BOID SIMULATIONS IN A MODERN BROWSER

DD2444 - PROJECT COURSE IN SCIENTIFIC COMPUTING

Johan Ledéus
KTH Royal Institute of Technology
jledeus@kth.se

May 17, 2020

***K*eywords** WebGL, Compute Shader, Reynold Boids, GPU

Introduction

Modern browsers, such as Google Chrome, are getting more sophisticated and introduced with more functionality. One benefit of using a modern browser is that it can run the same application on different platforms. HTML 5 provides access to Web Graphics Library (WebGL) in the browser with the help of JavaScript. WebGL is a hardware-based rendering. It utilizes the GPU on the clients' machine which could be a big performance gain for suitable applications. One drawback of WebGL is that it does not support General Purpose GPU (GPGPU) applications. Fortunately, WebGL 2.0 Compute included such functionality. It can be accessed in developer mode in Google Chrome or Windows Microsoft Edge. WebGL 2.0 Compute provides access to compute shaders. They are used for arbitrary computations, which means that they don't have to be connected to the graphics pipeline directly. This research will compare performance on similar implementations of Reynold Boids [1,2] on the CPU and GPU. It is designed to move boids individually and in groups. Simulations in the browser are used to compare real-time performance.

Research Questions

This research will examine the real-time performance of similar implementations of Reynold Boids. Most of the paper will cover the technical details of the compute shaders since the intended reader is expected to have a fundamental understanding of sequential programming. This paper is designed to get a better understanding of compute shaders and high-performance computing in the browser. To answer the research questions, performance is measured with and without rendering of the boids.

1. How suited are global and local tasks for boids in parallel computations?
2. How well does it scale when changing group size and how many groups?
3. How many boids can it run in the browser with real-time performance?

Background

Reynold Boids [1,2] is an early attempt to model flocks of birds, known as boids, with some simple rules. The boids should be separated and thus not collide; They should be aligned and stay within the flock; They should follow a global objective. The idea of model flocks of birds could be applied to other applications such as simulating crowds.

Simulating multiple agents is of interest in many applications since it could help us with urban planning, social science, and so forth [4]. A single agent has the possibility to be aware of its local surroundings while it follows some global goals [5]. The local goals could be connected to follow a specific group or adjusting velocity and speed depending on one's surroundings. Simulating virtual agents is not that far away from Reynold Boids. This project will look at the possibility to outsource computations to the graphics card in a modern browser when simulating behaviors of boids.

Reynold Boids

The algorithmic aspects of the boid simulation follow three rules or steering behaviors. The CPU and GPU implementation of the Reynold Boids are inspired by the pseudocode provided by [3].

Rule 1 - Cohesion

The first rule makes the boids to move to the "center of mass" or average position of the group they belong to. [1,2]

```
PROCEDURE rule1(boid bJ)

        Vector pcJ

        FOR EACH BOID b
            IF b != bJ THEN
                pcJ = pcJ + b.position
            END IF
        END

        pcJ = pcJ / N-1

        RETURN (pcJ - bJ.position) / 100

END PROCEDURE
```

Pseudocode provided by [3].

Rule 2 - Separation

The second rules steer each boid so they avoid crowding. If a boid gets to close to other members in the group it steers away to avoid crowding. [1,2]

```
PROCEDURE rule2(boid bJ)

        Vector c = 0;

        FOR EACH BOID b
            IF b != bJ THEN
                IF |b.position - bJ.position| < 100 THEN
                    c = c - (b.position - bJ.position)
                END IF
            END IF
        END

        RETURN c

END PROCEDURE
```

Pseudocode provided by [3].

Rule 3 - Alignment

In alignment, each boid steers to the average heading of the group they belong to. [1,2]

```
PROCEDURE rule3(boid bJ)

        Vector pvJ
        FOR EACH BOID b
            IF b != bJ THEN
                pvJ = pvJ + b.velocity
            END IF
        END

        pvJ = pvJ / N-1

        RETURN (pvJ - bJ.velocity) / 8

END PROCEDURE
```

Pseudocode provided by [3].

