An Approach to Parallel Processing with Unity*

Jeremy Servoz

Integrated Computer Solutions Inc.

Table of Contents

Table of figures

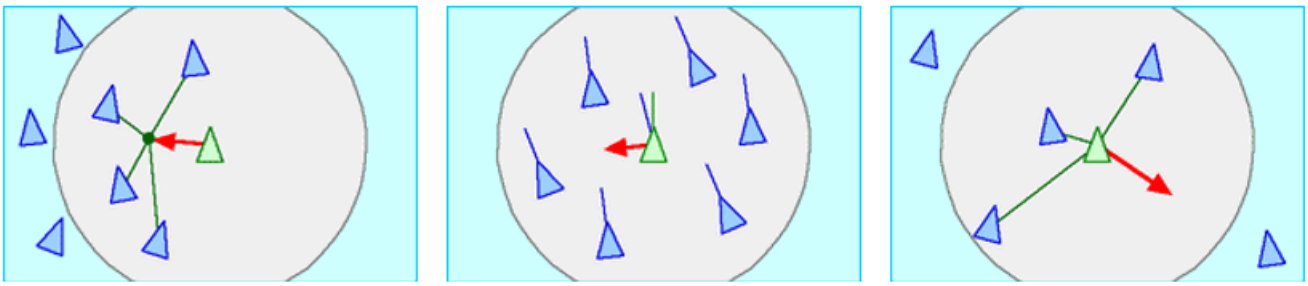
Idea behind this project

The idea behind this project was to provide a demonstration of parallel processing in gaming with Unity* and how to perform gaming-related physics using this game engine. In this domain, realism is important as an indicator of success. In order to mimic the actual world, many things need to happen at the same time, which requires parallel processing. Two different applications were created and then compared to a single-threaded application run on a single core.

The first application was developed to run on a multi-threaded CPU, and the second to perform physics calculations on the GPU. To demonstrate the results of these techniques, the developed application presented schools of fish which were created utilizing a flocking algorithm.

Flocking algorithm

Most flocking algorithms rely on three rules:



Cohesion :
Steer to move towards the average position of the flock's members.

Alignment :
Steer towards the average heading of the flock's members.

Separation :
Steer to avoid crowding the flock's members.

Figure 1. Description of the three flocking rules (source: http://www.red3d.com/cwr/boids/)

- **What is a flock?**

In this case, a flock was defined as a school of fish. Each fish was calculated to "swim" within a school if it was within a certain distance from any other fish in the school. Members of a school will not act as individuals, but only as members of a flock, sharing the same parameters such as speed and direction.

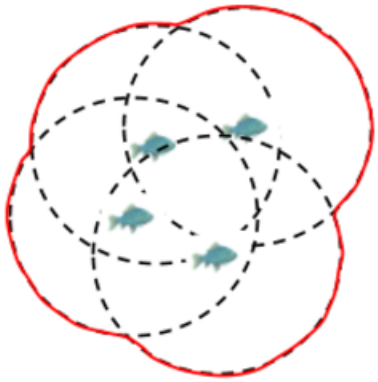


Figure 2. A flock containing four fish

- **Complexity**

The complexity of this algorithm is $O(n^2)$, where n is the number of fish. To update the movement of a single fish, the algorithm needs to look at every other n fish in the environment in order to know if the fish can: 1) remain in a school; 2) leave a school; or 3) join a new school. It is possible that a single fish could "swim" by itself for a time, until it has an opportunity to join a new school. This needs to be executed for every fish n times.

The algorithm is as follows:

For each fish (n)
    Look at every other fish (n)
        If this fish is close enough
            Apply rules: **Cohesion, Alignment, Separation**

Implementation of the flocking algorithm using C#

To apply the rules for each fish, a *Calc* function was created, which needed one parameter: the index of the actual fish inside the environment.

Data is stored inside two buffers that represent the state of each fish. The two buffers are used alternatively to read and to write. The two buffers are required to maintain in memory the previous state of each fish. This information is then used to calculate the next state of each fish. Before every frame, the current Read buffer is read in order to update the scene.

