Advanced Tools Report

Arjen de Aldrey – 444711

Concept

During the Advanced Tools module, I was put to the task to optimize an existing or newly created piece of software, and subsequently comparing the two using testing and eventually processing the data to come to conclusion on how and why our adapted software is optimized.

Due to scope limitations, a decision was made to use an existing project that was made around 2 years ago. The project is made in the Unity Engine and is a rendition of J.H. Conway's 'Game of Life'. The concept of the game is as follows: A Euclidean grid containing tiles is created with each tile having the possibility of either being alive (white) or dead (black). The game plays out in generations as it is supposed to simulate the constant balance between over- and underpopulation. Each generation, each cell takes into account their own state of living, and the states of all 8 directly neighboring tiles. After collecting the states, it follows a certain set of rules that determine if in the next generation the tile will either be alive or dead.

The pre-existing project contained a setup that spawned individual GameObjects for each tile, which acted like both the storage container of the data and the visual indicator of the state of the tile. It allows players to finely control the time between generations, tiles in the width, tiles in the height, and lastly made use of the strategy pattern to let the user decide which spawning behavior and which neighbor-checking behavior it will use.

As the Central Processing Unit (CPU) was clearly the bottleneck, when it comes to the scalability of the grid, a decision was made to try and convert it to use code that can be run on the Graphics Processing Unit (GPU). A distinct difference between the CPU and GPU, being the parallel processing power, the GPU provides for anything graphics related but also for computing use. While not as directly powerful as the CPU, it allows many calculations to be done side by side, freeing up the CPU to handle more processing intense features.

The goal is to speed up the project and make it more scalable through the use of the GPU. The TTT that is related to this project is the '**Rendering Techniques**' group.

Sidenote: The Language used by Unity is C# and the programming language used for the GPU code (shaders) is called High-Level Shader Language (HLSL). Unity provides different types of Shader file types, one being the Compute Shader. It is the most commonly used in this project, due to its specialization in the computing of arbitrary data.

Process

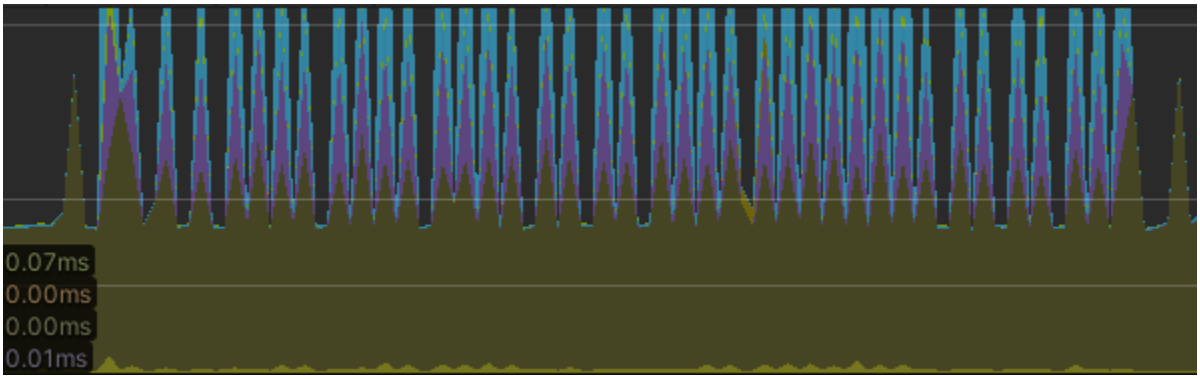
The process started off by trying to use the strategy pattern to spawn the tiles using the GPU. Quickly I realized that it was beyond my scope to completely convert the setup of the program to use GPU instanced meshes. Instead of Meshes, SpriteRenderers were used to render the tiles onto the UI Canvas.

