Lab 1 Steering and Boids
Models and Simulation (DD1354)

January 15, 2019

1 Introduction

In this lab we will explore the concept of Boids, within the context of game physics and computer graphics. Boids is an algorithm, first developed in 1987 by Craig W. Reynolds[1], which attempts to simulate *flocking behaviour*. Each individual boid steers according to local rules reflecting "realistic" behavioural strategies such as fleeing, avoiding, grouping, separating etc.

1.1 General Particle Systems

Particle systems is a common term used to refer to techniques that use a large number of graphical objects, such as particles or sprites - in this respect a Boid Simulation is a form of particle system. The general concept is used for much more than flocking behaviour.

Typically, the implementation of particle systems have a special point in 3D space referred to as: emitter. This is the point where the particles spawn. The particle system has attributes that allow the creation of different effects. The particles are continuously generated, moved and eventually destroyed according to these parameters. The update loop of the particle system can be separated in two phases, simulation and rendering, which are described below:

1. *Simulation*: during this phase the new particles that ought to be created according to the system's parameters are placed at the emitter. For each of the particles in the simulation, a condition first checks if they have exceeded their lifetime (in which case they are removed from the simulation). Otherwise, the position, color and other attributes of the particles are updated according to the system's parameters. Some of the calculations that are usually taken into account are: collision of the particles with meshes in the environment, gravitational interactions, and how the velocity of the emitter affects the particles themselves.

2. *Rendering*: during this phase the position of the particles obtained in the previous step, along with the other attributes calculated are used to render the particles on the screen. Often, the particles are rendered as quadrilaterals that always face the viewer. This can be done in more complex ways depending on the hardware capabilities, e.g. using 3D meshes for each of the particles. Normally, due to the amount of particles involved, it is not feasible to use complex methods. Nonetheless, the results that can be achieved using the simple rendering have a high degree of realism.

Particle simulations come primarily in two flavors, dynamic and static. In the first scenario particles are rendered in real time as their position is updated due to the simulation step which alternates with the rendering step. In the second scenario particles are rendered after the simulation has been run completely. In

these cases the simulation phase merely stores the position of the particles in order to form trajectories which are later plotted in the rendering phase.

Examples of the dynamic scenario are particle systems used to simulate fire, smoke, water, etc. Examples of the static scenario are particle systems used to simulate hair, fur, grass, etc. The primary distinction between both scenarios comes from the presence of strands which can be simulated using the "static" trajectory of each particle for a given timestep.

2 Boids

2.1 Reynold's "Boids"

Boids originally corresponded to a bird-like object - a *bird-oid* object. The idea was that complexity arises from action of individual boids, taking into account their local surroundings and personal goals.

2.2 Motion model

The boids in this lab consists of a position, a velocity and an acceleration. As you recall from lab 1, you can iteratively integrate these variables and end up with the following equation of motion:

For the sake of simplicity we have ignored the mass of the boids, which is assumed to be 1 in all equations. There is also no gravity in this simulation, each fish is considered to be sentient and buoyant enough for gravity to be negligible.

2.3 Reynold's steering force

Reynolds introduced the idea of a *steering force*. The acceleration variable is used to accumulate all the different steering forces acting upon the boid during a simulation step and it is cleared before starting a new simulation step. This is very similar to the gravitational force in the lab 1, but here there will be many different and independent forces acting upon it, forming a more complex behavior.

The flocking behaivour forces are in focus in this lab, but some "extra" forces prove to be useful too:

1. Drag:

   A Drag is a force acting opposite to the velocity as a result of resistance in the medium e.g *air resistance* or *fluid resistance*.[2]

   Linear drag, or viscous resistance, is described by the following equation:

   $$F_d = -b\vec{v}$$

   where b is a contsant derived from the property of the fluid.

   This is actually a special case of a more general drag formula. If you are intrested in this you can read more about it on the wikipedia drag equation page.[3]

2. Bounds

   In a big ocean, using bounds as we do in this lab would not really make sense. But it is a extremely convenient way of keeping the boids in a defined area, close to the camera. It is implemented as a linear spring according to Hooke's law[4]:

   $$F = k * \vec{v}$$

where k is the stiffness constant and $\vec{v}$ is a vector from the boid to the wall it has passed through. This is only applied when the boid is actually outside of a wall.

3. Speed constraints

   To keep the simulation more stable and interesting we have added constraints for the maximum and minimum speed of the boids. These are not hard constraints and will work as soft linear springs when the speed is either too high or too low. This will give the effect of boids going slow accelerating up to the minimum speed and the boids going too fast decelerating to the maximum speed. These forces are given by:

$$F = k * (|v_{desired}| - |v|) * \hat{v}$$

   where v is the current velocity, $\hat{v}$ is the normalized velocity (the direction) and k is a force factor constant.

2.4 Flocking

Your tasks in this lab will revolve about implementing a flocking behavior in this simulation. The aim is make the fish swim in small groups, which can dynamically break off into smaller groups or merge into larger ones. This is done by implementing three different behaviors and combining them.