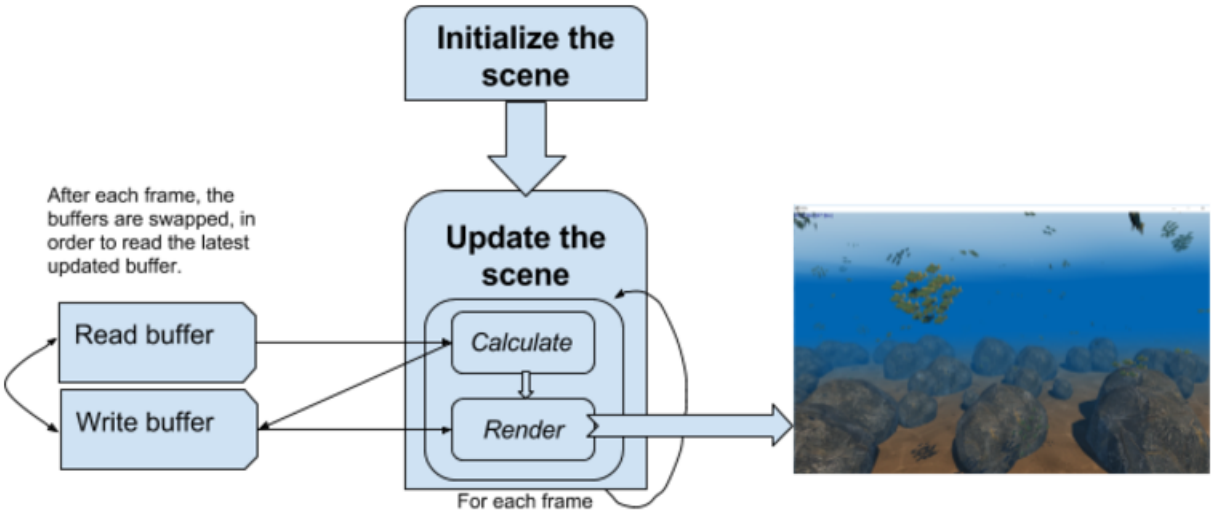


Figure 3. Functional flow block diagram

- **State of fish**

The state of each fish contains the following:

```
fishState {
      float speed;
      Vector3 position, forward;
      Quaternion rotation;
}
```

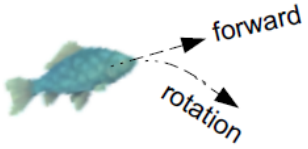


Figure 4. A fish with its fishState

The variable *forward* contains the direction the fish is facing.

The variable *rotation* is a quaternion, representing a 3D rotation, which allows the fish to rotate to face the direction it is aiming.

- **Flocking algorithm**

The complete flocking algorithm used was:

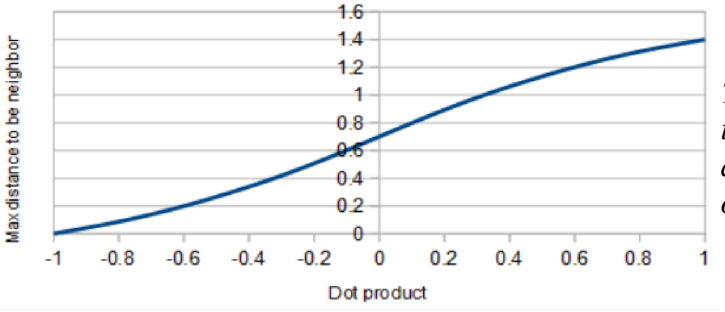
```
for (int j = 0; j < numFishes; j++) {
        if (index != j) {
                other = states [read, j];
                Vector3 othpos = other.position, othfwd = other.forward;
                if (Call(position.x,othpos.x,distNeighbor)) {                    // if along
                        float dist = Vector3.Distance (othpos, position);
                        if (Neighbor (othfwd, curfwd, dist, distNeighbor)) {
                                size++;
                                center += othpos;                                // COHESION
                                gen_speed += other.speed;
                                forward += othfwd;                               // ALIGNMENT

                        if (dist <= 0.30f)                                 // SEPARATION
                                avoid += (position - othpos).normalized / dist;
                }
```

- **Neighbor state**

Once the flocking algorithm was executed, each fish was identified as either being a member of a school or not.

The *Neighbor* function was created using as parameters the distance between two fish and the forward direction of each fish. The idea was to get the behavior to be more realistic. If the distance between any two fish was small enough, and the two fish were traveling in the same direction, there was a possibility that they could merge together. However, if they were not traveling in the same direction, they would be less likely to merge. This merging behavior was created using a piecewise quadratic function and a dot product of the forward vectors.



The distance between two fish must be smaller than the maximum distance, which is dynamically changed based on the dot product of the forward vectors.

re 5. Representation of the mathematical function

Before calling the *Neighbor* function, which is pretty heavy, there is a call of another function: *Call*. The *Call* function tells the algorithm whether or not the *Neighbor* function is required to determine if any two fish are close enough to have a chance of being in the same flock or not. The *Call* function only checks the positions of these two elements (fish) regarding their x-position. The x-position is preferred because this is the widest dimension, allowing the fish to be distributed the furthest apart.

- **Update of the state**

If a fish is alone, it moves forward in a certain direction and speed. However, if a fish has neighbors, it will need to adapt its direction and speed to the flock's direction and speed.

```
direction = (direction_velocity * forward / size) + (avoid_velocity * avoid) + (center / size) + curfwd - position;
gen_speed = curspeed + (((gen_speed/size)-curspeed)*0.50f);     // linearly converges to the average speed of the flock
```

Speed is always changed linearly as a matter of smoothness. Speed does not change to another speed without a transition.

