Advanced Games Programming Report – Urho3D

INTRODUCTION

I created a game using C++ and Urho3D. In the game, planets generate cone-shaped aliens or 'boids' and the player must shoot them to earn points. The boids gather into swarms like birds or fish, and will harm the player if they get too close. In Single player mode, the player can aim for a high score by shooting as many boids as possible before the time limit. In multiplayer mode, the game is controlled over a network and the player to shoot the most boids wins.

Figure 1: Gameplay screenshot

The leader board on the left shows the current scores of each player. There is also a health-bar and time limit.

Controls

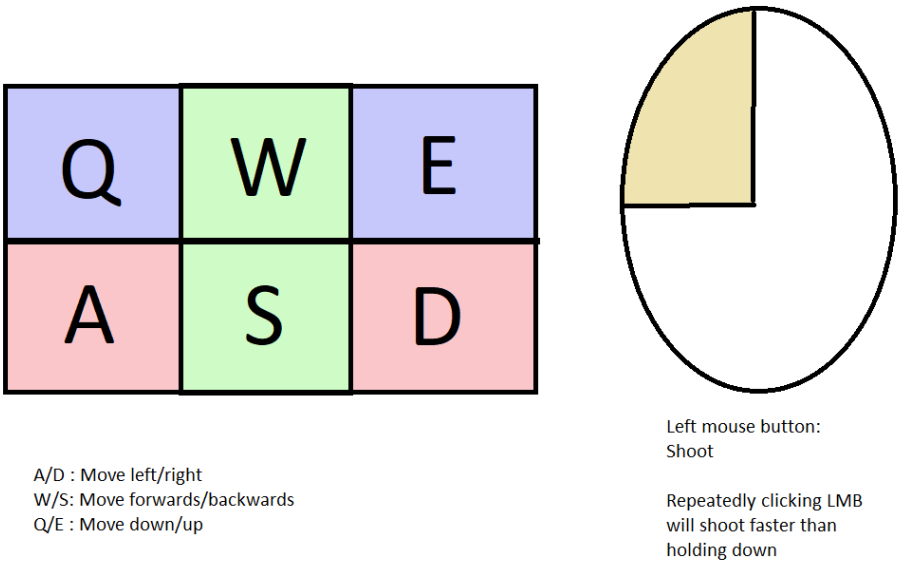

Figure 2: Controls

Networking

The game features a networking option for multiplayer. The player can choose to play locally (using multiple instances of the game on the same machine) or on different machines. The game uses a server-client model, which works like this:

Client: Sends input data to the server. This includes keyboard controls, mouse input and the current camera angle. Receives information from the server such as positions and scores for each player.

```
// Client: collect controls
if (serverConnection)
{
    UI* ui = GetSubsystem<UI>();
    Input* input = GetSubsystem<Input>();
    Controls controls;

    // Copy mouse yaw
    controls.yaw_ = yaw_;
    controls.pitch_ = pitch_;

    // Only apply WASD controls if there is no focused UI element
    if (!ui->GetFocusElement())
    {
        controls.Set(CTRL_FORWARD, input->GetKeyDown(KEY_W));
        controls.Set(CTRL_BACK, input->GetKeyDown(KEY_S));
        controls.Set(CTRL_LEFT, input->GetKeyDown(KEY_A));
        controls.Set(CTRL_RIGHT, input->GetKeyDown(KEY_D));
        controls.Set(CTRL_UP, input->GetKeyDown(KEY_E));
        controls.Set(CTRL_DOWN, input->GetKeyDown(KEY_Q));
        controls.Set(CTRL_SHOOT, input->GetMouseButtonDown(MOUSEB_LEFT));
    }

    serverConnection->SetControls(controls);
    // In case the server wants to do position-based interest management using the NetworkPr
    // tell it our observer (camera) position. In this sample it is not in use, but eg. the
    serverConnection->SetPosition(cameraNode_->GetPosition());

}
```

Figure 3: Client Code (in HandlePhysicsPrestep)

