Compute Shader Boids Simulations

Philip Sköld, Anton Warnhag, Axel Lewenhaupt

March 11, 2015

ABSTRACT
Boids is a useful technique for simulating flocking behaviour, first developed by Craig W. Reynolds in 1986[1]. In this report, we present a comparison between two Boid implementations. The first is a Central Processing Unit (CPU) implementation, and the second is a Graphics Processing Unit (GPU) implementation using Direct Compute technology i.e Compute Shaders[2]. The specific Boids simulation that is implemented incorporates path following and common flocking behaviours such as separate, align and cohese. The results show that even without GPU specific optimizations a Compute Shader implementation increases performance significantly.

1 INTRODUCTION AND MOTIVATION

The Performance of GPUs has been growing at a faster rate than that of CPUs, and as a result there is now interest of bringing that performance to general purpose programming[2]. DirectCompute in DirectX11 and High Level Shading Language (HLSL) is Microsoft's solution to this[3].

Since the increase in GPU performance and more general purpose algorithms can be developed for it, it is interesting to look which problems might benefit from this. Specifically problems with inherent parallel features are suitable for "GPU Acceleration", among them Boid simulations.

1.1 Project Summary and Main Focus

The project consisted of designing and implementing a Boids algorithm on both the CPU and GPU, the latter with Compute Shader technology.

It is important to make a distinction between the following two scenarios: Either (1) the result from the simulation is needed on the GPU i.e it should be used in the rendering, or (2) the result is needed on the CPU. An example of when the result would be needed on the CPU is if it should be used in an AI procedure.

The implementation in this paper does not copy the data back to the CPU, but rather draws particles directly from the GPU. However, the rendering could easily be extended to draw more than single particles.

2 BACKGROUND

2.1 Boids

Boids is a model which can be used to simulate a crowd of animals, like a fish school or a bird flock, and the name comes from "bird-oid object". In the model each boid can see a limited range and has rules for how to act depending on the actions of other boids in this view field. The basic rules of this model are *s*eparation: make the boid avoid other boids. *a*lignment: follow the direction of the group. *c*ohesion: stay with the group [4]. Some other rules are, *s*eek: move towards a target location. *P*ath following: the boid tries to following a predetermined path.

2.2 Compute Shaders

A compute shader is program which run on a graphic card's GPU. The GPU is a specialized hardware for running programs which calculations are not dependent on each other. This requirement makes it possible for the GPU to execute the work on many computation cores in parallel, to run the program faster.

3 IMPLEMENTATION

The source code can be found at `https://bitbucket.org/philipshield/dgi-path-following-gpu/admin`.

4 TEST METHOD

For the tests a laptop was used: CPU I7-3517U with 10 GB and as GPU GF 620M. A test sequence was used where it played a simulation for 20 seconds, then increased the number of particles and restarted the simulation. The same procedure was done for both the GPU implementation and CPU implementation.

The time measured was done using unity's `Time.unscaledDeltaTime` which is the time since the last frame.

5 PERFORMANCE

The plots in this section are aimed to show the difference in run-time of the CPU and Compute Shader implementations.

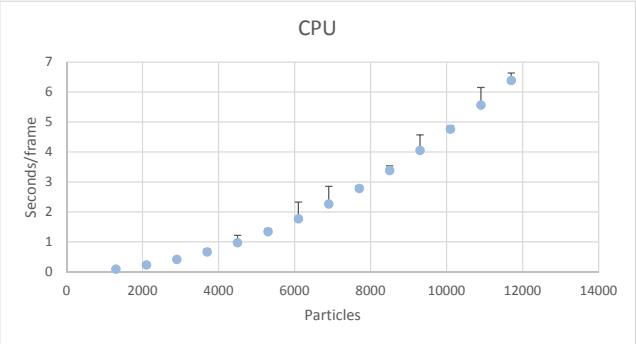

Figure 1: The algorithm run on a CPU and showing a bar with min and max, and a dot for the average frame rendering time when running a 20 seconds simulation.

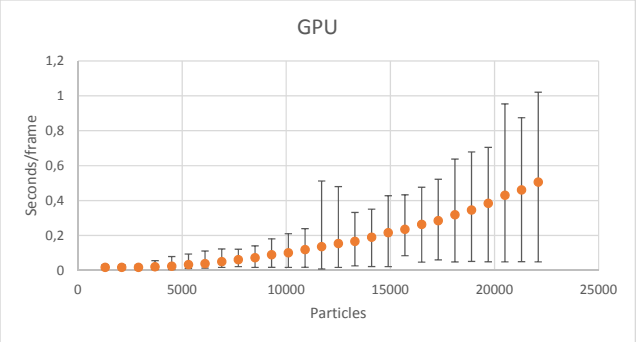

Figure 2: The algorithm run on a GPU and showing a bar with min and max, and a dot for the average frame rendering time when running a 20 seconds simulation.

