2/1/2014

Fast-forward your development.

ANT SIM STARTER KIT

Programming Guide

Published By:

4Cast

PREMIUM SCRIPTS FOR UNITY 3D

Introduction

Ant Starter Kit, introduced in Feb 2014, is a high-performance ant life simulation framework. It includes high-level functions for colony management, ant behavior, ant breeding, and a simplified ant DNA model that can be easily applied to your applications. You can use Ant Starter Kit to establish a foundation for building insect-related applications.

Ant Starter Kit optimizes your project by using a tried-and-true Model-View-Controller design pattern combined with simple and concise C# code that won't add bloat to your projects.

To understand the information in this document, you should be familiar with the C# programming language and Unity 3D v4 or greater.

Ant Starter Kit is the only starter kit of its kind — allowing you to start on a new project that will stand out from the rest. Ant Starter Kit is simple to understand and learn, yet the design supports an endless amount of games, applications, and simulations.

Who Should Read This Document?

This document is for developers who need to write Unity 3D-based programs that perform basic ant life

simulation. Ant Starter Kit is a **C# code framework,** and as such it is not intended for artists who want a zero-code solution for an ant game. This document expects that you already know how to code and build a Unity project, and you are looking for an insect subsystem. You should comfortable working in code. While it does include a 2D tile renderer to help you get started visualizing simulations, Ant Starter Kit's focus is on the challenges specific to building simulations of ants — not tile renderers. You can use other great graphics libraries (like NGUI or Tidy Tile Mapper) to creatively express the data you pull from Ant Starter Kit's bio-simulation engine.

Starter Kit is particularly suited for:

- Making games/applications about ants
- Making life cycle simulations
- Modeling combinations of traits/digital DNA
    - **Note:** Ant Starter Kit does *not* include a full genome for ant or any other species of animal. This would add a lot of bloat to the project unnecessarily. Instead, the included digital DNA model simply hosts a collection of traits (four out of the box) that can be combined with other ants during breeding, and then used to create offspring. The quantity of traits and how they're expressed is application-specific.

Organization of This Document

This document is organized into the following chapters:

- Overview of Ant Starter Kit
- Performing (X) Operations

Overview of Ant Starter Kit

Ant Starter Kit is a computational biology-inspired ant simulation framework. There are many starter kits on the Unity Asset Store already, but none of them focus on the low-level life and biology of insects, and our goal with this starter kit is to facilitate the next generation and ant simulation applications for mobile platforms.

When to use Ant Starter Kit

If simulate the lives of little ant-like creatures is a need for your application, you should use Ant Starter Kit (even if those "ants" are actually "people" in your game design!)

Aside from ant sims, Ant Sim Starter Kit is also well-suited for applications that require large/complex hierarchical animal kingdoms.

Generally speaking, Ant Starter Kit is a timesaver, but there are certain cases where the use of Ant Starter Kit would be impractical. For example, it would not be practical to use Ant Starter Kit to make a game about the life of a single ant. While Ant Starter Kit is perfectly capable of representing that single ant, its benefits lies in representing the genetic relationships between multiple ants. If you're not tracking a colony of ants, a character-driven starter kit might be a better choice.

Data Formats Available in Ant Starter Kit

Ant Starter Kit is designed to be as lean as possible. You will not find a lot of resource files scattered around the project. This keeps performance fast, and reduces clutter. More importantly — this means that whatever data storage techniques you're currently using for other parts of your application will have fews problem co-existing. This is a design philosophy embraced by all of 4Cast's software. We follow in the footsteps of companies such as Apple which believe the best code is the shortest & easiest to read code[1].

When possible, we use simple primitive data types to represent the simulation (e.g. two-dimensional array of integers used to represent the environment) to keep things simple [2]. By keeping things simple, you also can get significant performance boosts. For example, anytime you allocate memory, the operating system has to make a costly system call. This means something as innocent as calling "new YourDataStructureHere()" in a loop could have user experience-impacting consequences and therefore should be avoided when possible.

---

[1] This guideline has been confirmed by several 4Cast team members who worked at Apple, Inc. as software engineers 2007-2009.

[2] KISS. "Keep It Simple, Stupid". You think you know what this means now, but only after you've dug yourself into enough unnecessarily complex holes do you realize the true meaning of this statement. Experiment for yourself if you are skeptical, and/or heed our advice and KISS.

The core of Ant Starter Kit's functionality is the AntSimulation class found in AntSimulation.cs. As the name suggests, this class is responsible for performing the simulation. It also contains within it as private classes definitions for other helper classes for the simulation. Those helper classes include:

- Ant
- Colony
- DNA
- Egg

There are also several enums used to represent other attributes of these classes, and they include:

- Ant.Gender
- Ant.Stage
- Ant.Color

Model-View-Controller in Ant Starter Kit

There are many reasons why Model-View-Controller is useful software design pattern, however those reasons are outside the scope of this documentation. Instead we will focus on how MVC is used specifically in this code base.

Data starts with the model — the core storage/data representation of that object. The model is where all key information regarding an object is stored, regardless

of it's representation. Examples of models in this project include pretty much everything within `AntSimulation` including the inner classes. You'll notice there's not much with regard to user interface specifics, or how the world is rendered. The model just tracks objects for the core data, and leaves rendering up to whatever object is responsible for rendering.