Network: Receives input from the client. Updates players according to input. Handles updates for boids. Sends essential information to client such as the new positions of objects, and the current score for each player.

```
const Vector<SharedPtr<Connection> >& connections = network->GetClientConnections();

for (unsigned i = 0; i < connections.Size(); ++i)
{
    //printf("Server running %i\n", i);
    Connection* connection = connections[i];
    // Get the object this connection is controlling
    Node* playerNode = serverObjects_[connection];
    if (!playerNode)
        continue;

    // Get the last controls sent by the client
    const Controls& controls = connection->GetControls();
    //Torque is relative to the forward vector
    Quaternion rotation(controls.pitch_, controls.yaw_, 0.0f);

    //cameraNode_->SetRotation()

    //RigidBody* pRigidBody = playerNode->GetComponent<RigidBody>();
    Player* clientPlayer = playerNode->GetComponent<Player>();
    RigidBody* pRigidBody = clientPlayer->pNode->GetComponent<RigidBody>();
    //printf("setting prigidbody %p not %p rotation to %f, %f\n", pRigidBody, ballNode->GetComponent<RigidBody>(), pitch_, yaw_);

    //pRigidBody->ResetForces();
    playerNode->SetRotation(Quaternion::IDENTITY);
    pRigidBody->SetRotation(rotation);

    //if (clientPlayer)
    clientPlayer->controls_.Set(CTRL_FORWARD, controls.buttons_ & (CTRL_FORWARD));
    clientPlayer->controls_.Set(CTRL_BACK, controls.buttons_ & (CTRL_BACK));
    clientPlayer->controls_.Set(CTRL_LEFT, controls.buttons_ & (CTRL_LEFT));
    clientPlayer->controls_.Set(CTRL_RIGHT, controls.buttons_ & (CTRL_RIGHT));
    clientPlayer->controls_.Set(CTRL_UP, controls.buttons_ & (CTRL_UP));
    clientPlayer->controls_.Set(CTRL_DOWN, controls.buttons_ & (CTRL_DOWN));
    clientPlayer->controls_.Set(CTRL_SHOOT, controls.buttons_ & (CTRL_SHOOT));

    clientPlayer->teamId = i; //TODO: Don't need to set this every frame, just at start
    clientPlayer->SetColour(playerColours[i]);  //TODO: Don't need to set this every frame, just at start

    //playerNode->SetPosition(clientPlayer->pNode->GetPosition());

    /*intVal++;
    clientPlayer->score = intVal;*/



    //Event data for player scores (server to client)
    VariantMap remoteEventData;

    //Send player health
    remoteEventData["aScoreValue" + String(i)] = playerScores[i];
    remoteEventData["aHealth" + String(i)] = clientPlayer->health; // clientPlayer->health;
    remoteEventData["gameState"] = gameState;
    remoteEventData["gameTime"] = gameTime;
                                                                   //printf("health=%i\n", remoteEventData["aHealth" + String(i)]);
    //printf("clientp=%i", clientPlayer->health);

    connection->SendRemoteEvent(E_CLIENTCUSTOMEVENT, true, remoteEventData);
    network->BroadcastRemoteEvent(E_CLIENTCUSTOMEVENT, true, remoteEventData);

}
```

Figure 4: Server Code (in HandlePhysicsPrestep)