There is a defined environment. Fish are not permitted to swim beyond the dimensional limits of that environment. If a fish collides with a boundary, the fish is deflected back inside the defined environment.

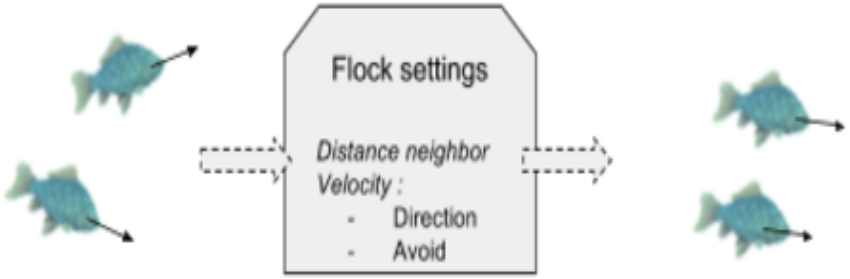


Figure 6. Flocking behavior

If a fish is about to swim out of bounds, the fish is given a new random direction and speed in order to remain inside the defined environment.

Figure 7. Boundary behavior

It is also necessary to check if a fish is about to collide with a rock. The algorithm must calculate if a fish's next position will be inside a rock. If so, the fish will avoid the rock in a similar fashion as avoiding a boundary.

Once the states have been calculated, all of the fish can be made to "swim," along with making any required rotations. The next state of each fish is then updated with new direction, speed, position, and rotation variables (for n fish). This occurs for every new frame update.

For example, any fish has its direction added to its position, in order to "swim" inside the environment.

```
position += (forward.normalized * deltaTime * gen_speed);
```

Integration inside Unity*

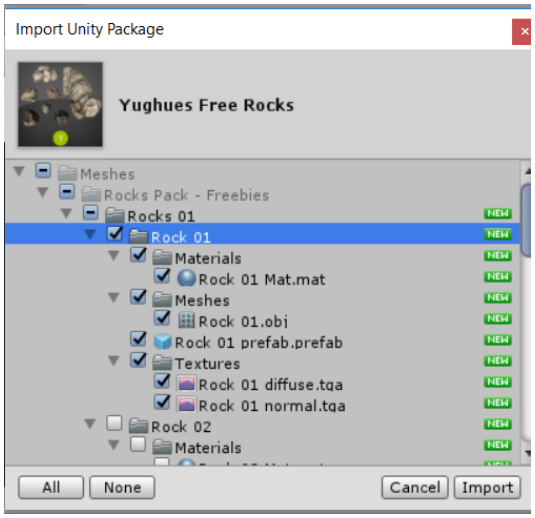
The main programming component in Unity is a GameObject. Inside GameObjects you can add different things such as scripts to be executed, colliders, textures, or materials to customize the objects in order to have them behave as desired. It is then convenient to access these objects inside a C# script. Each public object in a script will create a field in the editor that allows you to drop any object which matches the desired requirements.

C# scripts were used to create flocking behavior.

- **Import assets into Unity**

1) Go to: *Assets…Import package…Custom package*
2) Click on *All*
3) Click on *Import*

Next, drag and drop the *Main* scene from the Project tab to the Hierarchy tab. Right click on the default scene and select "Remove Scene."

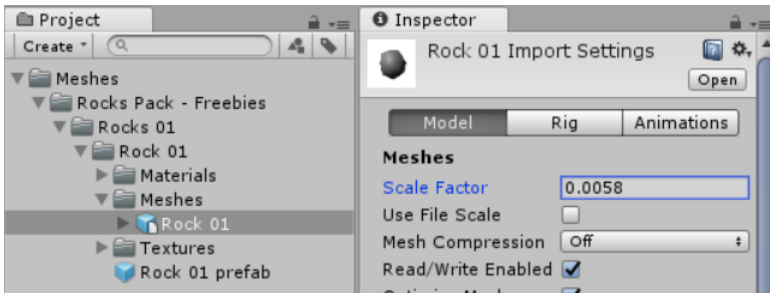


All game objects needed to run the application, along with attached scripts, are ready to run. The only missing part is the rock model, which must be added manually.

Download "Yughues Free Rocks" from the Unity Asset Store. The Asset Store can be accessed within Unity (or by using this link: http://u3d.as/64P). Once downloaded, the window on the left appears. Choose "Rock 01" and import it.

Before using the rock model, an adjustment needs to be made because the scale of the default model is too large. The scale factor of the import settings of its mesh should be resized to 0.0058. Once a rock is added to the scene, if it has a scale of 1, it will match the scale of a 3D sphere of 1, which will be used as a collider for the object.

re 8. Pop-up window: Import Unity Package



Drag and drop the *Rock 01 prefab* to the Rock field, inside the *main* script contained by the *fishManager* object.

re 9. Inspector tab of an imported object – change of the scale

- **Initializing the application**

The initialization of the application is done inside the "Start" function of the *main* script. This function is called once at start-up. Inside this initialization the following steps are executed:

1) Creating a 2-dimensional array of fishState and creating as many arrays of 4x4 matrices as are needed.
2) Every fish is given a random position, forward direction, and speed. Their rotation is calculated considering their other properties.
3) The fishState image of each fish and the corresponding TRS matrix are initialized.

