Instituto Tecnológico y de Estudios
Superiores de Monterrey

Programming Languages

GPU Flocking in Unity

Eric Buitrón López
A01704340

Index

Introduction

This project is a comparison of an unoptimized flocking AI that is run in the CPU vs a flocking AI that is run in the GPU via a compute shader in Unity. The example scene models an ocean with several flocks of fish. However, the same scripts could be applied to any other animals that have this behavior.

Context

Flocking Behavior

What is a flock?

Figure 1. Flock of fish

Flocks are a group of animals that travel together with a particular behavior.

Flocking behavior 101

In a flock, every animal has to react differently depending on the behavior of their neighbors which are considered from a certain distance and angle.

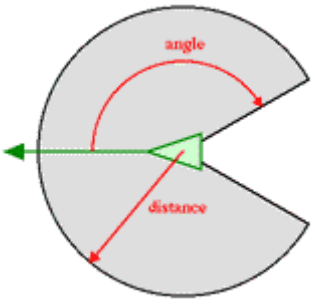


Figure 2. Neighborhood of a flock

That behavior is defined by the avoidance, alignment and cohesion of every animal in the flock.

Avoidance

Every animal in the flock needs to have an average separation distance between its neighbors.

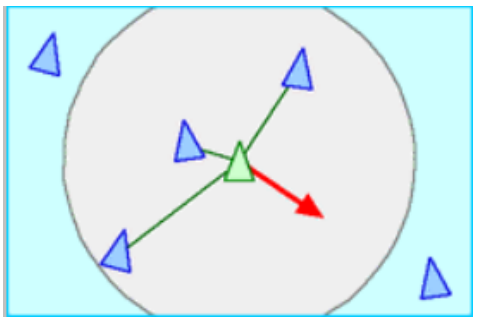


Figure 3. Avoidance

Alignment

Every animal in the flock needs to steer towards the average angle at which their neighbor is moving.

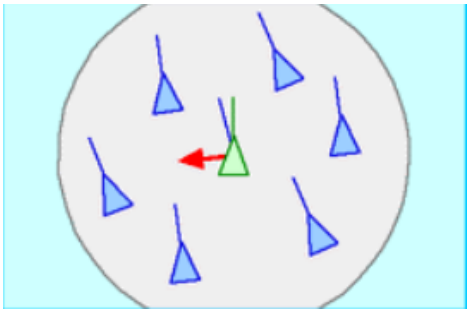


Figure 4. Alignment

Cohesion

Every animal in the flock needs to move towards the average position of their neighbors.

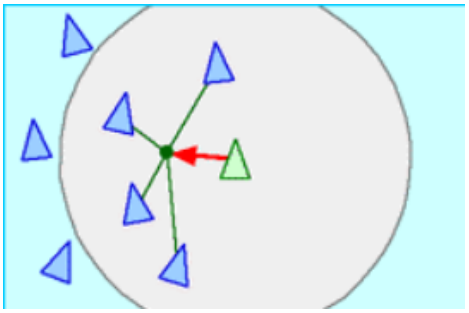


Figure 5. Cohesion

The flocks in the simulation also move within a certain bound.

Compute Shaders

Compute shaders are programs that run on the GPU that can be used for massively parallel General-Purpose Graphics Processing Unit (GPGPU) algorithms, or to accelerate parts of game rendering.

Differences and similarities with CUDA

Main kernel function

```
__global__ void add(int *a, int *b, int *c, int max){
    int index = threadIdx.x + blockIdx.x*blockDim.x;
    int id = index;
    while (id < max){
        c[id] = a[id] + b[id];
        // for problems larger than number of threads increase
        // the index is increased by the total number of threads
        id =  id + blockDim.x* gridDim.x;
    }
}
```

Figure 6. Main function in CUDA

```
[numthreads(1,1,1)]
void FillWithRed (uint3 dtid : SV_DispatchThreadID)
{
    res[dtid.xy] = float4(1,0,0,1);
}
```

Figure 7. Main function in a Compute Shader

Calling the main function

```
cudaMemcpy(d_a, a, tam, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, tam, cudaMemcpyHostToDevice);


add << <N*N / THREADS_PER_BLOCK,  THREADS_PER_BLOCK >> >(d_a, d_b, d_c, N*N);

cudaMemcpy(c, d_c, tam, cudaMemcpyDeviceToHost);
```