Compute Shaders

A simulation with many boids might be slow if it is calculated sequentially. This holds because every boid needs to update its velocity and position. Fortunately there exist compute shaders. Compute shaders are a shader stage for arbitrary computations on the GPU [8]. If used properly with the right application, it can increase performance compared to a sequential implementation.

Compute Shaders uses OpenGL Shading Language (GLSL) [7]. Every thread in a compute shader runs the same kernel code and can be identified with any of the following:

```
    in uvec3 gl_NumWorkGroups;
    in uvec3 gl_WorkGroupID;
    in uvec3 gl_LocalInvocationID;
    in uvec3 gl_GlobalInvocationID;
    in uint  gl_LocalInvocationIndex;
```

[8].

Work Group

The smallest unit of a compute shader is a work group [9].

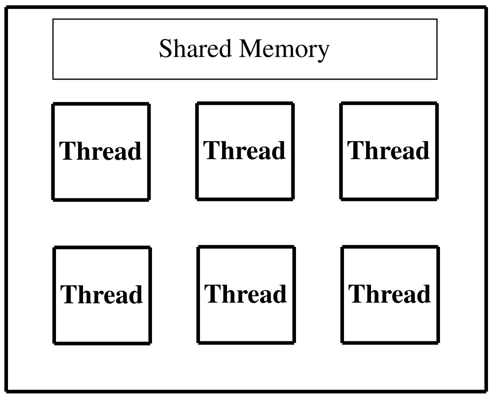

Figure 1: Illustrates a work group with local size (local_size_x = 3, local_size_y = 2, local_size_z = 0). Each thread or work item in the same work group has access to shared memory.

The size of the work group is decided in the compute shader source with the following declaration:

```
    layout (local_size_x = x, local_size_y = y, local_size_z = z) in;
```

The space of a work group can be 3-dimensional and the threads in a work group runs in "parallel" [8]. Figure 1 shows an example of a two-dimensional work group.

Work Groups

How many work groups there are is defined in a dispatch call. Similarly to work group, the dispatch call is three dimensional [8]. Figure 2 shows a two-dimensional dispatch call.

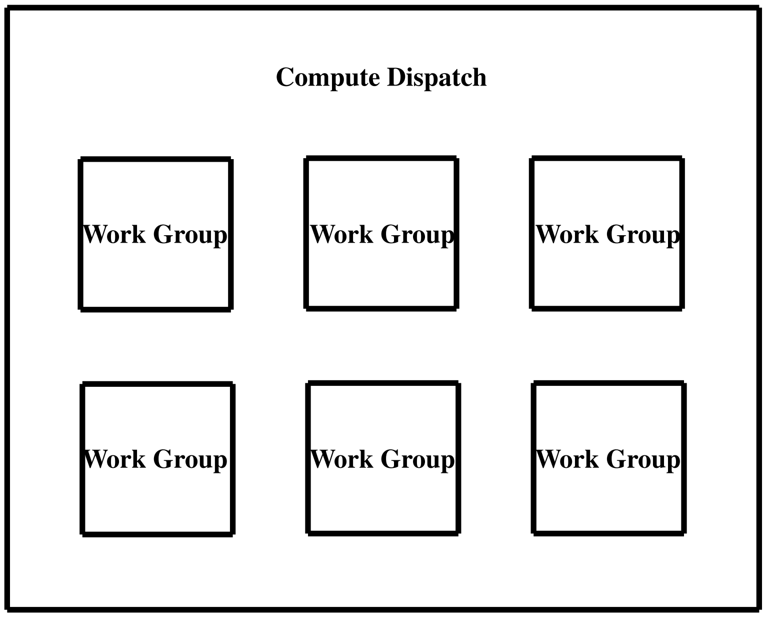

Figure 2: Illustration a dispatch call with 6 work groups.

The execution order of the work groups can be arbitrary so it is important that they can be processed individually. There exist shared variables within a work group so communication is suitable here. Communication between work groups might deadlock the system. [8]

It is important to get the right amount of occupancy to get high performance. The threads of the work groups run in parallel, so the work items per work group impact performance. Optimization can decide how to configure the occupancy. [7]

Shader Storage Buffer Object

Shader Storage Buffer Objects (SSBO) can hold arbitrary data and compute shaders can read and write to them in parallel. [6]

An SSBO can be bound to an Array Buffer (WebGL). This makes the data accessible in the rendering pipeline and there is no need to pass data from the CPU to the GPU.