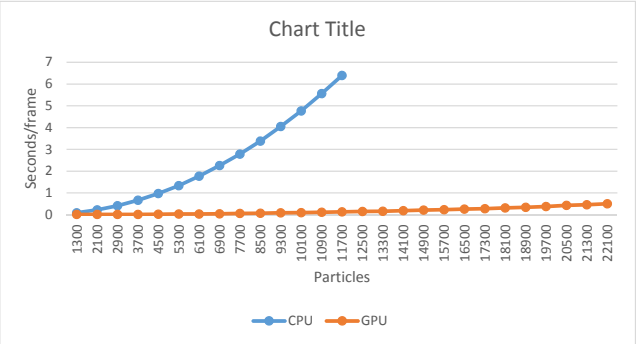

Figure 3: A comparsion between the graphs in figure 1 and figure 2. To make it possible to have both plots in the same frame the maximum number of particles used in the CPU is lower than the CPU.

6 CONCLUSION

A very clear conclusion is that the Direct Compute technology dramatically increases the performance of the implemented algorithm. Although (almost) identical in number of operations the Compute Shader implementation is running faster. In a real time scenario where a frame rate of about 30 frames (33 ms/frame) per second is needed for a smooth play back the CPU implementaion can handle about 1500 particles while the GPU implementation can handle up about 5000 particles.

6.1 Improvements

Several aspects can be improved upon. By using hierarchical structures (possible both on the CPU and GPU) such as grids or binary space partitioning trees, the computational complexity can be reduced to $\mathcal{O}(NlogN)$. The Compute Shader implementation could be optimized to better take into the account the parallel performance of a GPU.

References

[1] (2015). Craig w. reynold, [Online]. Available: `http://en.wikipedia.org/wiki/Craig_Reynolds_(computer_graphics)`.

[2] (2014). Directx 11 compute shaders, [Online]. Available: `http://s08.idav.ucdavis.edu/boyd-dx11-compute-shader.pdf`.

[3] (2014). Direct compute, nvidia, [Online]. Available: `https://developer.nvidia.com/directcompute`.

[4] C. Reynolds, "Flocks, herds and schools: a distributed behavioral model.", *SIGGRAPH '87*, 1987.

APPENDIX A: VALUE TABLE

	CPU			GPU		
Particles	Average	Min	Max	Average	Min	Max
1300	0,09348594	0,08625217	0,1389789	0,01738427	0,003793564	0,03198081
2100	0,2283364	0,220031	0,2781964	0,01736342	0,003853434	0,03205607
2900	0,4164895	0,4052669	0,4730401	0,01726588	0,003831624	0,03193761
3700	0,6657412	0,6473516	0,7566311	0,01986492	0,00401337	0,05533746
4500	0,9737206	0,9495175	1,218848	0,02337253	0,004086923	0,07862356
5300	1,341459	1,314278	1,434471	0,03254526	0,01046983	0,09329491
6100	1,773595	1,73337	2,329137	0,03839101	0,01241258	0,1106125
6900	2,26482	2,204923	2,854656	0,04931412	0,01656579	0,1221146
7700	2,782533	2,745962	2,875581	0,06148389	0,02098242	0,121007
8500	3,381157	3,333236	3,536475	0,07244296	0,01656707	0,1400963
9300	4,056073	4,006319	4,571528	0,08890375	0,01672487	0,1802715
10100	4,765306	4,700299	4,862649	0,1007343	0,01664191	0,209589
10900	5,564119	5,499292	6,154085	0,1185832	0,01723418	0,2389184
11700	6,392034	6,320112	6,635835	0,135308	0,007160775	0,5117425
12500				0,15312	0,01656622	0,4796022
13300				0,1655608	0,02552777	0,3315505
14100				0,1898018	0,02134292	0,3502789
14900				0,2151886	0,02099482	0,427454
15700				0,2346373	0,0836594	0,4324433
16500				0,262504	0,04679627	0,47596
17300				0,2847276	0,05963179	0,5217017
18100				0,3188569	0,04786922	0,6376096
18900				0,3453266	0,0509931	0,6783728
19700				0,3842532	0,04847304	0,7046844
20500				0,430197	0,0483533	0,9536126
21300				0,4604979	0,0491594	0,87459
22100				0,5058732	0,0481305	1,020654

Figure A1: All the measured values.

APPENDIX B: SCREENSHOTS