Figure 8. Calling main function in CUDA

```
    //SEND DATA TO COMPUTE SHADER
    computeKernelIndex = flockComputeShader.FindKernel("CSMain");
    flockComputeShader.SetBuffer(computeKernelIndex, "_flockUnitBuffer", flockUnitBuffer);
    flockComputeShader.SetInt("_flockSize", flockSize);
    flockComputeShader.SetFloat("_cohesionDistance", cohesionDistance);
    flockComputeShader.SetFloat("_FOVAngle", FOVAngle);
    flockComputeShader.SetFloat("_cohesionWeight", cohesionWeight);
    flockComputeShader.SetFloat("_smoothDamp", smoothDamp);
    flockComputeShader.SetFloat("_minSpeed", minSpeed);
    flockComputeShader.SetFloat("_maxSpeed", maxSpeed);
    flockComputeShader.SetFloat("_avoidanceDistance", avoidanceDistance);
    flockComputeShader.SetFloat("_avoidanceWeight", avoidanceWeight);
    flockComputeShader.SetFloat("_alignmentDistance", alignmentDistance);
    flockComputeShader.SetFloat("_alignmentWeight", alignmentWeight);
    flockComputeShader.SetFloat("_boundsDistance", boundsDistance);
    flockComputeShader.SetFloat("_boundsWeight", boundsWeight);

    //SEND DATA TO MATERIAL
    flockUnityMaterial.SetBuffer("_flockUnitBuffer", flockUnitBuffer);
}

Unity Message | 0 references
private void Update()
{
    flockComputeShader.SetFloat("_Time", Time.deltaTime);

    //dispatch kernel
    int groups = Mathf.CeilToInt(flockSize / THREAD_GROUPS);
    flockComputeShader.Dispatch(computeKernelIndex, groups > 0 ? groups : 1, 1, 1);
```

Figure 9. Calling main function in a Compute Shader

Solution

For testing purposes, the simulation was built using a fish as the animal for the flocks.

CPU solution

Computing all the information needed for the simulation is an $O(n^2)$ problem since the CPU needs to calculate the parameters of every fish and every fish needs to check every other fish to check if it's their neighbor. The scripts used are based on the "Flocking AI Algorithm in Unity Tutorial" by GameDevChef.

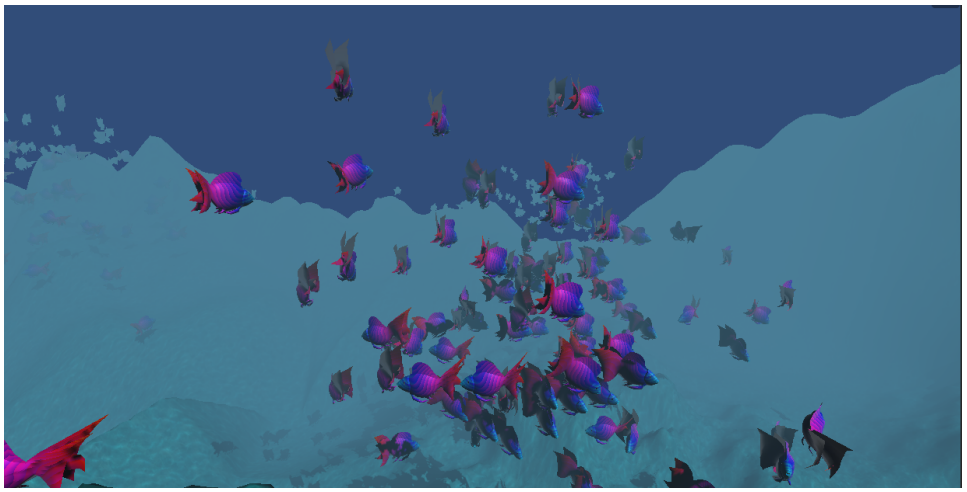


Figure 10. Flocks of fish in the CPU simulation

GPU solution

With the use of compute shaders, each thread in the GPU can calculate the parameters of each fish. Therefore, the complexity of the problem becomes O(n) since each thread only needs to check for the neighbors.