Method

Initialization

Each boid was represented with a data-structure with information about its **position**, **velocity** and **color**. The position and velocity are randomly initialized. Each group is assigned a random color for interpretability.

Group Size Group size decides how many boids each group consists of.

Number Of Groups The number of groups determines how many groups of boids in the simulation.

Algorithm

The pseudocode below describes the basic simulation on the CPU and GPU. The main steps of the algorithm are to update the position of each boid based on some rules. Each boid updates its position with respect to the members of the group it belongs to.

```
Boids groups = Initialization

FOR group IN groups:
        FOR boid IN group:

            rule1(boid, group)
            rule2(boid, group)
            rule3(boid, group)

            limit_velocity(boid)
            control_boundaries(boid)

            // Update position
            boid.pos += boid.vel

      END FOR
END FOR
```

The purpose of rule1, rule2, and rule3 are described in the background. They are similar to the pseudocode provided by [3].

Other

- The velocity was controlled to be within sensible limits during the simulations.
- If a boid moved off-screen, it was relocated to the other side of the screen.
- The constants for the different rules in the pseudocode were adapted to each implementation.

CPU

The CPU implementation is comparable to the pseudocode above and implemented with JavaScript. Each boid was rendered during their update of position. The GPU is responsible for drawing the boids with WebGL2.0.

Performance is measured with performance.now().

See cpu/ folder for implementation in JavaScript.

GPU

The outline for the GPU implementation is comparable to the CPU implementation.

The size of each work group for the compute shader is defined by the group size for the boids. The following code snippet shows the main function for the compute shader:

```
....
layout (local_size_x = ##GROUP_SIZE##, local_size_y = 1, local_size_z = 1) in;
....
void main() {
    uint threadIndex = gl_GlobalInvocationID.x;

    rule1(threadIndex);
    rule2(threadIndex);
    rule3(threadIndex);

    limit_velocity(threadIndex);
    control_boundaries(threadIndex);

    ssbo.boids[threadIndex].pos += ssbo.boids[threadIndex].vel;
}
```

The number of work groups dispatched is determined by the number of groups. This can be seen in the following code snippet:

```
gl.useProgram(computeProgram);
gl.dispatchCompute(NUMBER_OF_GROUPS, 1, 1);
gl.memoryBarrier(gl.SHADER_STORAGE_BARRIER_BIT);
```

After the dispatch function, a memory barrier function is called to ensure that the calculations are complete before rendering.

The SSBO is used as a vertex attribute in the shaders related to rendering. It was bound as an ARRAY_BUFFER. A performance gain with this approach is that there is no need to copy memory back and forth between the CPU and GPU besides the initialization of the boids.

Performance is measured differently compared to the CPU implementation since the GPU runs asynchronously. EXT_disjoint_timer_query is used to measure the duration for GL commands [10]. The snippet below demonstrates how to do that:

```
var ext = gl.getExtension('EXT_disjoint_timer_query_webgl2');
var query = gl.createQuery();
gl.beginQuery(ext.TIME_ELAPSED_EXT, query);

** CALLS TO BE MEASURED **

gl.endQuery(ext.TIME_ELAPSED_EXT);
```

The compute shader is not part of the rendering pipeline. It was executed before the rendering of the boids. The subsequent snippet shows how to run the compute shader program first followed by the rendering.

```
// Do the compute shaders calculations first
gl.useProgram(computeProgram);
gl.dispatchCompute(NUMBER_OF_GROUPS, 1, 1);
gl.memoryBarrier(gl.SHADER_STORAGE_BARRIER_BIT);

// Render the boids after compute shader
gl.clear(gl.COLOR_BUFFER_BIT);
gl.useProgram(renderProgram);
gl.drawArrays(gl.POINTS, 0, GROUPS * NUM_PARTICLES);
```

For full implementation of the compute shader see the gpu/ folder.

