Flying Drone Toolkit for Unity Programming Notes

Clockworks Games

These Programming Notes are intended to supplement the User Manual for users that would like to extend the Flying Drone Toolkit by controlling drones through scripts or by adding additional behaviors or new 3D drone models.

Before attempting any of this, you should understand the Flying Drone Toolkit User Manual and Unity programming.

It is difficult to anticipate all questions that may arise if you try to extend the Toolkit at the programming level. These brief Programming Notes cover the main concepts. Please send questions to info@clockworks-games.com. We will answer as quickly as possible, and your questions, along with our answers, will be added to a future more extensive Programming Manual.

Script Control of Drones

Each Flying Drone should have the FlyingDroneScript.cs script component. This script includes the following public API for changing the mode of the drone:

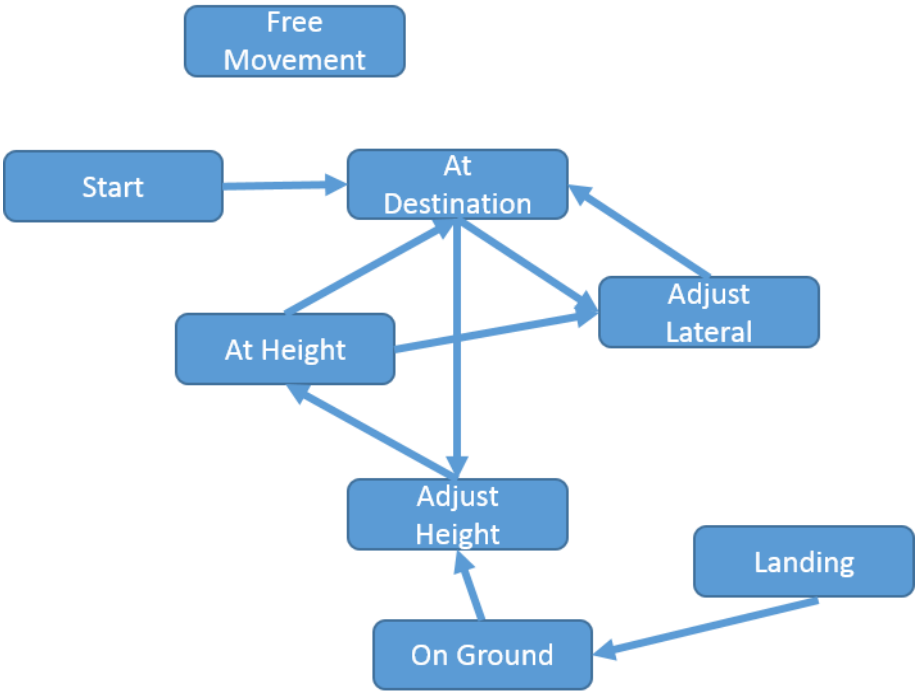
- public void SetMaintainMode()
- public void SetHoverMode()
- public void SetLandMode()
- public void SetManualMode()
- public void SetPatrolMode()
- public void SetFollowMode()

These are the behavior modes implemented in the first release of the Flying Drone Toolkit. Additional modes could potentially be added in the future. (Suggestions are welcome.) For a simple scripting example, please see SampleSceneGUI.cs, included with the Drone City sample scene.

Flying Drone Software Architecture

Finite State Machine

Finite state machines (FSMs) are a well-established way to manage complex but predictable behaviors. The Flying Drone Toolkit implements the state machine shown in this diagram:



All the drone modes (hover, patrol, follow, etc.) are mapped to this FSM. At any given time, a flying drone is in exactly one of these state. This state machine seems reasonable for the initial Toolkit, but there are some assumptions and limitations. For example, you may notice that a drone adjusts its location by first adjusting its height and then moving laterally. This behavior can be changed, but requires change at the code level to change the finite state machine itself.