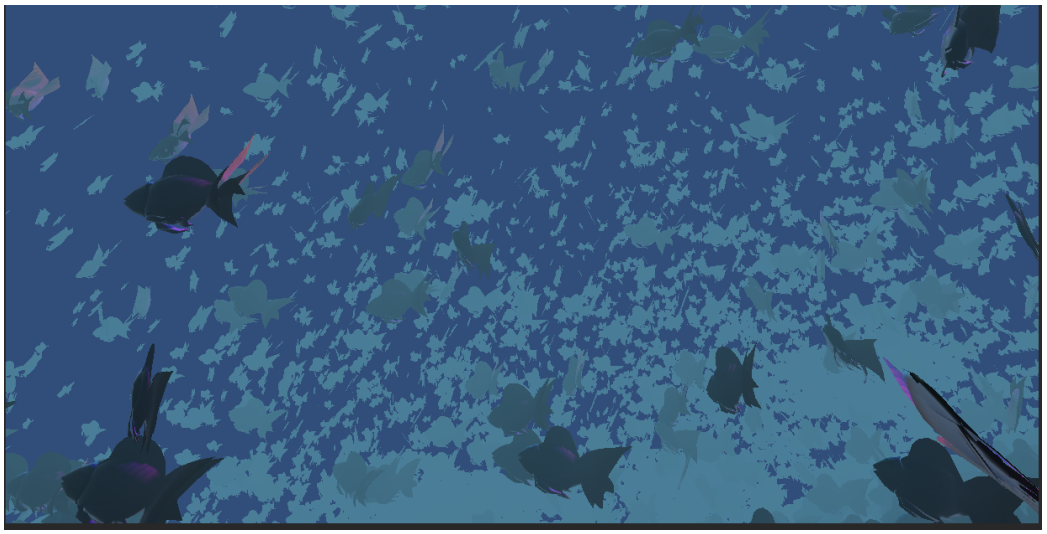


Figure 11. Flocks of fish in the GPU simulation

Results

The simulation was tested in a laptop with an Intel Core i7-7700HQ CPU and an NVIDIA GeForce GTX 1060. A total of 4 tests were performed in order to properly compare the performance between running the simulation in the CPU vs running it in the GPU. Each test increased the amount of fish in the flock by one order of magnitude, going from a flock size of 1 up to 1,000. The scene had a total of 10 flocks which means that the amount of fish units in the scene went from 10 up to 10,000. The benchmark being compared in the tests is the Frames Per Second (FPS) at which the simulation runs.

flock size	# of units	CPU	GPU
1	10	190.3125	206.9231
10	100	163.9375	199.9524
100	1000	16.91667	176.0526
1000	10000	0	190.3158

Table 1. Average FPS in each test.

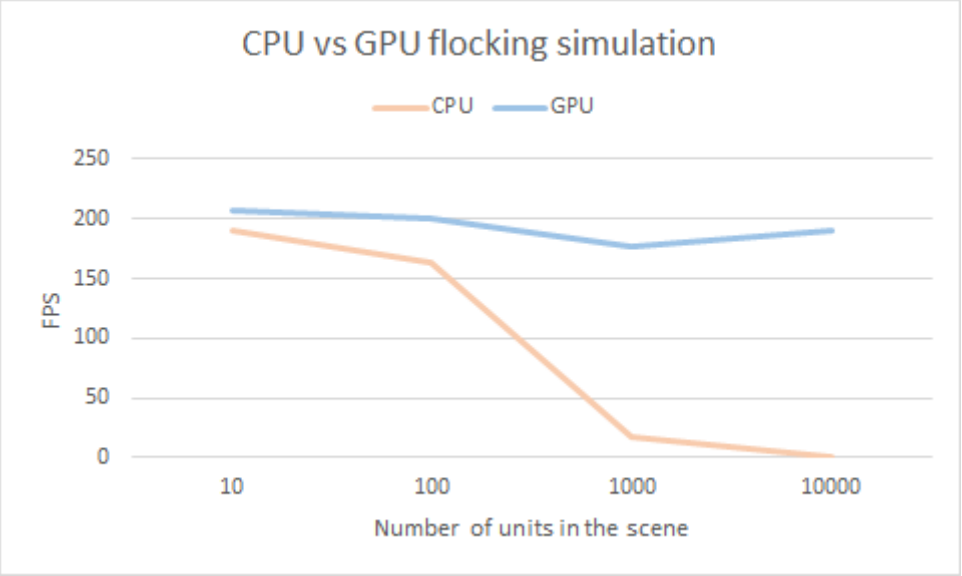


Figure 12. CPU vs GPU flocking simulation line chart.