Visuals

The visuals are retained to a minimum since the focus lies on the performance of the algorithm. Therefore, each boid is portrayed by a colorful square. Each group of boids has a distinct color randomly selected. Figure 3 shows an example of a simulation.

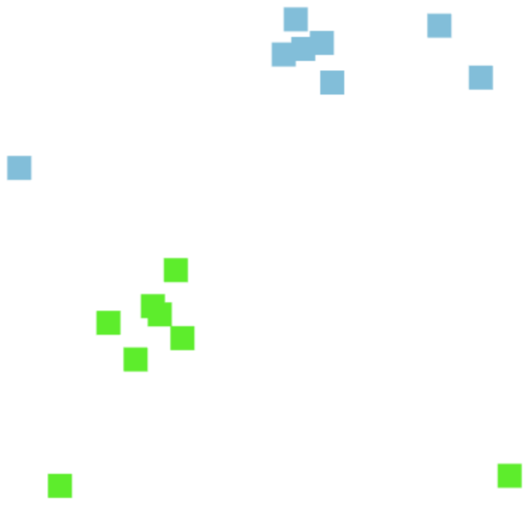

Figure 3: Example of visuals with two groups of boids.

Algorithmic Considerations

The purpose of this research is not to capture any realistic motion or decision making of the boids. The intention is to compare a similar workload of a sequential implementation compared to a parallel. For the GPU implementation, it is possible to speed up the calculations with shared memory and so forth.

Software

Google Chrome: Version 81.0.4044.92 (Official version) (64 bits)

OS: Windows 10 Home

Following command line flags where used for the GPU implementation:

1. –use-gl=angle
2. –enable-webgl2-compute-context
3. –use-angle=gl
4. –use-cmd-decoder=passthrough

Hardware

CPU: AMD Ryzen 7 3700X 8-Core Processor 3.59 GHz

GPU: GeForce RTX 2070 SUPER

RAM: 16.0 GB

Measuring Performance

The measured time is the average time to update the position of all boids. This was performed several times to reduce the initialization phase of the hardware.

Rendering the boids can favor the GPU implementation since it does not have to copy memory between the CPU and GPU for each frame. Therefore, performance is measured with and without rendering.

The group size varied from 8, 64, 128 and the number of groups varied from 100, 1000, 10000 in the simulation. The amount of boids in the simulation ranges from 800 to 1280000.

Even if the GPU can render more than 60 fps ($\approx$ 16 ms per frame), it is limited by the animation frame. The animation frame is usually 60 callbacks per second in most browsers [11]. Therefore, none of the drawings were faster than 60 fps.

Results

Without Rendering

Groups size: 8

Number of Groups	100	1000	10000
CPU (ms)	0.2114	2.4621	24.5572
GPU (ms)	0.0626	0.1159	0.8028

Table 1: Runtime in ms for group size 8

Groups size: 64

Number of Groups	100	1000	10000
CPU (ms)	12.6602	120.6312	1215.0044
GPU (ms)	0.3125	2.1075	4.4824

Table 2: Runtime in ms for group size 64

Groups size: 128

Number of Groups	100	1000	10000
CPU (ms)	47.9020	481.5475	4885.5201
GPU (ms)	0.6450	2.2072	6.0252