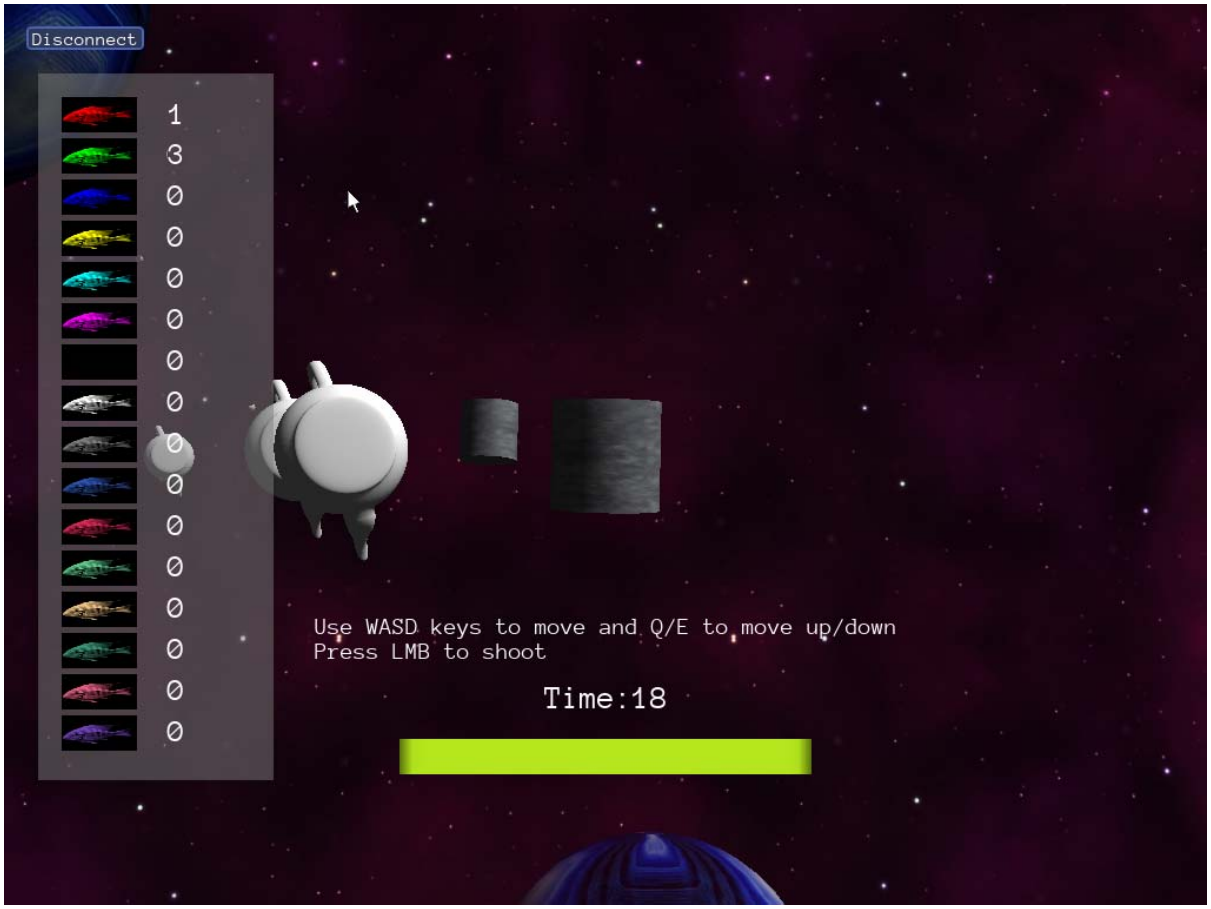

Figure 5: Two players connected to the server, engaging in combat. The timer will start counting down as soon as two or more players join the server, and will reset every time a new player enters.

ANALYSIS

To simulate the boid creatures, an AI algorithm inspired by (Reynolds, CW (1987)) is used. The boids use a force-based model to simulate believable flocking behaviour, similar to how large groups of birds or fish move together. The default algorithm looks at the position of all other boids and calculates the new force for the boid to move.

The game also implements a Factory programming pattern. The static 'Factory' object is responsible for updating all game objects each frame (except for the player). Each frame, every boid is updated, and each boid looks for the positions of each other boid.

The problem

The default algorithm can be slow when handling a large number of boids. This is because of the 'time complexity' of the algorithm. Time complexity is a concept in computer science that describes how much time an algorithm takes as the number of elements increases.

An O(1) algorithm takes the same amount of time regardless of the number of elements.

Importantly, the time complexity is not dependent on the physical duration an algorithm takes, it is concerned only on the way that duration grows with the input size. An O(1) algorithm that always takes 5 seconds to perform may be more desirable than an $O(n^2)$ algorithm that takes 1 second with 1 element, but 25 seconds with 5 elements.

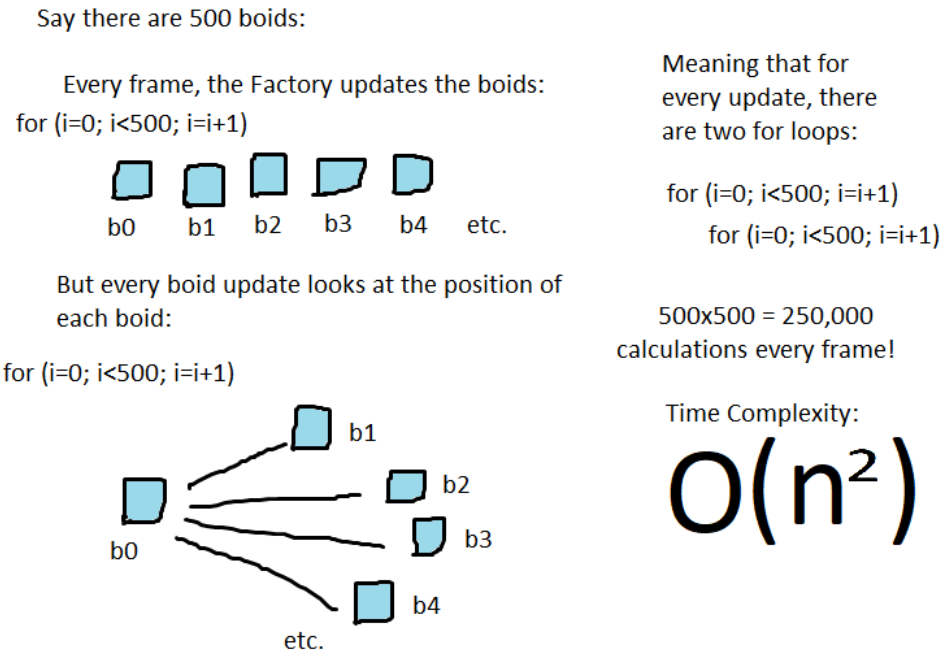

Figure 6: Describes the time complexity of the default algorithm.