Each individual behavior will result in a force vector and can be combined by simply adding them together, along with the previous forces above. It is important to think of scale when doing this sorts of calculations, if one force is significantly higher than the other, it might become so dominant that the other forces become negligible. This is something to keep in mind while implementing these behaviors if you get unexpected results.

Each boid will have a set of neighbours, Adj, of size N. These are the other boids that are within a certain radius of the boid in question. The "pivot" boid is located at position $\vec{p}$ with velocity $\vec{v}$.

1. Separation

   The separation behavior will make the boids avoid each other if they get too close. For neighbouring positions p_i the separation force is:

$$\vec{F}_{separate} = k_{separate} \sum_{p_i \in Adj} \frac{r_{separete} - d}{d} (\vec{p_i} - \vec{p})$$

   where $k_{separate}$ is the separation force factor, $r_{separate}$ is the radius of the neighbor area and d is the distance between $\vec{p}$ and $\vec{p_i}$, $d = |\vec{p_i} - \vec{p}|$

2. Alignment

   The alignment behavior makes the boid want to steer in the same direction as its neighbors. It is calculated by taking the average velocity of all neighbors and taking the difference between the boid's velocity and the average velocity. When applying this force it will make the boid steer towards the average velocity.

$$\vec{v}_{average} = \frac{1}{n} \sum_{\vec{v}_i \in Adj} \vec{v}_i$$

   where $\vec{v}_i$ is the velocity of a neighbouring boid. This average velocity can then be used to calculate the alignment force:

$$F_{alignment} = k_{alignment}(\vec{v}_{average} - \vec{v})$$

3. Cohesion

   In contrast to the separation behavior, the cohesion behavior will make a boid want to steer towards the center of its neighbors. This might seem like a direct opposite to separation, but they are used in different ways. The cohesion force is often used with a greater radius, while the separation force usually has a smaller radius and a greater force.

   The cohesion force is calculated by taking the average position of all neighbors within the cohesion radius and applying a force towards that point, with the magnitude of the force being directly corresponding to the distance between the boid and the average point:

$$\vec{p}_{average} = \frac{1}{n} \sum_{\vec{p}_i \in Adj} \vec{p}_i$$

   This average position can then be used to calculate the cohesion force:

$$F_{cohesion} = k_{cohesion}(\vec{p}_{average} - \vec{p})$$

3 The Lab

3.1 The Unity setup

You will find a set up scene somewhere in the Unity project folder.

A clarification: **Changes made in runtime are not saved**. By changing parameters in runtime you can see what happens with the simulation, but remember that when you stop the game these changes revert back to what they were before.

3.1.1 The School Class

The School class represents a *school of fish*. It holds information about what behaviour the boids in the school has, how they are parametrised and also which classes and *prefabs* should be instantiated for that particular school.

3.1.2 The Boid Manager

This class handles the instantiation of all boids, and also provides functions to enumerate them. Each boid have access to the Boid Manager and can use it to iterate through the other boids and find its neighbours. The Boid Manager finds the Schools in the scene, tells them to instantiate their boids and then takes care of all boids i.e has a list of all boids in simulation.

3.1.3 SerializeField

The so called "Attribute" [SerializeField] can be found in most classes. Technically, this means that the field can be serialized using built-in serialization in Unity. What it implies it that the Unity Editor can change the field, while it remains private to other classes (C# convention is to never have publicly accessible fields, but expose them via properties);

3.1.4 The Boid Class

The Boids class provides the meat of the simulation. Each boid is instantiated by a School object and then handled by the Boid Manager. It contains fields necessary for our motion model: *Position*, *Velocity*, *Acceleration*.

The UpdateSimulation method, believe it or not, updates the simulation. It is called each timestep and calculated the new position of the boid. Provided in

the lab you will have code that handles the updating of the position, velocity and accelleration, as well as some behavioural code. These are the behavioural forces that *are already implemented*:

1. Separation: as described in section 1
2. Bounds: prevents boids from going outside of a bounding volume

Also, code for *drag* and *speed constraints* are provided.

3.1.5 The Fish Class

In Unity each gameobject will have transform containing the data which is needed to draw the object in the game world. What the Fish class does is to read of the boid simulation (A Boid Component attached to the same game object) and updates the transform accordingly.

3.2 Setup

Step 1: Install Unity

First you will have to install Unity. This is done directly via their webpage: `http://unity3d.com/unity/download`

Step 2: Download the Lab Unity Project

The project files for this section is available at DD1354 lab materials link *Lab 1 download*: `https://www.kth.se/social/course/DD1354/page/lab-materials/`.

3.3 Tasks

The first thing you will want to do is to setup the lab environment. This is explained in the Setup section (3.2)

These are the main tasks:

1. Implement separation

   This task is about you getting used to the code, since separation is already implemented.