The following video demonstrates the simulation running using the different test parameters: https://youtu.be/dORFwoVXMsU

Conclusions

The results from the tests are very clear, running the simulation in the GPU drastically improves its performance. This means that using this type of simulation for games could be achievable without sacrificing its performance and the experience of the player. However, it is important to take into account that in order to run compute shaders in Unity, the graphics card of the user should have support for DirectX 11, Metal graphics, Vulkan or modern OpenGL platforms.

Further Improvements

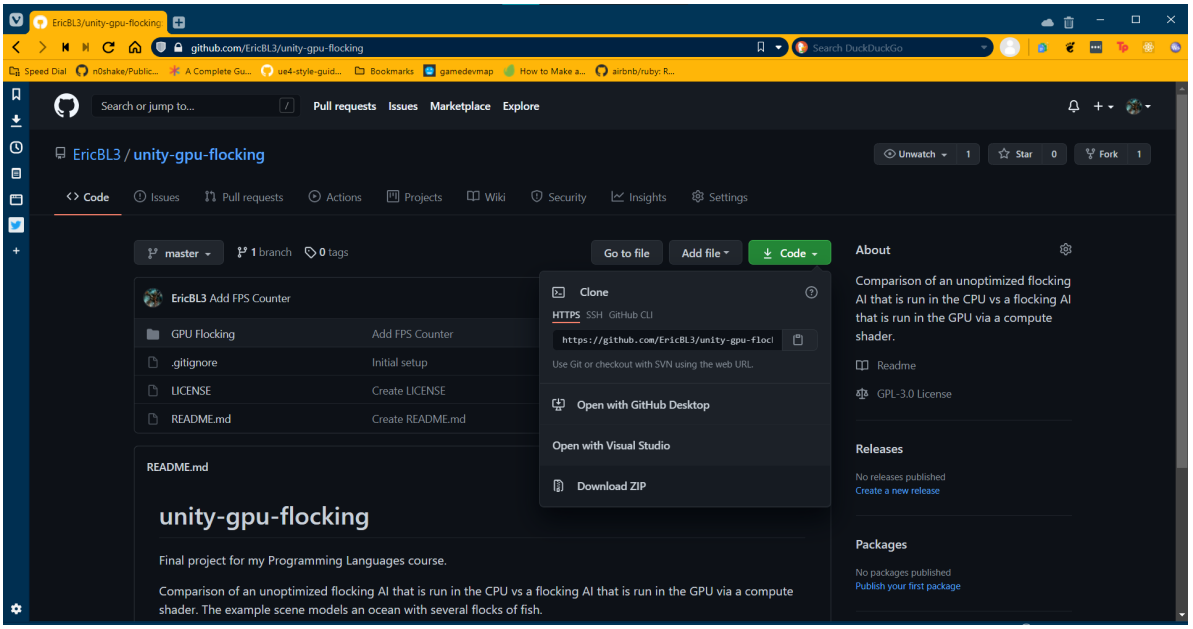
The implementation of the solution is far from perfect. There are still several areas of improvement that were not done for this project due to a lack of knowledge in advanced computer graphics topics such as quaternions and using/writing normal shaders in Unity. The main areas of improvement are:

- Fixing the rotation of the fish, which has to be done manually in the shader.
- Fixing the “duplication” of fish when the flock size is bigger than 250.
- Fixing the increment of speed when more than one flock is active in the scene.