Time complexity is a part of Big-O notation which is used to describe the complexity of algorithms. Big-O is used to measure the best, average and worst case scenarios of algorithms. Big-O can also be used to measure space-complexity, the amount of memory space an algorithm requires. We are mainly concerned with the worst-case scenario, and space-complexity is not very important as the amount of data used is relatively small.

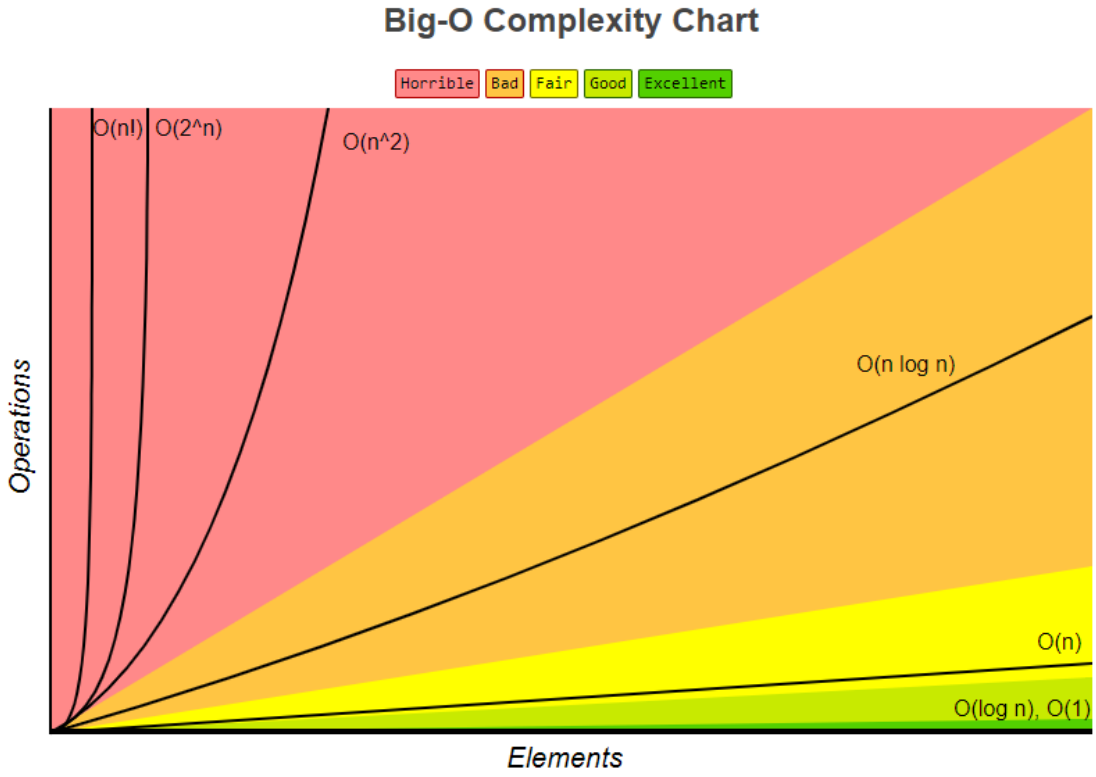

Figure 7: Big-O Complexity chart. Our default O(n^2) algorithm is not desirable because the operations quickly scale with the number of elements.

Optimisation

An attempt was made to optimise the game so it could run a large number of boids with a decent frame-rate. Ideally, we want to see the maximum number of boids possible while maintaining a frame-rate of 30FPS (33ms). There is a separate mode used specifically for optimisation testing. There are options to handle the optimisation of the game in the 'Factory.h' file.

Factory.h

```
//Optimisation testing
const bool OPTIMISATION_TESTING = false; //default false
const int O_BOIDS = 50; //no. of boids. default 50
const bool O_UPDATE = true; //update at all? default true
const int O_SPLIT = 1; //size of split. updates noOfBoids/split per update. default 1
```

Figure 8: Optimisation testing

Boid Splitting

The first optimisation method test used is 'Boid Splitting'. This means that boids are split into equally sized groups, and only one group is updated each frame. For example, if O_SPLIT is set to 3, then one third of the boids are updated every frame. The idea is that we can reduce the amount of calculations each frame, while still maintaining believable behaviour.