Views, as the name suggest, represent the user interface and what is actually visible to the user. This is where you can substitute in your specific representation. These are usually Unity game objects that have some kind of rendering component attached. To give you a simple visual representation to begin with, we've provided you with a NGUI-based view hierarchy that uses sprites to display what's on screen. As we stressed earlier, this is NOT a UI project, so we will not go into much detail on how to create tile maps or how to render objects as this is left up to you. You can choose any visual representation you like just as long you can provide a way to interface it back to the core model, which brings us to view controllers.

Controllers are what handle communication between the model and the views. For each custom view you make (to represent your ants, for example), you will need to have the corresponding view controller which maintains a reference to both the view and the core model it is based on. We've provided an example of this in `AntViewController`, partially shown below:

```
public static AntViewController Create(AntSimulation.Ant a)
    {
        // This loads a default Ant view prefab
        GameObject view = (GameObject) Instantiate
(Resources.Load ("AntSimPrefabs/AntView"));
        AntViewController viewController =
view.GetComponent<AntViewController>();

        // Associates the core Ant model with this
ViewController
        viewController.model = a;
        viewController.sprite =
view.GetComponent<UISprite>();
        if(viewController.model.IsQueen())
        {
            // Replace the current sprite with the
appropriate name
            viewController.sprite.spriteName =
"QueenToWestFromEast";
        }
        viewController.sprite.MakePixelPerfect();
        view.transform.parent =
AntSimulation.singleton.colonyView.transform;
        view.transform.localPosition = startPos;
        view.transform.localScale = startScale;

        a.viewController = viewController;
        return viewController ;
    }
```

Creating Ants

There are two main ways to create ants — creating them programmatically, or breeding them.

Creating Ants Programmatically

To create an ant programmatically, simply call:

```
AntSimulation.Ant a = new AntSimulation.Ant();
```

This will create a random male, winged ant. This can also be done by passing `0` to the constructor (which it does by default).

To create a black queen ant programmatically, simply call:

```
AntSimulation.Ant queen = new AntSimulation.Ant(1);
```

This informs the constructor to create a Queen-style ant capable of reproduction. These ants are required if you wish to see ants created through the reproductive cycle.

Breeding Ants

To breed ants, you must first have a Queen ant that is impregnated with seed DNA (stored in the **`seedDna`** field in the **`AntSimulation.Ant`** class). The first Queen must be created programmatically (as shown above), and the default constructor should also provide starter

seed DNA (which is just a clone of the Queen's DNA). The `eggs` field for the ant will tell you how many eggs are left that can be laid.

After creating the Queen, she will navigate the colony, looking for places to lay eggs every so often[3]. Keep in mind the egg laying process begins first in the **model**, and is then later on represented in the **view**. For example, the AntSimulation.Colony maintains several arrays to keep track of what objects are inside the grid-based colony. The ant movement logic reads this array to determine if a move is valid or not (such as laying an egg). If there is room available, the Queen lays an egg, and an egg is added to the `eggLayer` array in the Colony class.

This is the general mechanism which governs how an already-impregnated Queen can lay eggs, but to impregnate a Queen in the first place she must perform what is known as the nuptial flight[4]. During the nuptial flight, the Queen will be exposed to a collection of fertile, male ants ("drones"). While swarming, several drones may attempt to mate with the Queen, but most of these attempts aren't successful. Once mating is successful, the Queen returns to her colony carrying the

---

[3] You have complete control over how frequently eggs are laid and can modify this in the `LayEgg()` method in the Ant class.

[4] Nuptial flight is an important phase in the reproduction of most ant, termite and some bee species.[1] During the flight, virgin queens mate with males and then land to start a new colony, or, in the case of honey bees, continue the succession of an existing hived colony. A good example is the nuptial flight undertaken by those ants of the genus Crematogaster.

new DNA which will be passed on to her offspring that will later emerge from eggs.

To trigger the nuptial flight, simply tap/click on a Queen and then tap the "Nuptial Flight" option.

FAQ

I'm confused. How do I get more help?

We're always happy to help you work through any difficulties you may encounter. Please email support@4cast.co with any questions.

What do I need to run this?

Unity 4.x, and NGUI (if you wish to use our visual examples).

How do I change the behavior of the ants?

Ant movement/behavior is performed by a non-deterministic algorithm which randomly choose from a collection of moves, and then sanity-checks it to see if that move is consistent with the current rules for your visual environment. The project includes a default/basic ant AI, but you will most likely need to adjust this to better suit the needs of your project.

To do so, simply modify these functions:

- `AntSimlation.Ant::ChooseRandomMove()`
  - Lists all moves out
  - Randomly chooses one of them

- `AntSimlation.Ant::DetermineNextMove()`
  - Takes the randomly chosen move and then calls `IsMoveConsistent()` to see if the move is acceptable or not
- `AntSimlation.Ant::IsMoveConsistent()`
  - Determines whether or the move is valid based upon information contained in the model and other application-specific criteria.
- `AntSimlation.Ant::Simulate()`

  - Moves that are consistent are added to a walk list for the ant, and then removed on each step of the simulation.