The application can be built as an .exe file. In this case, some parameters can be changed if launched from the command line prompt using the following arguments:

- "-t": height of the allowed area for the fish (<~> tank)
- "-f": number of fish to display
- "-r": number of rocks to add to the scene
- "-n": maximum distance for two fish to be neighbors (i.e., interact together)
- "-s": displays a simpler scene with a visible tank (see images below)
- "-m": mode to launch the application. 0: CPU Single-thread; 1: CPU Multi-thread; 2: GPU

Considering the size and the neighbor distance,these are unit-less parameters.. As a matter of comparison, the smallest distance fish are allowed to swim side by side to avoid collision is 0.3.
For example, the command *"-t 6 -f 1200 -n 1.5 -m -s"* will launch a no-rocks multithreaded CPU application of 1,200 fish, with a maximum neighbor distance of 1.5, inside a tank having a height of 6. The depth and length of the environment depend on the height. These coefficients are stored and can be changed in the code to alter the look of the scene.

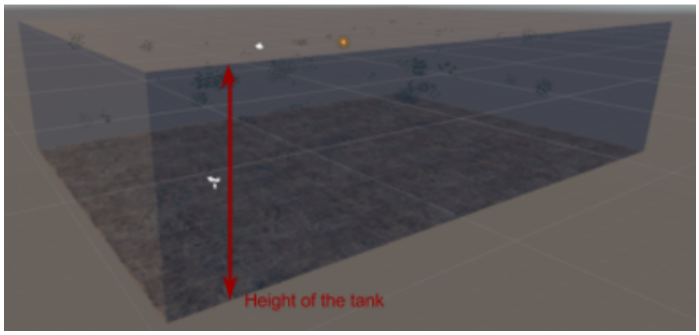


Figure 10. Simpler Scene

Figure 11. Fish area to interact with another fish

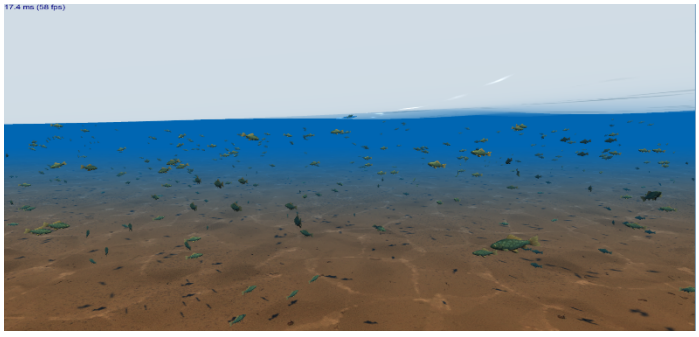


Figure 12. Scene with underwater effects

Constants which can be changed to tweak the behavior of the fish:

- **Speed:** Maximum speed for a fish, rotation speed, speed to add to a fish when it nears a boundary.
- **Velocity:** Parameters related to the three flocking rules: cohesion, alignment, and separation. (Cohesion is set to 1 by default. It is not recommended that this setting be changed.)
- **Dimension:** Determines the depth and length of the environment. These parameters are calculated based on the given height.

- **How to draw the instances**

The *DrawMeshInstanced* function of Unity is used to display a fish. This allows the drawing of N instances of the same fish. Each fish is represented by a mesh and a material. To use this function, three parameters are required: a mesh, a material, and an array of a 4x4 matrix.

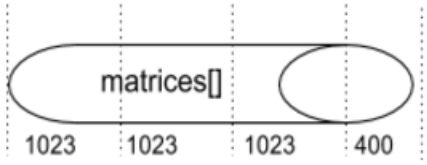
Each 4x4 matrix is configured as a Translation Rotation and Scaling (TRS) matrix. The TRS matrix is reconfigured after each update of all of the fish's states, using their updated positions and rotations as parameters. The global variable scale is the same for each fish in order to resize them if needed (in this case, the factor is 1).

```
.SetTRS (position, rotation, scale);
```

The mesh has been previously resized and rotated in Blender to avoid any mismatch.

Inside each script, Unity's *Update* function is used to refresh the states for each frame.

DrawMeshInstanced provides good performance, but has a limit of 1,023 instances drawn per call. This means that in the case of more than 1,023 fish in an environment, this function has to be called more than once. The array containing the matrices to display the fish must be split to create chunks no larger than 1,023 items. Each chunk is then updated in the *Calc* function, which is called several times. Several calls will be made to both the *DrawMeshInstanced* and *Calc* functions to update and then display all the fish for each frame.




```
nbmat = Mathf.CeilToInt (numFishes / (float)instanceLimit);
rest = numFishes - ((nbmat - 1) * instanceLimit);

Calc (j + i*instanceLimit);}
```

Figure 13. Splitting the array of matrices – calculation of variables

The variables are calculated: *nbmat* represents the number of arrays of matrices that the application is using; *rest* represents the number of fish in the last matrix.