There are conventions within the code for implementing the FSM, and these conventions should be followed if the FSM is changed. The implementation can be seen in detail in FlyingDroneScript.cs. In summary:

- Each state is a data type within the FSMDroneState enum structure.
- Each state is a case within FSMUpdate().
- Each state XXX has an EnterXXXState() function and an UpdateXXXState() function. The appropriate UpdateXXXState() function is called from FixedUpdate(), reflecting the current state.
- A state transition is performed by called EnterXXXState() from an UpdateXXXState() function.
- Each state has a case within the LeaveState() function.

Strategy Pattern

The software architecture for the flying drone makes use of the Strategy Pattern, which is an approach for flexibly allowing different behaviors and even changing the behavior at run-time. The Strategy

Pattern is documented, among other places, in the book *Design Patterns* by Gamma et. al. Wikipedia also has a reasonable summary at http://en.wikipedia.org/wiki/Strategy_pattern. To implement the Strategy Pattern, a programming language must have a way to store and reference different code to implement variations on a behavior. This can be achieved in Unity using Scriptable Objects. Unity Scriptable Objects are different from the more common Mono Behaviors in that Scriptable Objects do not need to be attached to Game Objects.

Motion Strategy

As the simplest example, we would like to give a seemingly random, drifting motion to the flying drone to simulate moving air currents.[1] There is an abstract Scriptable Object class called DroneMotionStrategy that defines a function called AddRandomMotion(). Three subclasses are provided: DroneMotionNone, which adds no random motion, DroneMotionSinusoidal, which adds a sinusoidal vertical motion, and DroneMotionSimple, which adds a simple linear up-down motion. The latter seems to work reasonably well; user-controllable parameters can adjust the magnitude and speed of the motion. If we wanted to add more complex motion to the flying drone, we could simply add and use a new subclass to DroneMotionStrategy (along with some setup code in FlyingDroneScript.cs); no other code would need to change.

Private variables within FlyingDroneScript.cs contain references to the classes that implement the motion strategy. The variable `motionBehavior` contains the one that is actively being used. The value can be changed dynamically at run time, to swap between the various kinds of motion.

A similar approach is taken for travel strategy (different types of lateral movement) and target strategy (how the drone chooses its next location target). The latter makes use of the concepts of "target" and "effective target." The effective target is the point that the drone is able to reach within its constraints, such as if the drone is constrained to travel only among waypoints.

Adding Additional 3D Drone Models

A flying drone can have any 3D structure and textures that you would like. You could create your own flying drone using a 3D modeling program, or you can acquire some type of flying vehicle model from the Unity Asset Store. To be compatible with the Flying Drone Toolkit, your new drone needs to have:

- A RigidBody component. Suggested settings are these, but you may want to experiment with these to find the best values for your own drone:
    - Mass, Drag, Angular Drag – all 1.
    - Use Gravity, Is Kinematic – unchecked
    - Interpolate – None
    - Collision Detection – Discrete
    - Constraints – Freeze X and Z rotations.

[1] An alternative would be to fully simulate the aerodynamic physics of the flying drone's flight. After some experimentation, it was decided this would be overkill for a flying drone game object, both because it is more computationally intense, and because this approach makes the drone quite a bit more difficult to control!

- Colliders to prevent your drone from passing through solid objects. These need to be tailored to the shape of your own flying drone, as appropriate.
- If you would like your drone to emanate a sound, add an Audio Source component and drag in a sound clip.
- Add the FlyingDroneScript. The parameters can be adjusted as described in the Flying Drone Toolkit User Manual.
- Subclass the FlyingDroneSpecificScript, and add this class to your drone. Note that there are several functions that you will need to provide, such as functions for determining whether the drone is on the ground, the distance from the ground, etc. You can refer to the provided DomeDroneScript.cs and FlyingSaucerScript.cs scripts for examples. You can also provide Start and Update functions for doing things specific for your own drone, such as spinning propellers.
- If you would like your drone to tilt, you can also add DroneTiltScript. This script requires that the drone has at least a two-level hierarchy in which the top level is use to control the drone position and yaw (y rotation), while the second level controls the pitch and roll (x and z rotations) of the drone, as shown for example in the Dome Drone object hierarchy below. The tilting occurs at the "assembly" level. This was done so that changes to pitch would not also change the altitude of the drone.