A different direction took me to trying to optimize the neighbor checking behavior using a shader. Instead of using the CPU to iterate through the tiles 1 by 1, the shader provided a way to check and update the states of each cell in parallel.

```
#pragma kernel CSMain

RWStructuredBuffer<int> oldStates;
RWStructuredBuffer<int> newStates;
RWTexture2D<float4> renderTexture;
int stateListSize;
int stateListWidth;
int stateListHeight;
int blockHeight;
int blockWidth;

int2 Get2DIndex(int id, int Width)
{...
}

int Get1DIndex(int2 id, int Width)
{...
}

int GetAmountOfAliveNeighbors(int2 coordSelf)
{...
}

[numthreads(32,1,1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    if (id.x >= stateListSize)
        return;
    int currentState = oldStates[id.x];
    int2 index2D = Get2DIndex(id.x, stateListWidth);
    //Check how many are alive
    int countAliveNeighbors = GetAmountOfAliveNeighbors(index2D);
    //simple if check to see if self is alive
    if (currentState == 0 && countAliveNeighbors == 3)
    {
        newStates[id.x] = 1;
    } else if (currentState == 1 && (countAliveNeighbors == 2 || countAliveNeighbors == 3))
    {
        newStates[id.x] = 1;
    } else
    {
        newStates[id.x] = 0;
    }
    renderTexture[index2D] = float4(newStates[id.x],newStates[id.x],newStates[id.x],1);
}
```

The one issue with this approach was that the GPU was not taking over much of the workload. Most of the neighbor checking code is a series of if statements and do not require too much processing power. Using the profiler, it was clear to see that every generation the framerate would spike as the Update loop of the tile manager class had to update the states of each individual GameObject (Tile). Even

forgoing the changing of color of the tiles by instead rendering the tile states onto a texture and displaying that on a quad that is directly in front of the camera, did not change the framerate. Allowing me to come to the conclusion that it was not the rendering but the processing of the GameObjects and the CPU having to pass and get data from the new GPU code.



The first solution that came to mind that would fall within my scope, was to forgo the GameObject system to hold the tile data and instead saving everything on the GPU with the use of textures. One texture is used to hold the data of the current states of all the tiles, with each pixel being either black or white representing the tiles. Then sampling over each pixel to check if in the next generation they are alive or not. The result of that is then put in the target texture and is subsequently displayed on the quad inside the scene. Lastly, the target texture is then copied onto the first texture, to save it for the next generation.

The solution cleared away a lot of the headway holding the CPU back, instead keeping the data on the GPU and only using CPU to control the update loop that dispatches work to the GPU. This had a significant impact on the framerate of the project, enabling me to up the grid size by 5x. By speeding up the calculation of what needs to be rendered through computer shaders and keeping everything in the GPU memory instead of the CPU memory, I effectively sped up the calculating and rendering of the game of life, due to no longer needing to render many objects resulting in was less draw calls.

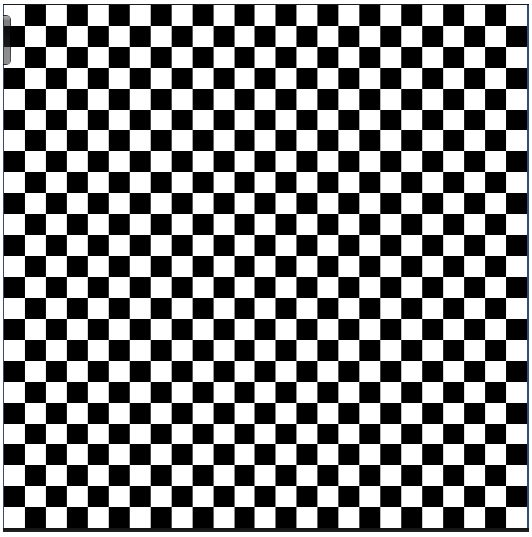
Due to the limitation of the number of threads a GPU can provide, at some point textures need to be split up into batches. This would greatly increase the scalability, but both would not fall in my scope, and the produced solution already provided enough evidence of the efficiency increase demonstrated by offloading work to the GPU.