Table 3: Runtime in ms for group size 128

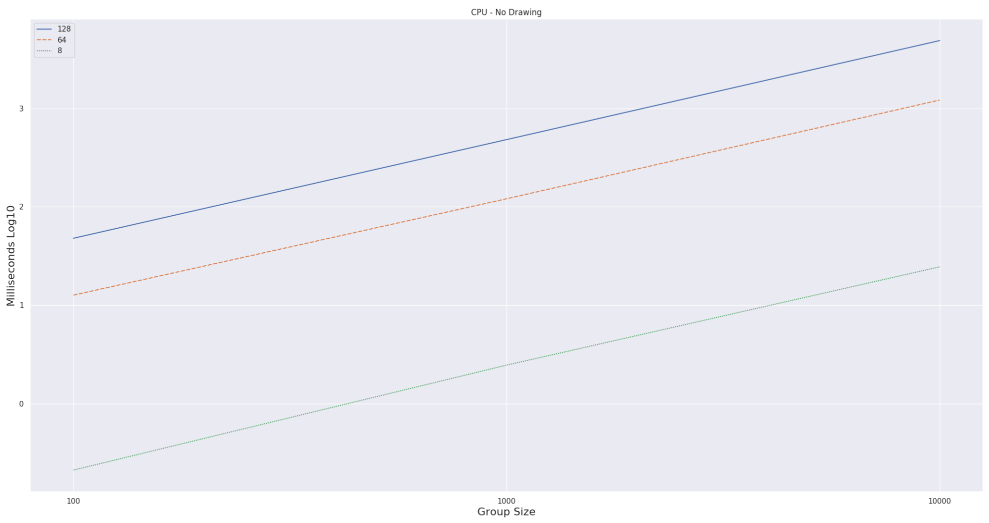

Figure 4: Plotted data for CPU without rendering

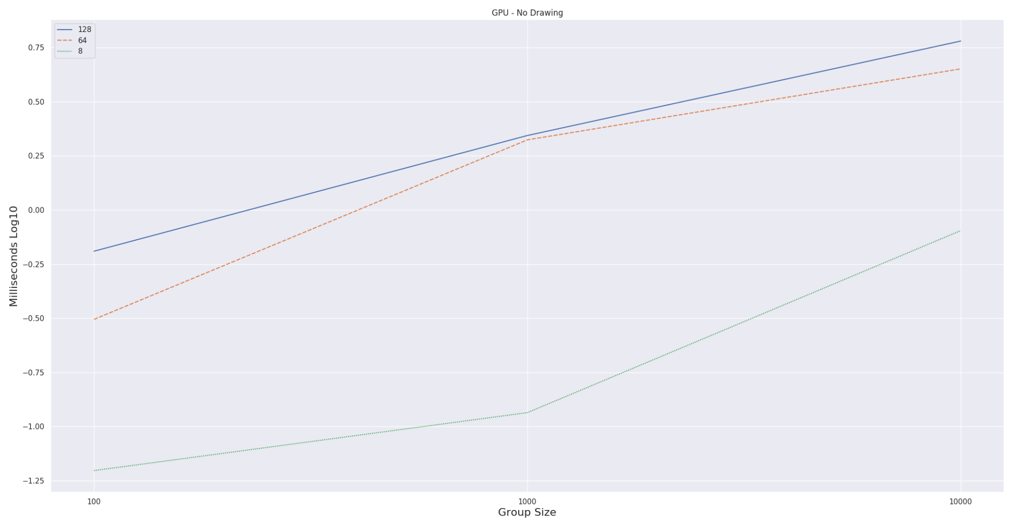

Figure 5: Plotted data for GPU without rendering

When increasing the number of groups throughout the group sizes, the workload for the CPU increases linearly. This can be observed in Figure 4. Going from 100 to 1000 groups increases the amount of time it takes to update the position of the boids by a factor of approximately 10. The only group size that is capable to handle real-time performance for all number of groups is 8. When each group consists of 64 or 128 boids, it struggles to keep up with 30 fps (32 ms). Only the lowest number of groups with 64 boids in each group could keep up with real-time performance.

The GPU is noticeably faster throughout compared to the CPU. For the higher workload, it is over 800% faster. An interesting observation is that the time it takes to calculate the larger groups are comparable, even if one of the groups has twice as many boids. This can be observed in Figure 5. The real-time performance was not an issue, even for the largest workload.

With Rendering

Groups size: 8

Number of Groups	100	1000	10000
CPU (ms)	0.4801	4.5600	48.0131
GPU (ms)	0.3753	0.8935	3.5271

Table 4: Runtime in ms for group size 8

Groups size: 64

Number of Groups	100	1000	10000
CPU (ms)	14.0901	140.4932	1405.1521
GPU (ms)	1.8800	3.0857	14.9731

Table 5: Runtime in ms for group size 64

Groups size: 128

Number of Groups	100	1000	10000
CPU (ms)	51.8221	510.4012	5130.1024
GPU (ms)	1.5636	5.8171	32.3603