Tests were run using the Urho3D profiler. The game would run for one minute for each test, and the average, maximum and minimum FPS were measured.

Boid Test Results

O_SPLIT=1			
Boids	F. Avg	F. Max	F. Min
50	149.4545	151.0346	153.4213
100	35.35193	34.7935	34.64163
200	16.35216	15.75274	16.62455
400	7.397599	7.169384	7.400884
800	3.903719	3.701771	3.998049
1600	1.79639	1.79285	1.800148
3200	0.794758	0.794758	0.794758
6400	0.3871	0.3871	0.3871

O_SPLIT=5			
Boids	F. Avg	F. Max	F. Min
50	149.6782	84.51657	160.0256
100	68.36205	60.03122	71.4592
200	30.87659	27.66252	32.84612
400	16.26413	15.78956	16.50546
800	8.56443	8.310203	8.625597
1600	4.49418	3.893156	3.975274
3200	1.891897	1.891897	1.891897
6400	0.811351	0.811351	0.811351

O_SPLIT=10			
Boids	F. Avg	F. Max	F. Min
50	150.7386	118.7085	163.827
100	96.80542	74.78872	106.735
200	45.37205	37.74582	46.5853
400	23.89144	24.73656	25.26401
800	11.72113	12.49048	11.74922
1600	5.593655	5.381349	5.644328
3200	2.627417	2.627417	2.627417
6400	1.295489	1.295489	1.295489

O_SPLIT=15			
Boids	F. Avg	F. Max	F. Min
50	150.2404	143.2049	281.1358
100	127.2912	93.10987	140.3903
200	62.31694	51.49065	66.44518
400	31.98362	31.24024	37.06999
800	16.96669	18.96598	17.30972
1600	8.572947	9.089256	10.20387
3200	4.514815	4.514815	4.514815
6400	2.771765	2.771765	2.771765

Figure 9: Boid test results. All results are in frames per second.

We can visualise these results better if we use a graph:

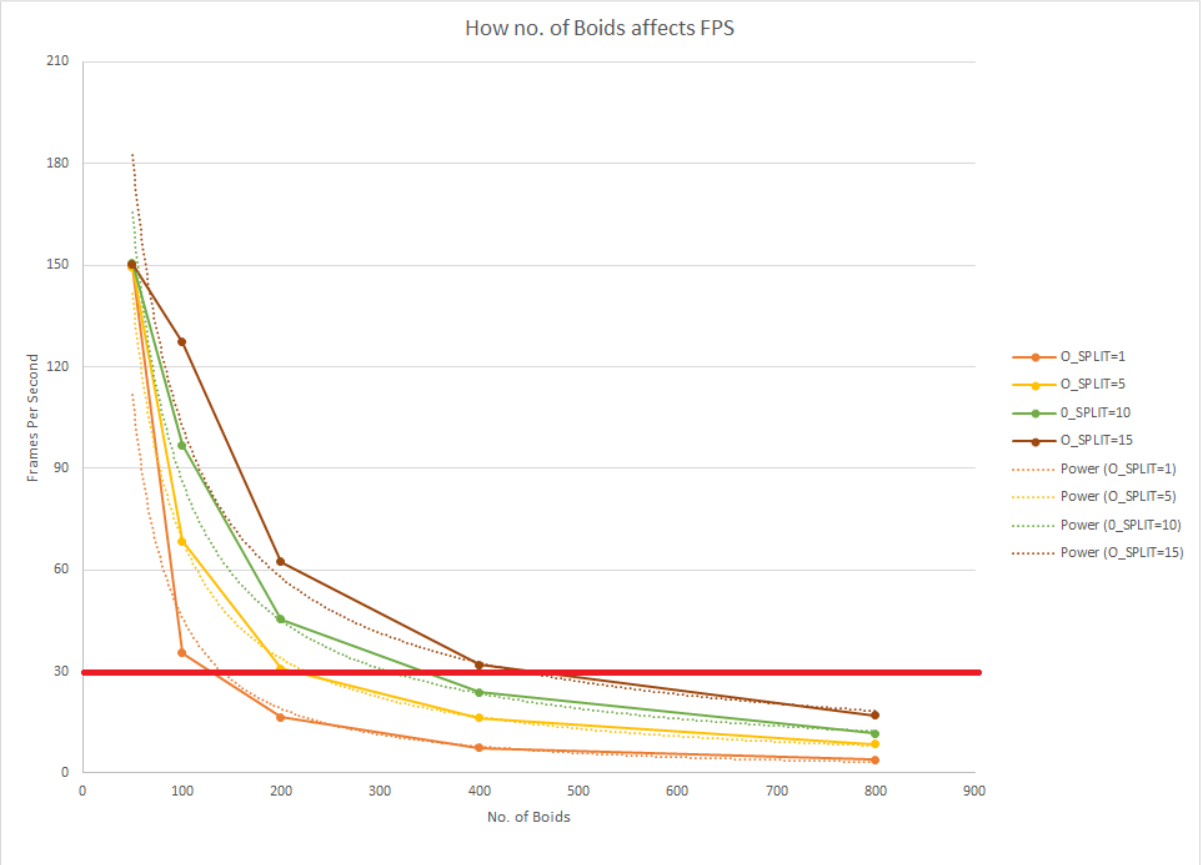

Figure 10: Graph representing how number of boids affects FPS with different O_SPLIT amounts. Boid numbers 1600 and above have been omitted. The red line represents how many boids can be used at 30FPS. The trendlines represent powers of two which correspond with $O(n^2)$.

Note: O_SPLIT=1 is the default algorithm with no optimisation.

We can observe that the FPS during testing corresponds with an $O(n^2)$ algorithm as theorised earlier. The the number of boids can be used at 30FPS is different with each O_SPLIT number:

O_SPLIT = 1 | 130

O_SPLIT = 5 | 200

O_SPLIT = 10 | 350

O_SPLIT = 15 | 450

This shows that Boid Splitting is effective, but will always follow $O(n^2)$ which will be a problem if we want to use more boids.

Boid Grouping

Another method to increase performance is to split all boids into preset groups of fixed group sizes. Boids will only look at boids of the same group.

```
const bool O_GROUPING = true; //grouping on/off. default false
const int O_GROUP_SIZE = 10; //group size, default 10
```

Figure 11: Optimisation testing, group size

Modifying the diagram we used earlier, we can see how this optimisation method may be beneficial:

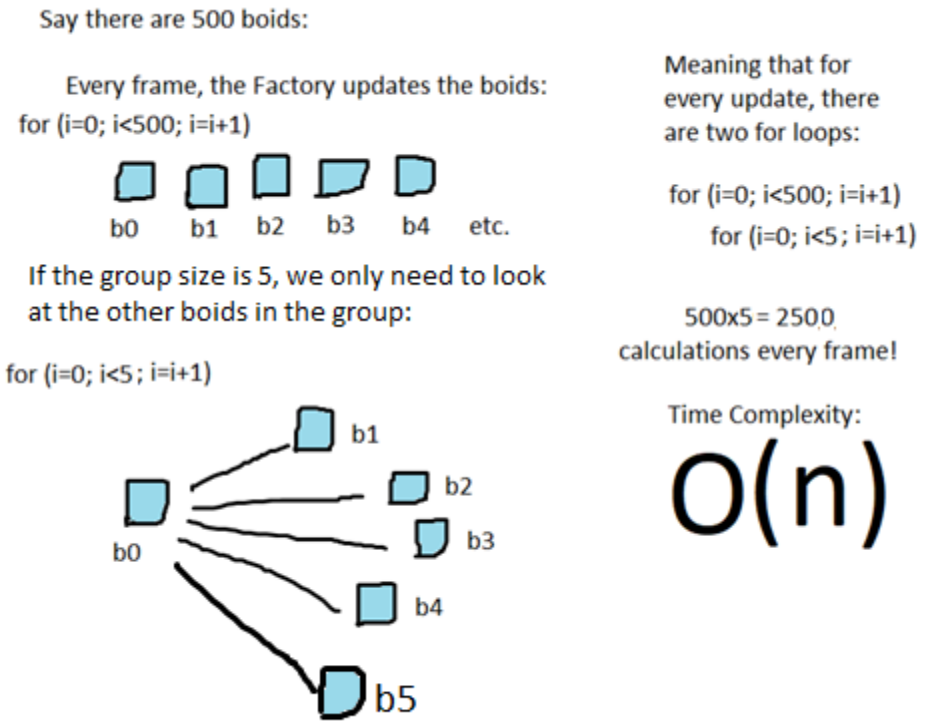

Figure 12: Modification of Figure 6 with Boid Grouping. The Factory still iterates through each boid, but each boid will only iterate through a fixed number of boids, meaning this is an O(n) algorithm.