Each fish is updated: *i* represents the index of the matrix; *j* represents a fish's index inside the current chunk. This is needed in order to update the right matrix inside the array shown above.

- **Additional features**
    - ***Underwater effects***

There are different assets that were added to this Unity project in order to create a number of underwater effects. Unity provides built-in packages, including water models, for example, that were used for this project. There are also many textures and associated materials ("skins"), which may be applied to any object. All of these (and more) can be found in the Unity Asset Store.

- **Caustics - Light reflections and shadows**

Caustics lighting effects were added to the underwater scene of this project. A "projector" (a Unity object) is required for caustics. The projector displays caustic effects in a scene. The projected caustic is changed by assuming a certain frequency (Hz), which provides an effect that the caustics are moving.

- **Blur**

A blur was added to the underwater scene. If the camera is below the surface level of the water, a progressive blur will be enabled and displayed. The background of the scene will be changed and become blue. The default background is a sky background (*skybox*). Additionally, the fog setting was enabled inside Unity. (*Windows… Lightning … Other Settings*, Fog box checked.)

- ***Moving the camera***

A script was added to the camera object in order to move inside the scene using the keyboard and the mouse. This provides controls similar to a first-person shooter game. The directional keys may be used to move forward/backward or strafe left/right. The mouse allows for moving up/down, along with turning the camera to point left/right.

```
transform.Rotate (0, rotX, 0);
```

The *move** variables represent the directional keys input, while *rot** represents the mouse orientation. Modifying the transform of an object, which holds a script, in this case the camera, makes it rotate and translate in the scene.

- ***Building an .exe file***

As previously mentioned, there is the possibility of building an .exe file to change the parameters of the application without changing the source code. To do so, follow these steps:

1) Go to: *Edit…Project Settings…Quality.*
2) In the *Quality* tab, scroll down to *Other,* and find *V Sync Count*.
3) Change the *V Sync Count* setting to "Don't Sync." This lets the application display more than 60fps, if possible.
4) Go to: *File…Build and Run* to build the .exe file.

Note: instead of using *Build and Run,* you may go to the *Build Settings* in order to choose a specific platform (i.e., Windows*, Linux*, Mac*, etc.).

Coding differences: CPU vs. GPU

CPU

There is only one difference between coding for a single-threaded and multi-threaded application: How the *Calc* function is called. In this case, the *Calc* function is critical to the execution time, as it is being called n times for each frame.

- ***Single-threaded***

Coding for a single-threaded application is accomplished in a classic way, through a "for loop" as shown here:

```
for (int j = 0; j < instanceLimit; j++) {Calc (j + i*instanceLimit);}
```

- ***Multi-threaded***

Coding for a multi-threaded application is accomplished by utilizing the "Parallel.For" class. The goal of the Parallel.For class is to split multiple calls of a function and execute them in parallel in different threads. Each thread contains a chunk of multiple calls to execute. Application performance depends, of course, on the number of available cores of the CPU.

```
Parallel.For (0, instanceLimit, delegate (int id) {Calc (id + i*instanceLimit);});
```

GPU

- ***Compute shader***

GPU processing is accomplished in a similar way to CPU multi-threading. By moving the process-heavy *Calc* function to the GPU, which has a larger number of cores than a CPU, faster results may be expected. To do so, a "shader" is used and executed on the GPU. A shader adds graphical effects to a scene. For this project a "compute shader" was used. The compute shader was coded using HLSL (High-Level Shader Language). The compute shader reproduces the behavior of the *Calc* function (e.g., speed, position, direction, etc.), but without the calculations for rotation.

The CPU, using the *Parallel.For* function, calls the *UpdateStates* function for each fish to calculate its rotation and create the TRS matrices before drawing each fish. The rotation of the fish is calculated using the Unity function *Slerp*, of the "Quaternion" class.

```
void UpdateStates(int i){              // called for each fish after the shader has done every calculations
        if (states [write] [i].forward != Vector3.zero)
                states[write][i].rotation = Quaternion.Slerp(states[read][i].rotation, Quaternion.LookRotation(states[write][i].forward), rotSpeed

        fishesArray [idx] [i % instanceLimit].SetTRS(states [write] [i].position, states [write] [i].rotation, scale);  // setting the TRS matrix
```

- ***Adaptation of the code to the compute shader***

Although the main idea is to move the *Calc* function loop to the GPU, there are some additional points to consider: random number generation and the need for data to be exchanged with the GPU.