Table 6: Runtime in ms for group size 128

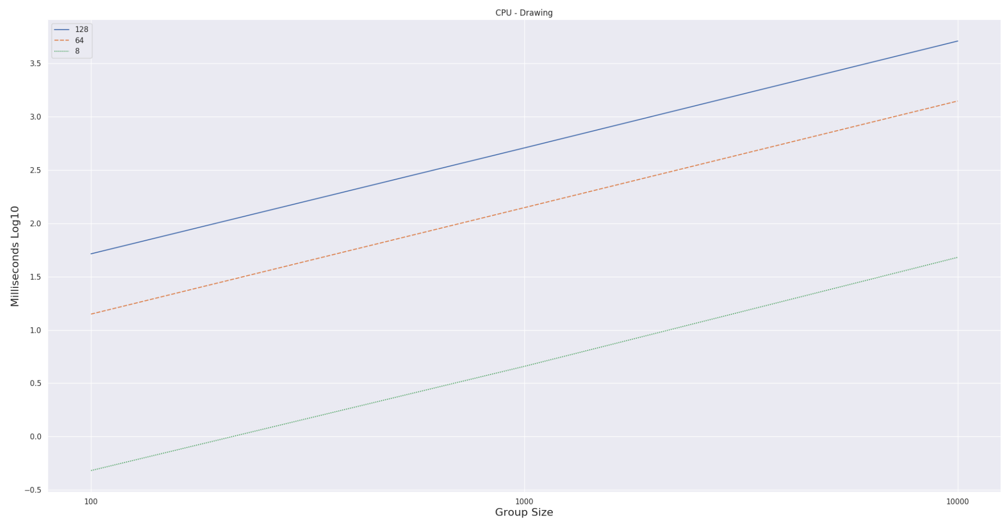

Figure 6: Plotted data for CPU with rendering

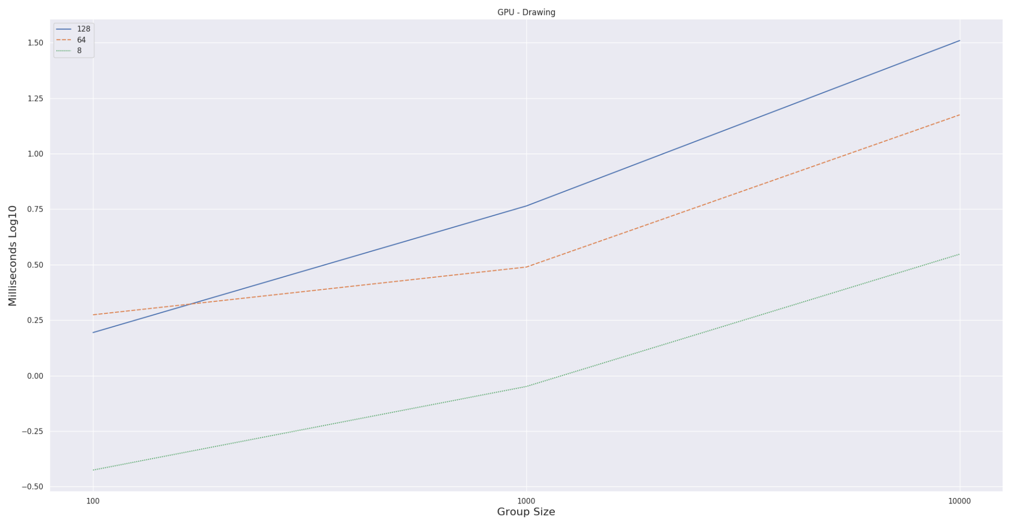

Figure 7: Plotted data for GPU with rendering

Even if the CPU needs to pass the calculated data to the GPU, the performance did not worsen significantly. Still, it performed linearly with the increased workload, which can be observed in Figure 6. It added some overlay and worsen the real-time performance. But, the real bottle-neck seems to be the time it takes to update the position of the boids. It did, however, lose the real-time performance for the smaller group size with 10000 groups.

The GPU had the advantage that it could use the same data-structure for rendering. All of the memory was set in the initialization stage and was not measured. Rendering did introduce some time, but it was still able to provide real-time performance for the larger group size of 128 with 10000 groups. It rendered about 30 fps for that setting. Performance differences for the larger groups were about a factor of two for the larger workloads. An observation is that the largest group size with 128 boids performed better compared to the one with 64 with 100 groups. This can be observed in Figure 7.

