Introduction

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, then compared to a single-threaded application run on a single core. This code and accompanying article (see References below) cover development of a flocking algorithm, which is then demonstrated as
schools of fish via two applications. The first application was developed to run on a multi-threaded CPU, and the second to perform physics calculations on the GPU.

1. Implementation of a flocking algorithm
2. Coding differences: CPU vs. GPU

Get Started

Implementation of a flocking algorithm

In this example, a flock was defined as a school of fish. For each member, the algorithm needs to worry about cohesion, alignment and separation. 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 same parameters such as the speed and the direction.

The complexity of the flocking 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 can be simplified as:

For each fish \((n)\)

\[
\begin{align*}
\text{Look at every other fish} & \quad (n) \\
\text{If this fish is close enough} & \\
\text{Apply rules: Cohesion, Alignment, Separation}
\end{align*}
\]

Data is stored inside two buffers which 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.
The basic flow through the application is:

- Initialize the scene.
- For each frame, update the scene
  - Read the current read buffer
  - Calculate scene
  - Render scene
  - Write to current write buffer
  - Swap buffers

**Coding differences: CPU vs. GPU**

The key difference between coding for a single- and a multi-threaded application is how the flocking computation is called. Remember, this calculation is called n times for each frame. The single-threaded application uses a regular for-loop, while the multi-threaded application utilizes the Parallel.For class.

To get the most performance responsibility for the physics calculation is shifted to the GPU. To do so, a "shader" is used and executed on the GPU. A shader is used to add 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.

Note, the accompanying article points out some additional points to consider when utilizing the GPU:

- Random number generation on the GPU
- Exchanging or sharing data between the CPU and GPU
- Cases where the CPU outperformed the GPU
References


Updated Log

Created March 20, 2018

Appendix 1:

N/A