**Project**: Boids

**Primary** **Goal**: Implement Boids algorithm in Unity3D game engine.

**Secondary** **Goals**: Use Model-View-Controller software design pattern. Build it as an external library(\*.dll) package.

**Introduction**: At the start of this project I had written a tool for Maya that used Boids algorithm to key frame animations. I had very little understanding of the inner workings of the algorithm and all of what it had to offer. This project in itself has been a very educational experience and the end result is a product that I am very proud of. The following information will be an overview of the issues that I ran into and learning outcomes I have achieved. I do not claim to be an expert on autonomous behavior so if any of the following is misinformation please contact me.

The overview will be presented in three sections including, Initial Project Setup, Implementation Details, and Project Completion. These sections will describe in detail the issues that occurred and how/if they were resolved.

**Initial** **Project** **Setup**: The setting up of the project was fairly easy. Version control client used was subversion and it was hosted on a friends server. Even though there were no issues with checking out or committing projects, I had a concern about what form of version control I was going to use because Unity 3.xx only allowed paid versions to use external version control. However with Unity 4 all versions are capable of 3rd party version control. Did some research on using external dlls instead of unity scripts as the source code. The main reason for this was that if I could write the boids as an external library then it would be portable and sellable across multiple platforms, mainly Maya. I was able to successfully create an external library that had a few simple functions inside of it. The library is set up by following these steps taken from Unity's website.

**Writing and Building the DLL**

1. Open MonoDevelop or Visual Studio.
2. Create a new project from the application's menu:
   * MonoDevelop:
     1. Open the menu File > New > Solution
     2. Choose C# > Library
   * Visual Studio:
     1. Open the menu File > New > Project
     2. Choose Visual C# > Class Library
3. Fill out the information for the new library:
   * **Name** is the namespace, for this example use "DLLTest".
   * **Location** is the parent folder of the project.
   * **Solution name** is the folder of the project.
4. Add references to the Unity API:
   * MonoDevelop:
     1. In the Solution browser open the contextual menu of References (right-click) and choose Edit references
     2. Choose the option .Net Assembly tab > File System > select file
   * Visual Studio:
     1. In the Solution Explorer open the contextual menu of References (right-click) and choose Add Reference
     2. Choose the option Browse > Browse > select file
5. Select the required Unity API file:
   * MacOS:
     1. <Applications>/Unity.app/Contents/Frameworks/Managed/UnityEngine.dll
   * Windows:
     1. <Program Files>\Unity\Editor\Data\Managed\UnityEngine.dll
6. For this example, in the Solution browser rename the class into "MyUtilities" and replace its code with this:

**C#**

using System;

using UnityEngine;

namespace DLLTest {

public class MyUtilities {

public int c;

public void AddValues(int a, int b) {

c = a + b;

}

public static int GenerateRandom(int min, int max) {

System.Random rand = new System.Random();

return rand.Next(min, max);

}

}

}

1. Finally build the project to generate the DLL file and its debug symbols.

**Using the newly created DLL in Unity**

1. Open Unity and create a new project.
2. Copy the built file <project folder>/bin/Debug/DLLTest.dll into Assets or a subfolder (e.g. Plugins)
3. For this example, create a C# script called "Test" in Assets, and replace its contents with the following code:

**C#**

using UnityEngine;

using System.Collections;

using DLLTest;

void Start () {

MyUtilities utils = new MyUtilities();

utils.AddValues(2, 3);

print("2 + 3 = " + utils.c);

}

void Update () {

print(MyUtilities.GenerateRandom(0, 100));

}

1. Finally assign the script to an object in the scene (ie. Main Camera) and run the scene. You will see the output in the **Console** window.

The process is very straightforward and seemed to be easy enough to program around this in order to achieve what I wanted. I will explain later why this was not the optimal solution for what I was trying to achieve.

After a working environment was setup I did some research on autonomous behaviors and other peoples implementations. Every implementation was a non- runtime simulation which worried me about the limitations of my project. Even the tool that I had written was not capable of runtime interaction. It was very important for me to make this program useable at runtime so that users can interact and manipulate the flocking system. It would also require the algorithm to work exactly as it is described leaving very little to faking the outcome. "placeholder for more notes"

**Implementation Details**: The algorithm is described as follows. Three rules govern a system of boids. These rules are cohesion, separation, and alignment. Cohesion is the 'tendency' of a system to either stay tight together or form loosely. Separation is the 'tendency' of a system to keep a distance away from one another. Alignment is the 'tendency' of a system to match velocity with its neighbors. Notice the frequent use of the word tendency in all three rules. Everything tends to do something based on given conditions. This tendency is calculated with a summation of vectors. The first rule calculates the positions of boids and averages all of the positions to get a perceived center of mass. The second rule calculates a displacement vector. The displacement can be thought of as the overall change in position over a given time. It is related to distance in that if someone walks a path that covers 12 total units, if they end up where they started, their displacement will be zero. This displacement vector is what keeps the boids away from one another. A boid will do a check on all boids around it within a certain distance (radius) then calculate how far (displacement) it needs to be until it is out of the radius. The third rule is the velocity matching or alignment. This rule is similar to the first in that it grabs the average velocity of its neighbors instead of positions and uses those to align itself.

Even though the three rules govern the system, there are other factors that contribute to the overall behavior of boids. One of these being the boundary that keeps them from shooting off into infinity. Whenever a boid breaks the defined region, it will then add to its position a vector in the opposite direction. This solves problems dealing with any type of unrealistic movement. If I had just used a collision volume the boids would crash into the wall and it would not look very realistic at all.

At the start of the project I had problems with boids basically flying off into infinity and couldn't figure out why. Initially I thought there were a problem with my implementation of the algorithm which resulted any many hours of frustration. Only until the point that I limited the system to only react to the first and second rules was I able to see the dependency of speed across all three rules. I then set a maximum velocity the boids could travel and they began to behave correctly.

The design of the project is a loose representation of Model-View-Controller. The GUIController object holds the slider values and the BoidAlgorithm object gets the information from it. It then does its calculations based on what the values for the rules are set to be. Those values are coefficients that scale the intensity of the rules. Inside the editor these values can also be manually set. The boundary that holds the boids can also be manually set as well as the strength that they move to the target object (sphere).

With the original design being centered around Model-View-Controller, it seemed like building this project with an external library would have seemed feasible. However, as the project went on, the realization of building this library for use between Maya and Unity would not be possible. This is because of the nature in which the program works. It is meant to supply a runtime solution for boids behavior whereas Maya would require a preprocess solution. If I were to tackle this problem again I would have to take a completely different approach.

**Project** **Completion**: Upon completion of this project I ran into a couple of issues with the distribution of the project. Supporting multiple platforms such as Linux and Windows along with a Windows web build needed to be accommodated for. The end result being multiple builds targeted at each operating system. This project also encouraged me to get my own website and web hosting site up at www.chusoftware.com where I posted the web build at <http://chusoftware.com/unity/web.html> .

Personal projects have always been a hobby of mine but none have been as rewarding as implementing boids algorithm. After working with boids I have a much better understanding of motion and its implementation. I have also decided to keep this boids project ongoing as I am not satisfied with its current state. After testing users were very confused as to what the program was actually meant to do. From a game design stand point this is unacceptable. A large amount of time went into the planning and implementation of the algorithm but fell short because the presentation was lackluster. I plan on making some type of step by step sequence that will allow the user to walk through the algorithm. How I plan to do this is still undecided. I do not intend to take on any additional help as I feel that this project is not for distribution but as a learning tool. Thank you very much for allowing me to do this project under your tutelage.