1 Discussion

The GPU outperformed the CPU both with and without rendering. This was also expected since the GPU has more computational power with the threads. Performance is only comparable to the smallest workload. When it is increased the GPU stands out. The GPU implementation is also suitable for real-time applications, even with the highest workload. However, with the highest workload, it did not have 60 fps, instead, it kept it at 30 fps.

The CPU was struggling when increasing the group size. With 64 boids in each group, it had a problem to keep up with real-time performance. Interestingly, passing data from the CPU to the GPU did not introduce a big bottle-neck.

Without rendering, the group size of 64 and 128 for the GPU was comparable. Most likely, it has to do with getting the right amount of occupancy and utilize the hardware efficiently.

The results and performance might have been impacted by other applications in the browser.

Optimization

As mentioned previously, the implementation of the boid simulation is not optimized for either. Both implementations are naive and tricks could be used to gain performance. The work group in the compute shader has shared memory. That memory could be used to eliminate some of the loops for all of the threads. For real applications, it is important to get the right amount of occupancy.

Final Words

This paper was designed as an introduction to compute shaders, especially how they work at a low level in the browser. But as a summary, the research question is answered:

1. How suited are global and local tasks for boids in parallel computations?

   Answer: Compute shaders are efficient to calculate many boids in parallel. Although it is important to understand the limitations of the compute shader and its memory. If designed poorly, the performance will suffer. It is okay to have dependencies inside a work group considering it has shared memory. But this does not hold between work groups. The compute shader is designed to assume that they run in parallel.

2. How well does it scale when changing group size and how many groups?

   Answer: The performance of the CPU had linear performance when increasing the number of boids in the simulation. This was also expected since the CPU had no threads and were sequential. The CPU was substantially slower when increasing the size of the groups. The performance of the GPU was almost linear when increasing the number of groups. However, when going from 64 to 128 boids. The runtime was comparable.

3. How many boids can it run in the browser with real-time performance?

   Answer: Given the configuration, the GPU implementation was always able to generate real-time performance. Only for the largest workload, with 1280000 boids, it dropped to 30 fps. The animation frame was a bottle-neck for the other GPU implementations, which is 60 fps. The CPU was only able to provide real-time performance for the smaller workloads with group sizes of 8 and 64.

References

[1] Craig W. Reynold’s home page, http://www.red3d.com/cwr/

[2] Craig W Reynolds. Flocks, herds and schools: A distributed behavioralmodel. InProceedings of the 14th annual conference on Computer graphicsand interactive techniques, pages 25–34, 1987

[3] Conrad Parker’s boid page, http://www.kfish.org/boids/ , http://www.kfish.org/boids/pseudocode.html

[4] Avneesh Sud, Russell Gayle, Erik Andersen, Stephen Guy, Ming Lin, andDinesh Manocha. Real-time navigation of independent agents using adaptiveroadmaps. InACM SIGGRAPH 2008 classes, pages 1–10. 2008.

[5] Rahul Narain, Abhinav Golas, Sean Curtis, and Ming C Lin. Aggregate dynamics for dense crowd simulation. InACM SIGGRAPH Asia 2009 papers,pages 1–8. 2009.

[6] Bailey, Mike. "OpenGL Compute Shaders." Oregon State Un iversity, http://web. engr. oregonstate. edu/ mjb/cs557/Handouts/compute. shader. 1pp. pdf (2016).

[7] Gunadi, Samuel I., and Pujianto Yugopuspito. "Real-Time GPU-based SPH Fluid Simulation Using Vulkan and OpenGL Compute Shaders." 2018 4th International Conference on Science and Technology (ICST). IEEE, 2018.

[8] Khronos, https://www.khronos.org/opengl/wiki/Compute_Shader

[9] ARM Developer Center, https://arm-software.github.io/opengl-es-sdk-for-android/compute_intro.html

[10] Khronos, https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_disjoint_timer_query.txt , https://www.khronos.org/registry/webgl/extensions/EXT_disjoint_timer_query_webgl2/

[11] Mozilla, https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame