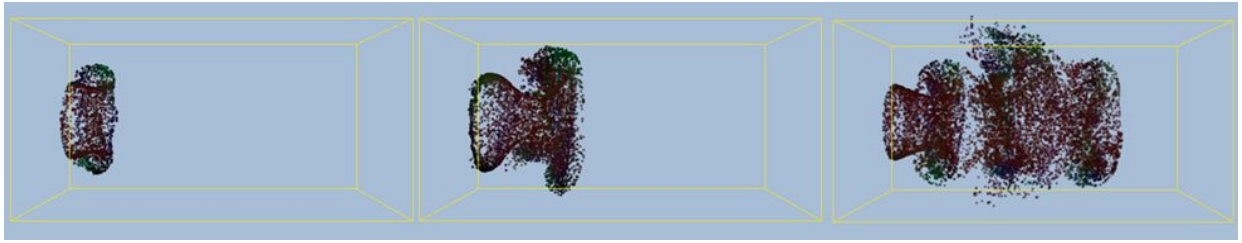


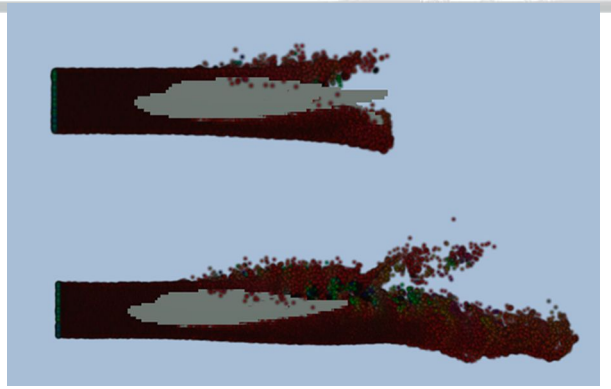
Simulation rendering time comparison

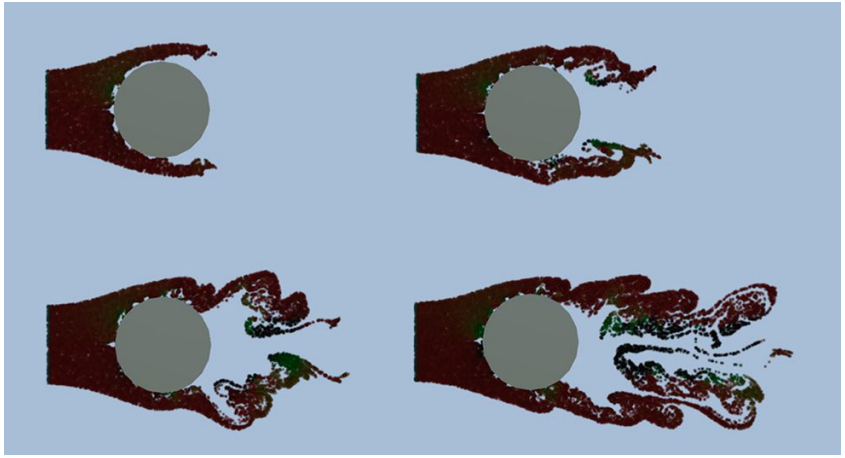
	10 тысяч	100 тысяч	1 миллион
1 [GPU Instancing + CUDA]	43	82	495
2 [Particles + CUDA]	41,6	70	390
3 [Job + Burst + CUDA]	40,8	57,5	250
3.1 [DOTS + CUDA]	40,4	51,6	165

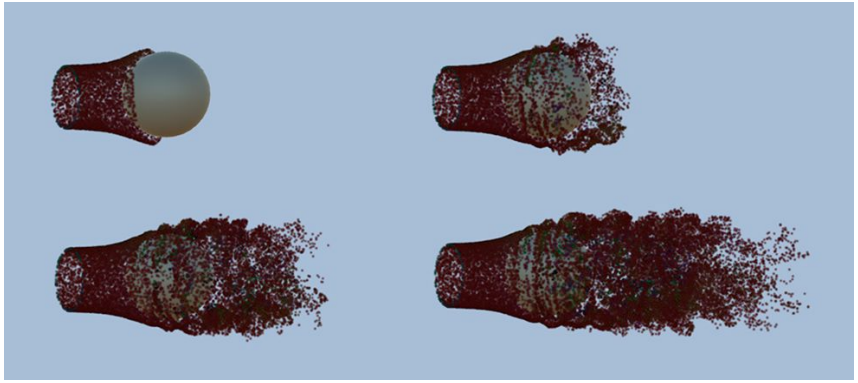


Simulation examples









Conclusions and further direction of work

- Consider optimized versions of DOTS from other developers, as their tests show big performance gains
- Implement a simulation task using compute shaders by storing data in a GPU buffer and using that buffer in a shader for rendering
- Minimize the amount of data transferred by passing only the position of the object, instead of all its points

Thank you for your attention!