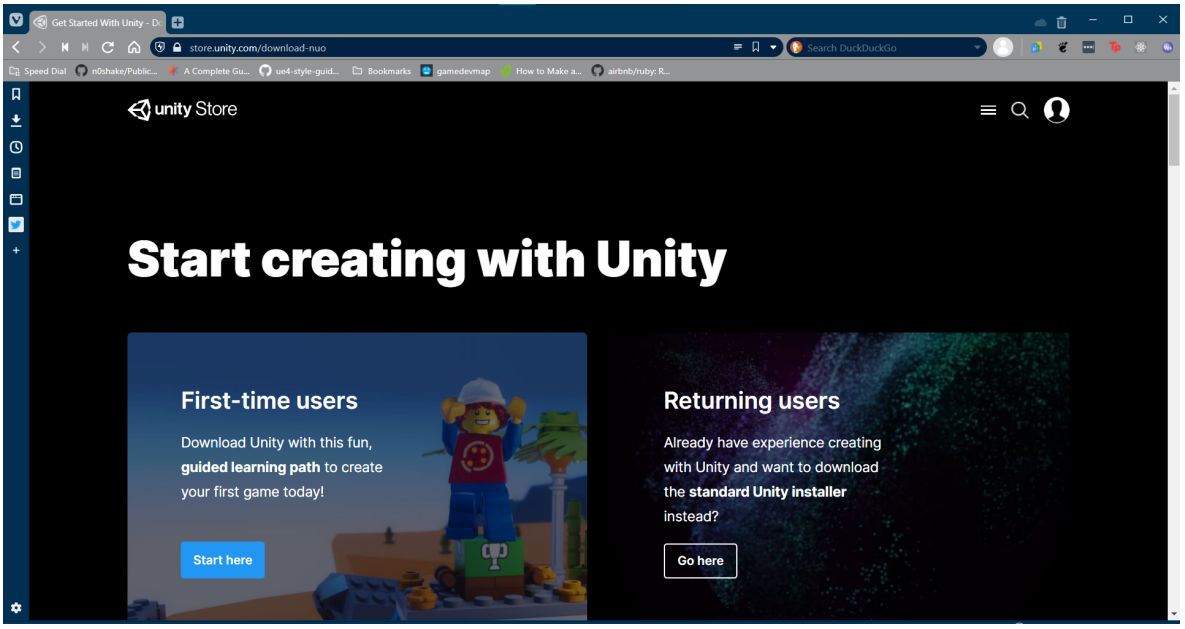
The following video demonstrates these errors during the simulation: https://youtu.be/TQxRf1WOiig

Setup Instructions

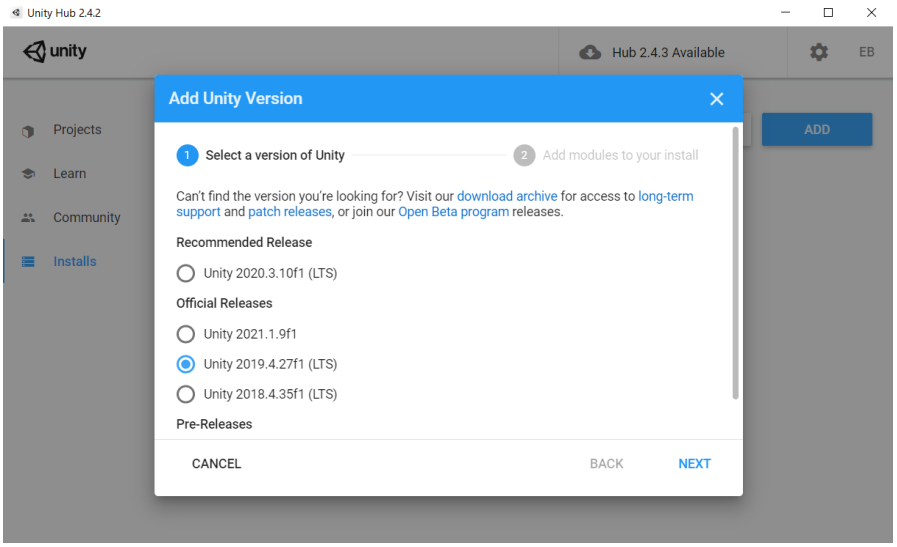
1. Download and unzip the source code from the github repository.