The biggest difference between the *Calc* function for the CPU and the compute shader for the GPU is random number generation. In the CPU, an object from the Unity Random class is used to generate random numbers. In the compute shader, NVidia* DX10 SDK functions are used.

Data needs to be exchanged between the CPU and GPU.

Some parameters of the application, like the number of fish or rocks, are wrapped either inside vectors of floats or in a single float. For example, a Vector3 from C# in the CPU will match the memory mapping of a float3 in HLSL on the GPU.

Fish-state data (fishState) in the Read/Write buffers and rock-state data (s_Rock) in a third buffer in the CPU must be defined as three distinct ComputeBuffers of the compute shader on the GPU. For example, a Quaternion in the CPU matches the memory mapping of a float4 on the GPU. (A Quaternion is a structure containing 4 floats.) The Read/Write buffers are declared as RWStructureBuffer<State> in the compute shader on the GPU. The same is true for the structure describing rocks on the CPU with a float to represent the size of each rock and a vector of three floats to represent each rock's position.

```
class State{
        float speed;
        float3 position, forward;
        float4 rotation;
```

On the CPU, the *RunShader* function creates ComputeBuffer states and calls the GPU to execute its compute shader at the beginning of each frame.

```
void RunShader(){                       // Setting and filling the buffers about the states of the fishes
        ComputeBuffer rState = new ComputeBuffer (numFishes, System.Runtime.InteropServices.Marshal.SizeOf (states[0][0]));
        ComputeBuffer wState = new ComputeBuffer (numFishes, System.Runtime.InteropServices.Marshal.SizeOf (states[0][0]));
        shader.SetBuffer (kernel, "readState", rState);
        shader.SetBuffer(kernel, "writeState", wState);
        rState.SetData (states[read]);
        wState.SetData (states [write]);
        // run the shader : flocking algorithm
        shader.Dispatch (kernel, nbGroups, 1, 1);
        // Once the work is done, get back the written data and save it
        wState.GetData (states[write]);
}
```

Once the ComputeBuffer states are created on the CPU, they are set to match their associated buffers states on the GPU (for example, the Read Buffer on the CPU is associated with "readState" on the GPU). The two empty buffers are then initialized with fish-state data, the compute shader is executed, and the Write buffer is updated with the data from its associated ComputeBuffer.

On the CPU, the *Dispatch* function sets up the threads on the GPU and launches them. *nbGroups* represents the number of groups of threads to be executed on the GPU. In this case, each group contains 256 threads (a group of threads cannot contain more than 1,024 threads).

```
int threadsPgroup = 1024/4;
nbGroups = Mathf.CeilToInt (numFishes / (float)threadsPgroup);
```

On the GPU, the "numthreads" property must correspond with the number of threads established on the CPU. As seen below, "int index = 16*8*8/4" provides for 256 threads. The index of each thread needs to be set to the corresponding *index* of each fish, and each thread needs to update the state of each fish.

```
[numthreads(16,8,8)]
void Calc (uint id : SV_GroupIndex, uint3 idT : SV_GroupID)
{
        int index = 16*8*8/4*idT.x + id;
```