Boid Test Results

G_SIZE=25			
Boids	F. Avg	F. Max	F. Min
50	342.9355	226.6546	355.1136
100	253.614	151.1031	332.5574
200	97.02144	72.10325	131.3025
400	39.72984	35.91954	48.33019
800	24.89792	22.43762	25.54148
1600	6.21118	6.029617	6.368615
3200	2.994873	2.945057	3.11286
6400	0.576967	0.560741	0.578292

G_SIZE=10			
Boids	F. Avg	F. Max	F. Min
50	338.7534	178.8269	868.0556
100	253.614	151.1031	332.5574
200	164.8805	111.0494	181.3237
400	43.43105	32.00717	48.90454
800	24.3736	19.39526	26.693
1600	4.370629	4.31999	4.745094
3200	3.013664	2.979454	3.124766
6400	0.821521	0.819573	0.826041

Figure 13: Boid test results for group size. (G_SIZE=25, G_SIZE=10)

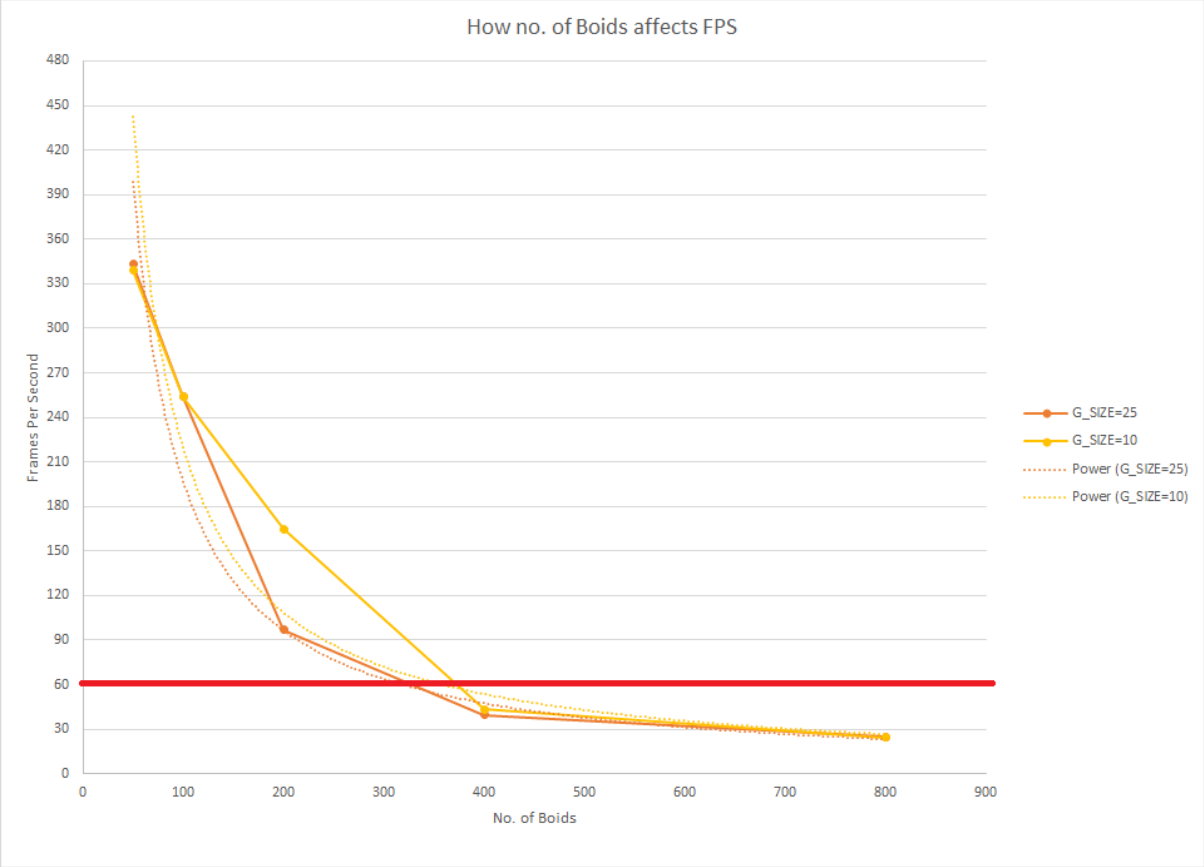

Figure 14: Graph representing how number of boids affects FPS with different G_SIZE amounts. Boid numbers 1600 and above have been omitted. The red line represents how many boids can be used at 30FPS.

The graph appears to be more linear which corresponds with our O(n) theory. The FPS appears to take a large dip around the 800 boids mark. It is unknown why this occurs. I suspect it is something to do with the Urho3D engine. G_SIZE = 10 holds up a lot better than G_SIZE = 25 does.