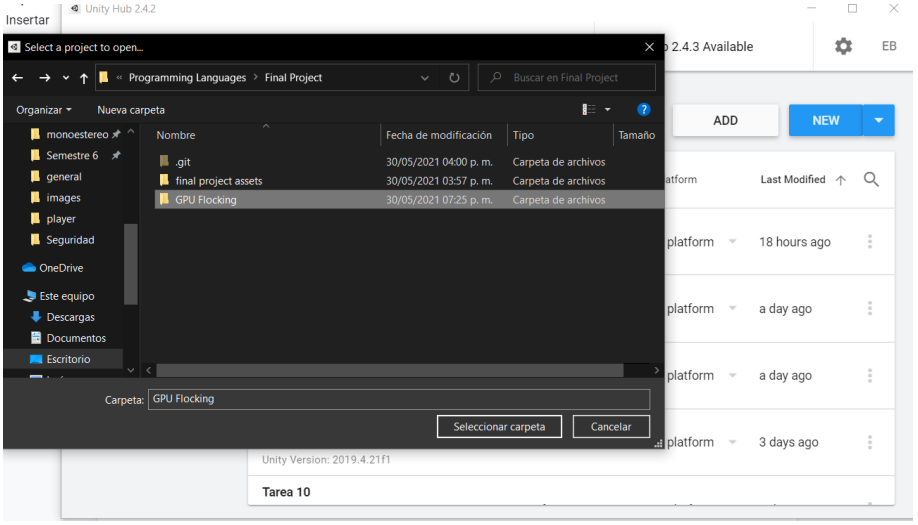
2. Download and install the Unity Hub



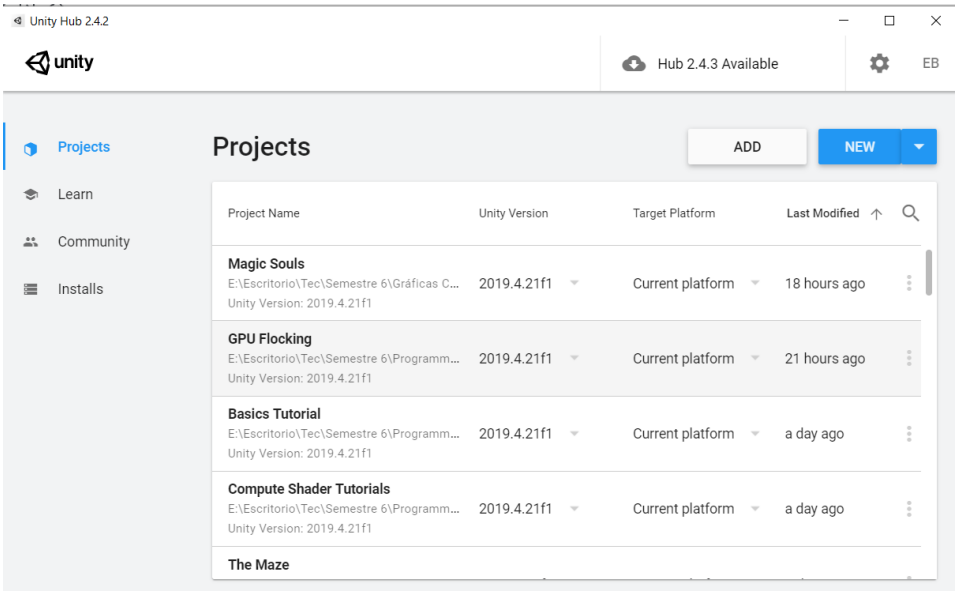
3. Install the latest 2019.4 version of Unity



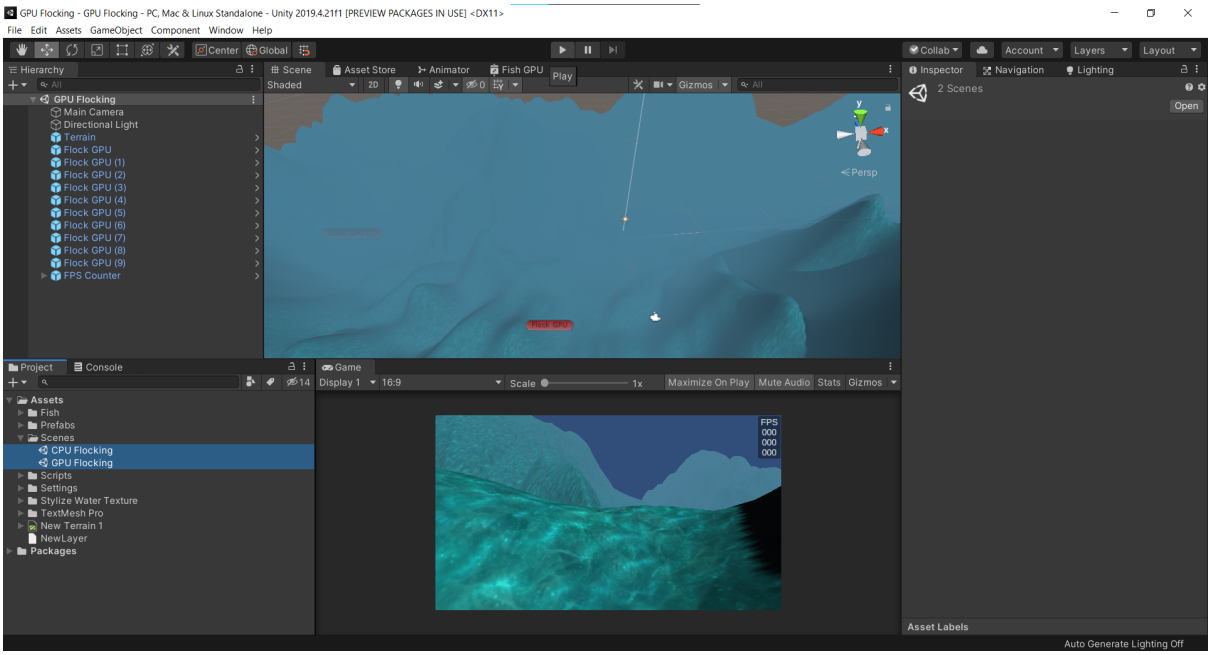
4. Click on Add project and select the GPU Flocking folder