Results

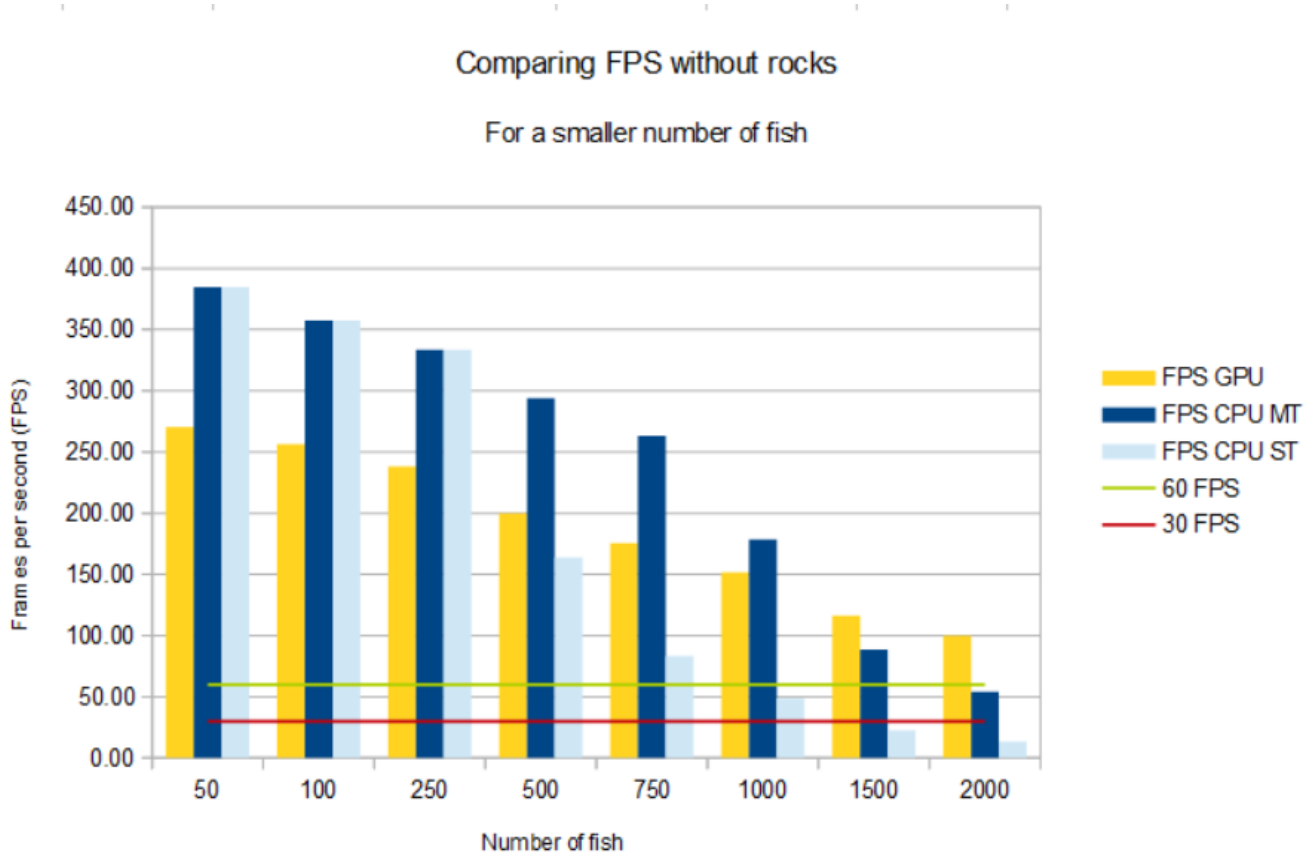


Figure 14. Results for a smaller number of fish

The results show that for fewer than 500 fish, both the single-threaded and multi-threaded CPUs performed better than the GPU. This may be because of the data exchanges which were completed for each frame between the CPU and GPU.

When the number of fish reached 500, the performance for the single-threaded CPU diminished compared to the multi-threaded CPU and GPU (CPU ST = 164fps vs. CPU MT = 295fps and GPU = 200fps). When the number of fish reached 1,500, the performance of the multi-threaded CPU diminished (CPU ST = 23fps and CPU MT = 88fps vs. GPU = 116fps). This may be because of the larger number of cores inside the GPU.

For 1,500 fish and above, in all cases the GPU outperformed both the single-threaded and multi-threaded CPUs.