   Look in *Boid.cs* and try to locate where the separation force is implemented. Compare this to the math for the separation force in the section above. Notice how the GetNeighbors works. (You don't need to understand the details of how the C# language works, just notice where and how you can access the neighbors of a boid). This is a good place to implement the following assignments.

   Notice that the force factor is read from the *School* class. Remember what was said about *[SerializeField]* and try to change different values in the editor while the program is running. You can do this by running the simulation, selecting the game object called "School of fish" and changing the values displayed in the *Inspector* view.

   What happens when you choose extreme values, such as a extremely great force factor, negative force factor or a huge radius?

2. Implement alignment

   Now it is time to do some coding.

   In *Boid.cs* find the variable called *alignmentForce*. It is currently set to zero and is never changed, but it is still added to the total force at the

end of the function. Your task is to implement the alignment behavior and make this force correct.

Remember the math from section 2. First calculate the average velocity of all neighbors within the alignment radius, then set the force according to the given equations. Make use of *School.AlignmentForceFactor* as the $k_{alignment}$ from the equation.

When this works the fish should start aligning with other nearby fish. Notice that they still don't keep together in a group, since we have not implemented the cohesion behavior yet.

Experiment with different values for the alignment force and radius. Make sure you have used *School.AlignmentForceFactor* and *School.AlignmentRadius* in your code, otherwise changes in the Unity editor will not be noticed. The default values should give reasonable results.

3. Implement cohesion

   Right now the fish try to swim in the same direction as their neighbors, but slowly drift apart. Now we will fix this.

   Recall the math from section 3. First calculate the average position of all your neighbors within the cohesion radius, then set the force according to the given equation. Just like last time, remember to use *School.CohesionRadius* and *School.CohesionForceFactor* accordingly.

   When done, the fish should group together and swim in groups. These groups form dynamically and is dependant on the parameters set in *School*. Experiment with different values and create some crazy behaviors.

4. Extend to 3D:

   A very (very) nice fact about boids in that the simulation works exacly the same in 3 dimentions as in 2. So far, the boids have been constrained to a 2D plane (xy-coordinates); the Fish class takes the updated 2D coordinate of the boid simulation and updates the x and y coordinates of the game object transform.

   Now, you should **extend this to 3 dimensions**. Locate all places where the boid simulation is constrained to 2 dimensions (hint: look for Vector2), and replace these with a 3D equivalent. The result is very rewarding!

3.4 Optional extra challenges

1. Use the Unity particle system:

   The boids simulation that you have worked with is a form of particle system. Take this opportunity and play around with the built-in Unity particle system! You can add things such as water effects behind swimming fish, "dust" particles in the water, clouds of sand when fishes swim near the ocean floor etc. Use your imagination!

2. Animate the Fishes: There are several way to do this. Explore your options!

3.5 Project ideas related to this lab

You might decide to extend this lab to a small project. Some ideas include:

1. Include Space Partitioning:

   Right now this simulation have a hard time handling more than 100 boids. This is mainly because the *GetNeighbors* implementation is naive. It goes through all the existing boids for each call. This is not very fast and have

a time complexity of $O(n)$, which will give a single step of the simulation the time complexity $O(n^2)$.

There are multiple different space partitioning techniques to solve this problem. One way to handle it is to have a grid of the same size as the simulation world and to add the boids to the correct cell after every update. The cost of maintaining the grid should be linear to the number of boids and the cost of looking up neighbors should be close to constant in the average case.

Your task is to implement a space partitioning technique of your choice. It should result speeding up the simulation. This code would most likely be implemented in *BoidManager*.

An easy way of keeping track of the performance is to press the *Stats* button in the *Game* view in Unity. This will show you different stats, frame rate being one of them.

When increasing the number of fish you will face another performance threshold as well, the rendering system. Right now the fish are rendered as individual objects and it can be a bit slow in large amounts. Fixing this is not part of this task, but can be done as an optional assignment using the Unity particle system.

What was your approach and which performance gains did you receive?

2. Implement Predator/Prey behaviour:

   In this task you will implement a predator / prey simulation. The goal is to have different type of fish, with one type of them trying to hunt the other type while they try to escape.

   To do this you need to create a new school of fish with another type of fish. This will require you to set a type to the fish and use this in the simulation to do different behaviors.

4 Resources

In the following section you will find links to useful Unity and Blender resources that will help you to get through this lab successfully.

`http://docs.unity3d.com/Manual/index.html`
`http://natureofcode.com/book/chapter-6-autonomous-agents/`

5 Troubleshooting

This lab was tested on a Windows installation of Unity 3D. The free version of the software is enough to complete the tutorial and requires only that you register the software with your email prior to running it. Unity is also available for Mac, but not for Linux.

References

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

[2] (2015). Drag (physics), [Online]. Available: `http://en.wikipedia.org/wiki/Drag_(physics)`.

[3] (2015). Drag equation, [Online]. Available: `http://en.wikipedia.org/wiki/Drag_equation`.

[4] (2015). Hooke's law, [Online]. Available: http://en.wikipedia.org/wiki/Hooke's_law.