Advanced programming classes #1

— IMAC 3rd year —

Boids insertion

Part 1/5 of « Boids Tower Defense »

Objectives:

- To be able to appropriate an existing project, under development, and especially:
  - To understand existing work
  - To start from what has been left
- To analyze a situation to be able to know when to reuse code
- To get into tools and methods of cross-platform compilation

Work to be done in two-persons teams, to be then sent to your teacher (neilb@free.fr).

“Boids – From *Wikipedia*, the free encyclopedia

Boids is an artificial life program, developed by Craig Reynolds in 1986, which simulates the flocking behaviour of birds. [...] The name "boid" corresponds to a shortened version of "bird-oid object", which refers to a bird-like object.

As with most artificial life simulations, Boids is an example of emergent behavior; that is, the complexity of Boids arises from the interaction of individual agents (the boids, in this case) adhering to a set of simple rules. The rules applied in the simplest Boids world are as follows:

- **separation**: steer to avoid crowding local flockmates
- **alignment**: steer towards the average heading of local flockmates
- **cohesion**: steer to move toward the average position of local flockmates

More complex rules can be added, such as **obstacle avoidance** and **goal seeking**. [...]
The boids framework is often used in computer graphics, providing realistic-looking representations of flocks of birds and other creatures, such as schools of fish or herds of animals. It was for instance used in the 1998 video game Half-Life for the flying bird-like creatures seen at the end of the game on Xen, named "boid" in the game files.”

Boids work in a manner similar to cellular automata, since each boid "acts" autonomously and references a neighborhood, as do cellular automata.

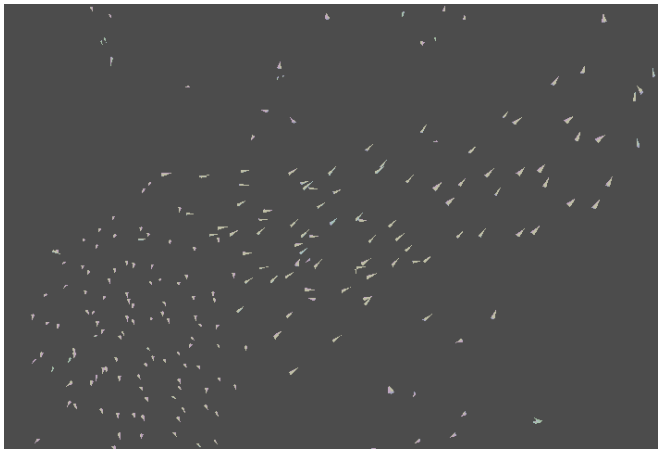


Illustration 1 : A boids simulation

Exercise 1 – Analysis phase

The project you're going to work on is a « tower defense », in which the player builds turrets to defend against ennemies. Our goal here is to insert « boids » as enemies in an existing code which basically displays a terrain, handles the inputs and the camera, etc.

Note: You won't implement a boids algorithm at all, since the code for it will be given after this exercise. However, you must discuss possible means to actually do that implementation, as if you were to do it.

To-do: First, explore the project. Then, read about constraints the (imaginary) client imposes about those "boids":

1. Boids don't fly. They move around a 3D terrain 3D (non flat).
2. Boids are attracted or "frightened" (= pushed away) by "targets" (or obstacles).
3. Boids are slowed down by the slopes of the terrain. They tr to avoid climbing difficult (very steep) slopes.
4. Boids can be often created and destroyed.
5. Boids don't really need an orientation. They can move in any direction and don't need braking / turning. Thus:
    - We can reduce their representation to a simple sphere (or even a point) for simulation.
    - "Steering for alignment" can be ignored.
    - Detection angel ("field of view") can be ignored.
6. Boids have (of course) a limited acceleration, as in the original algorithm.
7. Finally, code must be legible, maintainable and flexible. Indeed, there is a high probability that yet-unknown features will be added in the future.

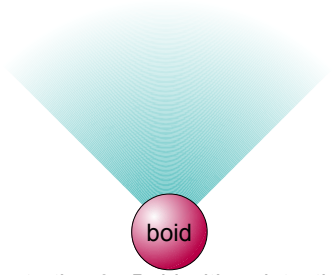


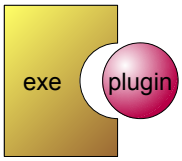
Illustration 2 : Boid with a detection angle

Some of those constraints get us away from the original boids algorithm. We must therefore think about how to implement our solution. Yet, two base ideas have been proposed:

1. Start from an existing boids algorithm, and add constraints. Terrain handling would thus be achieved using repulsive forces.
2. Start from a physics (Newtonian) simulation, on which we would add features to get the boids behavior by applying forces to objects.

To-do: Discuss **pros** and **cons** of those two ways to implement. Be relevant! Arguments must be thought and carefully proposed.

Exercise 2 – Integration



The game must be modular enough to replace the enemies behavior (here, the boids simulation) afterwards. This can clearly be useful, for instance, to create "mods" of the game without having to recompile the whole game[1]. A plugin was then the logical solution. The main program will send relevant data to the plugin, and use its simulation results to insert them int the visualization.

Objectives

Boids simulation is already coded and tested. But it was in a "traditional monolithic" executable. Migration to a plugin already started, but work is not done yet. You have to continue the task from where it has been left…

- First of all, learn how the main executable is compiled (here using "CMake", a cross-platform build system), and use it to compile.
  - You don't need to read all about the main executable code. This is too heavy.
- Read the coding rules.
- Read the simulation code (`Boids.h/.cpp`). See below for useful explanations.
- And finish the migration task!
  - Some "TODO TD1" in the code will help you locate where to add things).
  - Compile the simulation (`Boids.cpp`) as a plugin. Then make the executable work with it.
    - You're not required to use the same build system as the main executable. For this exercise, a Makefile or even a simple command line will be enough.

Plugin

What is a plugin?

- A plugin is a shared library (.so, .dll),
- which is loaded at run-time,
- and having an "interface" (= at least one entry point), such as functions known by the caller.

In our case, the plugin and the main executable share a common header (`Boids.h`). This eases the integration, by providing a simple way to make both parts "agree" on the interface. Please note that this is not mandatory to create a plugin.

Of course, if a plugin contains a class declaration that the main executable will use, and if that class must be defined (implemented) by the plugin, then the whole class (declaration + definition) can't be solely in the header – else the executable would already know everything about that class, and the plugin would become useless. For this kind of usage, we might define abstract classes in the common interface, and make the plugin implement concrete versions of them. The abstract class is then a "contract" that the executable "emits", and that the plugin must "agree on". This is the case in our project.

Most of the time, the entry point interface is a "pure C" function (not a C++ one), because compilers "decorate" C++ functions[2], and the C++ standard does not define the way to do it. Thus each compiler (and each compiler version!) has its own way. In "C", "`void f(int)`" will be referenced by the very straightforward "`f`" symbol. The caller (main executable) can then simply ask the operating system to find the "`f`" symbol (function) in the plugin, whereas with C++, it would probably be "`_Z1fi`", or "`?f@@YAXH@Z`", or anything else.

It is therefore often the case that "pure C" functions are used as an interface, and that functions allocate a concrete implementation of an abstract class. In our project:

- `BoidSim` is the base (abstract) class representing a boid from a simulation point of view (not in a graphical way).
  - Please note that the simulation code can handle both cases discussed in exercise 1 (`update()` will return either a position or a force, depending on the case).
- The executable will get back (concrete) instances of the `BoidSim` class (= derivates). Those are created by `BoidsUtil`.
- Here, the entry point is a "C" function, returning a `BoidsUtil` to the caller, so that it may create some

1 *In a "closed source" game (as often, actually), recompilation would imply that "modders" would have access to the game source code, which is obviously not an option.*

2 *Decoration is mandatory because you can overload in C++, whereas you can't in pure C. Indeed, having the un-decorated name "`f`", you cannot say if it refers to `f(int)` or `f(float)`.*

`BoidSim`. The expected function prototype is: `BoidsUtil & getBoidsUtil()`

- Plugin must be named "`btd_boids.so`" (Linux/MacOSX), "`btd_boids.dll`" (Windows, release), "`btd_boidsd.dll`" (Windows, debug) or "`btd_boids.sl`" (HP UX).
- If the executable-plugin seems incomplete or may be improved, tell your teacher.

Additional exercises

If you have enough time, you may improve your training by trying to reach some of the following secondary objectives. All are independent; you may choose the ones you want, in the order you want. Those are all modifications to the simulation; you will have to read and understand the plugin code.

Handling groups of boids

Add the ability to handle multiple groups of boids. Each group try to avoid other groups (« steering for separation ») but do not try to get close to other groups (« steering for cohesion »).

Steering impacted by health

Add health points management to the boids. “Wounded” boids have a modified algorithm. For instance:

- Maximum force applied to the boid is lower.
- The boid try to survive, by avoiding more “negative” targets.
- The boid has a shorter detection range (as if it was “seeing” less far away).
- The boid try to isolate from others (weaker « steering for cohesion » force).
- etc.

Other steering parameters

As for the steering impacted by health, propose and implement various parameters modifying the algorithm. Examples:

- A “frozen” state in which the boid cannot be applied forces during a given period.
- A “firing” state, which adds a recoil force to a weapon that the boid would carry.
- etc.

Detection optimization (difficult)

Boids do interact with their neighbors and targets, in a given detection radius. Trivially, the algorithm can make each boid check for all other boids and targets, compute the distance, and ignore those too far away. This solution is said “brute force” and is of course sub-optimal ($O(n^2)$).

Propose and implement an optimization, so that the algorithmic complexity is lower. Show your results with charts and graphs, about milliseconds or frame rate.