Testing

To test the difference between the framerates of the two versions, a similar setup had to be created to keep the conditions of the game the same, allowing me to focus on the processing speed. To do this, a simple math formula was used to create a checkerboard pattern. Turning one tile white and the next black.

```
int isAlive = (index2D.x + index2D.y) % 2 == 0? true: false;
```

Resulting in the following pattern:

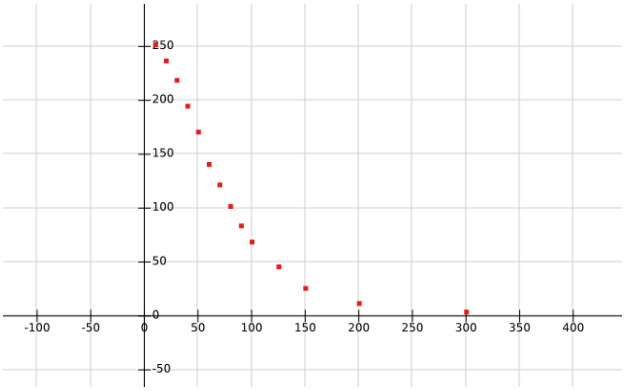


The pattern offered the same starting position for each grid size that I tested with. To get a wide variety of data points, 15 different grid sizes were chosen. Ranging from 10x10 to 1000x1000, with all the tests above a 300x300 not being possible for the CPU version as the framerate reaches nearer and nearer 0.

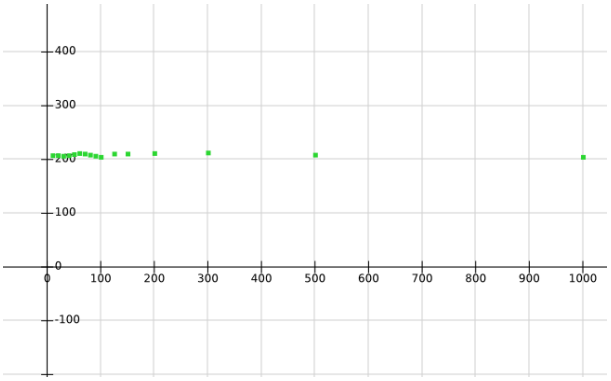
The other Axis of the graph is naturally the framerate, but to be more specific, is the average of the framerate over 60 seconds. The datapoints were gathered into an excel sheet and later plotter into graphs to provide a better visual overview of the efficiency lead the GPU version has over the CPU version.

Results

As mentioned before, the GPU version had a great impact on the performance. In the GPU version, a hard cap was placed on the number of threads that were usable at any given time. This is due to the physical limitations of the GPU and thus requires the texture to be cut up into pieces and processed in batches. Due to this, the maximum potential was not perceived during the testing.



First in line is the results of the CPU version. It quite clearly shoes a linear decrease in framerate. All the plotted points after the 100x100 grid size, use bigger step sizes between the grid sizes, thus looking like they differ from the linear decrease when in reality they are not. Resulting in a Big O notation of O(n). N being the number of cells that are needing to be processed. Best case scenario thus being O(1), but that would require a grid size of 1.



On the other hand, the GPU code showed a remarkable graph, barely fluctuating and showing itself as O(n). N being the number of pixels (grid

cells) needing to be processed. But due to the nature of parallel processing, the gradual decrease is significantly lower and thus the GPU lends itself better to computer large sets of data all at once (or in groups).

Conclusion

A significant increase in performance can clearly be seen, partly being caused by the dropping of unnecessary overhead created by having to manage GameObjects, and partly by offloading a lot of the data holding and manipulation from the CPU to the GPU.

This can be further increased and more versatile if instead of a texture, meshes are instanced using GPU instancing, and even further helped by indirect draw dispatches provided with the shaders. This requires a lot more setting up and if done wrong can result in a worse product, but it also opens up the project to using different types of meshes for the tiles, changing in position, and various other features. Most importantly allowing the scalability of the grid size to continue.

Extra's

GitHub link: https://github.com/tuhri444/physics-playground

Compute Shaders Tutorial by Game Dev Guide:
https://www.youtube.com/watch?v=BrZ4pWwkpto&t=336s