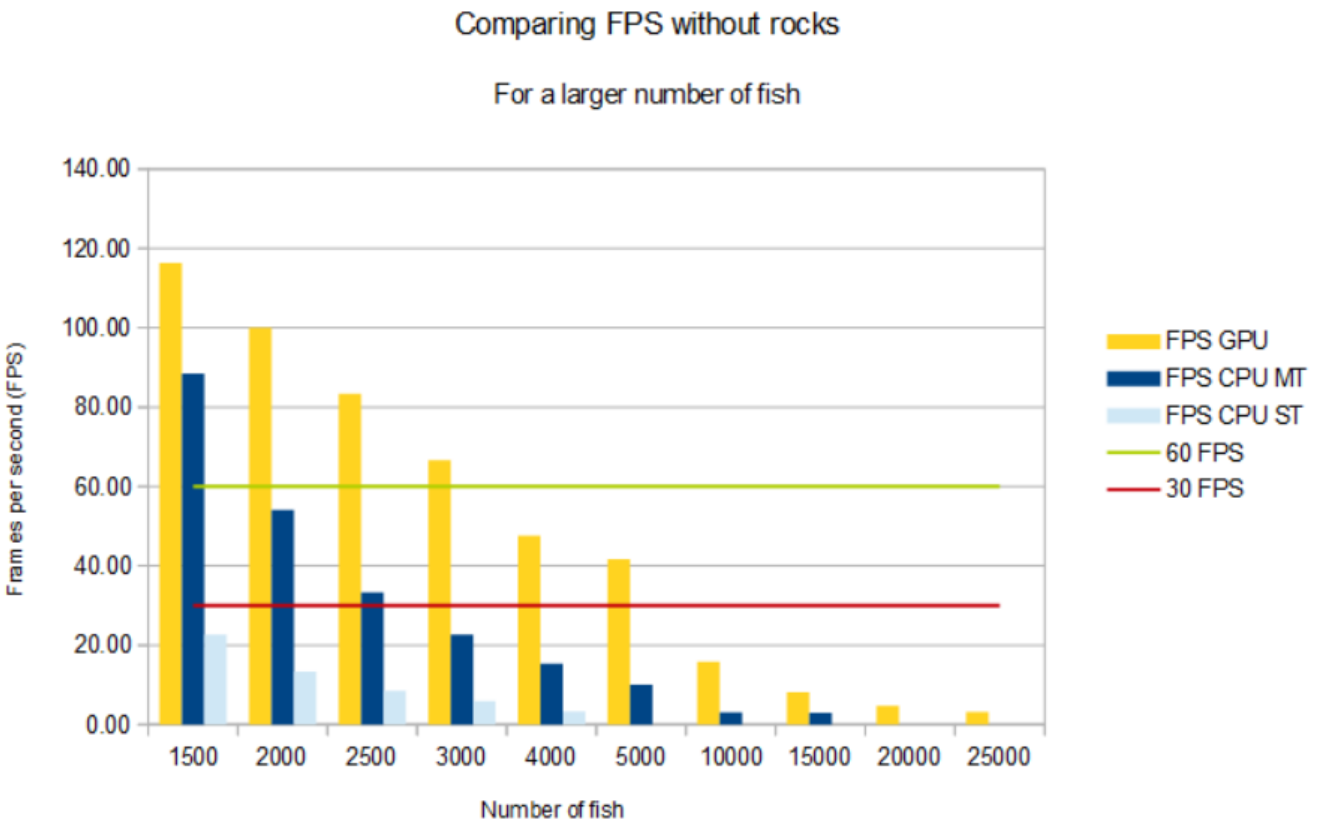


Figure 15. Results for a larger number of fish

Although in all cases the GPU performed better than both instances of the CPU, the results show that 1,500 fish provided for the best overall GPU performance (116fps). As more fish were added, GPU performance degraded. Even so, at 2,000 fish, only the GPU performed better than 60fps, and at 2,500 fish, better than 30fps. The GPU finally degraded below 30fps at approximately 6,500 fish.

The most likely reason why the GPU's performance degraded with larger numbers of fish is because of the complexity of the algorithm. For example, for 10,000 fish there were $10{,}000^2$, or 100 million iterations for each fish to interact with every other fish in every frame.

Profiling the application resulted in highlighting several critical points inside the application. The function which calculated the distance between each fish was heavy, and the *Neighbor* function was slow due to the dot product. Replacing the *Neighbor* call with the distance between two fish (which must be smaller than the maximum distance) would increase the performances a bit. This would mean, however, that any two fish that swim within proximity of each other would now be caused to swim in the same direction.

Another way to possibly improve performance would be to focus on the $O(n^2)$ complexity of the algorithm. It is possible that an alternate sorting algorithm for the fish could yield an improvement. (Consider two fish: f1 and f2. When the *Calc* function is called for f1, f1's *Neighbor* state will be calculated for f2. This *Neighbor* state value could be stored and used later when the *Calc* function is called for f2.)

Hardware used for this benchmark

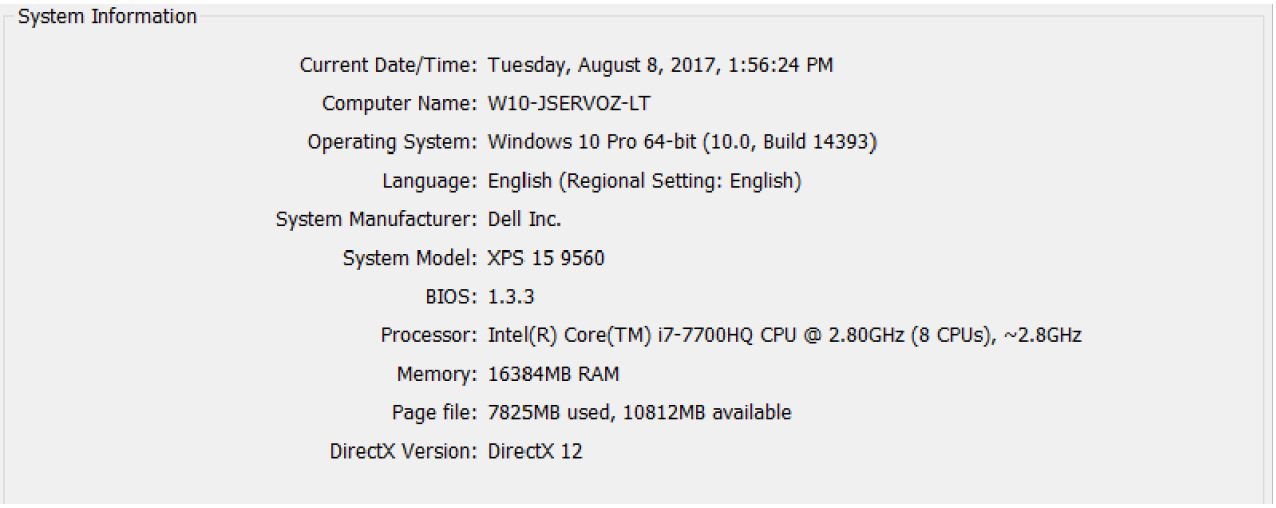


Figure 16. Hardware used to run the tests

Notices