Putting both together

Satisfied with both of the optimisation methods, I used them together at the same time and did some more testing.

Final Results

O_SPLIT = 15, G_SIZE = 10			
Boids	F. Avg	F. Max	F. Min
50	2.365	4.623	1.172
100	2.877	5.461	2.01
200	4.79	9.133	4.002
400	14.162	16.365	13.447
800	37.982	31.172	35.109
1600	156.154	159.138	152.252
3200	548.629	552.066	539.616
6400	1593.98	1589.181	1560.06

Figure 15: Final FPS Results

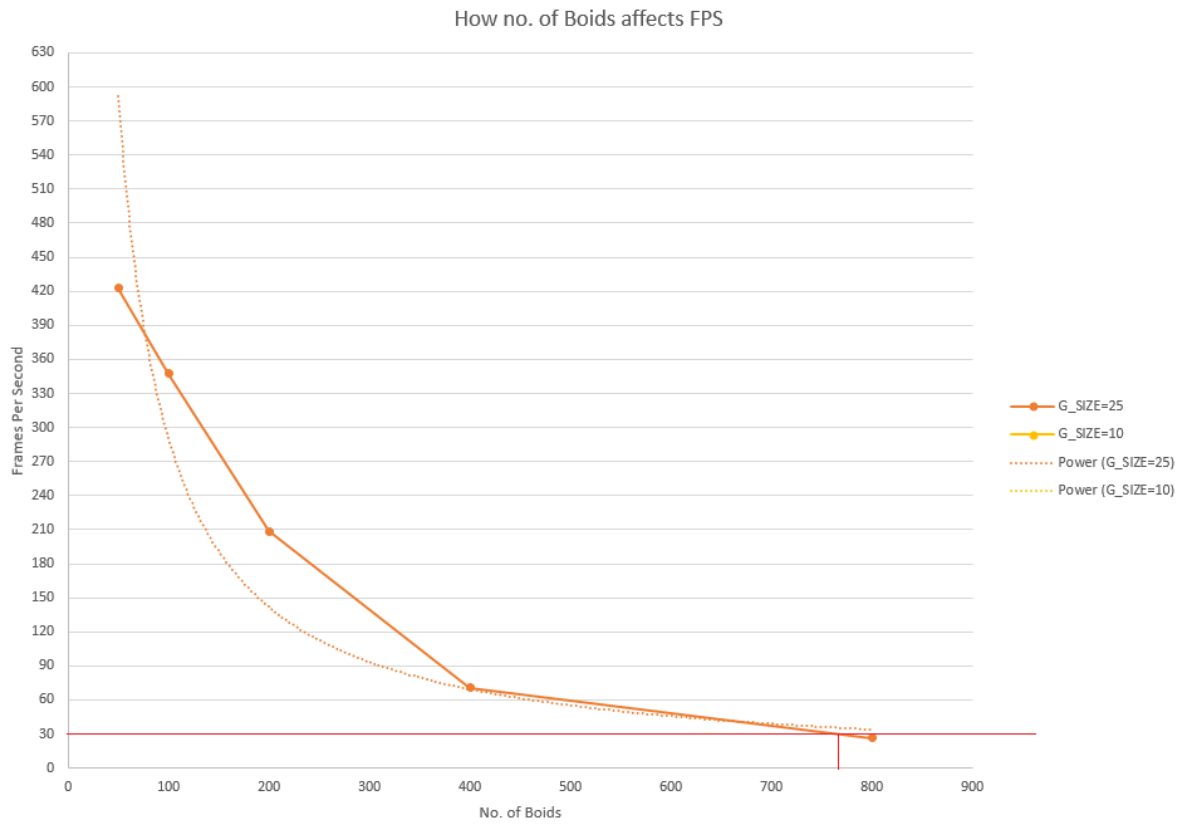

Figure 16: Final Results. Boid numbers 1600 and above omitted. From the results, we should be able to have a maximum of 760 boids at 30FPS.

From the results, we can have a maximum of 760 boids in our game. Considering other factors such as networking and rendering, I decided it was best to cap the number of boids at 700, a safe limit which should maintain the game above 30FPS.

Final Test (O_SPLIT=15, G_SIZE=15)

Boids	F. Avg	F. Max	F. Min
700	53.35325	43.12204	58.25469

Figure 17: One more final test. Average FPS is 53, which is satisfyingly over 30FPS.

REFERENCES

Reynolds, CW (1987) “Flocks, Herds and Schools: A Distributed Behaviour Model”

Figure 7: Diagram sourced from (http://bigocheatsheet.com/)