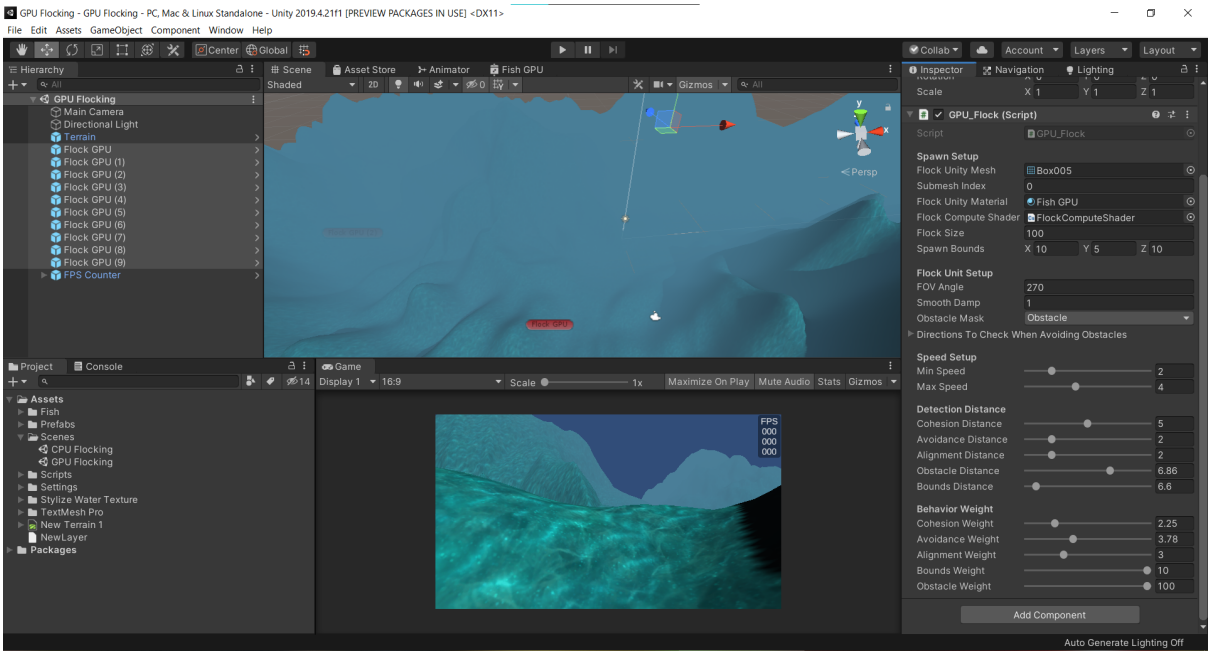
5. Click on the GPU Flocking project. **Note: make sure that the unity version is 2019.4.x**



6. Select the scene that you want to run (GPU or CPU) and click Play.



7. To change the parameters of the simulation, select all the flocks in the Hierarchy and modify the values of the script in the Inspector.



References

1. Docs.unity3d.com. n.d. *Unity - Manual: Compute shaders*. [online] Available at: https://docs.unity3d.com/2019.4/Documentation/Manual/class-ComputeShader.html
2. Reynolds, C., 2001. *Boids (Flocks, Herds, and Schools: a Distributed BehavioralModel)*. [online] Red3d.com. Available at: http://www.red3d.com/cwr/boids/