AI Fleet Commander GPL
BETA Reference Guide

Devon Ellis
devonellis@yahoo.com

Nils-Arne Dahlberg*
nilsarne@algonet.se

August 20, 2000

Abstract

What is Fleet Commander?
The vast expanses between worlds are patrolled exclusively by automated war machines. Recently, there have been a few skirmishes in some of the more important sectors. Your job: send a Centurion class Automated Assault Cruiser into the area to investigate. Should you encounter enemy forces well, you know what to do.

Programming games (such as Fleet Commander) require a little more effort to learn and become proficient in. It does not provide as much immediate satisfaction as some games, but the long-term enjoyment and satisfaction can be much greater. If you get stuck, don't give up. There is a community of players on the internet who are happy to assist.

Contents

*Conversion to LaTeX

1 The Universe

As a Fleet Commander, it is your job to construct a fleet. Carefully craft and design the stratagies and "artificial intelligence" to guide the ships in your fleet, then unleash them on the enemy.

The challange of Fleet Commander is the designing phase — creating and testing a fleet, tweaking and adjusting the stratagies, and continually adding newer and more sophisticated systems. Programming games are the ultimate strategy games. The strategy is designed and set before the battle even begins. After the battle begins, you as a user have no control over the outcome. Your fleet simply plays out your strategy as you designed it.

The great satisfaction in Fleet Commander comes when you watch your fleet do exactly what you told it to do, and the enemy is destroyed. The great satisfaction is knowing that your plan, your strategy, your programming was better than the enemy's.

The type of fleet you build is completely up to you. There are four distinct vehicles you can choose from, but what each ship really does is up to you. The fleets you design are limited only by your imagination.

1.1 The Warchief Fast Attack Fighter.

Figure 1: The Warchief Fast Attack Fighter

Individually, the Warchief is the weakest starship available. Although it is fast, it can only scan targets directly ahead of it, and can only fire unguided rockets. Against more advanced starships, a lone Warchief is nearly useless. The Warchief's strength is in its low cost and great numbers. One fighter will never destroy an alert cruiser, but no cruiser can stop a charge of twenty or thirty fighters. Fighters are the backbone of most (but not all) star fleets.

1.2 The Clawson Gunship.

The Gunship is a very versatile starship. It can be used in just about any role you can come up with, including heavy fighter, minelayer, escort, reconnais-

Figure 2: The Clawson Gunship

sance, and missile cruiser. It can take more hits than a fighter, can scan in any direction, and can fire the deadly Mark III Programmable Missiles. The Gunship is just waiting for you to invent a good application for it!

1.3 The Centurion Automated Assault Cruiser.

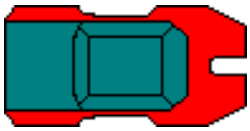

Figure 3: The Centurion Automated Assault Cruiser

The Assault Cruiser is a key element in every fleet. It is the only starship that can create other units — including more cruisers! It can scan long distances in any direction, letting the Centurion see most enemies coming before the enemies see it. Like all the starships in Fleet Commander, the roles you assign to your Assault Cruiser are entirely up to you. It can be used offesively or defesively. Good uses include long range reconnaissance, fighter interdiction, and blockade running. You can even park a few in tight formation as an ad-hoc starbase!

1.4 The Mark III Programmable Missiles.

Figure 4: The Mark III Programmable Missiles

The Gunship and the Centurion can both launch programmable missiles. These missile are starships in their own right, and follow your instructions and fill your role. Although they cannot scan, they stay locked onto their assigned targets until they either destroy them or run out of fuel. Programmable missiles are far more than just guided rockets. At your command, they can follow elaborate patterns, send information back to the fleet, or even stop dead in their tracks to become mines. Use their ECM hardware to jam enemy scanners, or even to paralyze an enemy altogether!

So you've seen what's available. Or at least, most of it. There are also Starbases, Freighters, and other types of vehicles (and creatures) you'll encounter later.

If you haven't done so already, run a few sample battles to get the idea. Go to the battle menu and choose "Select Fleets." Pick a few sample fleets, then

close the box. Go back to the battle menu and choose "Start Battle." Watch as each of the example fleets begins executing the instructions that were pre-set for them. Watch them launch fighters, maneuver, and fire missiles. Hold down the F1 key to see where each starship can scan. Watch what they do when their scanners detect enemies.

How do you create a fleet? You program it. It takes a bit of work, but it's not that hard. First, you'll need a brief overview of events and objects. If you are familiar with events and objects from other programming languages, read the following section anyway. Things work a little differently here. See the "Note to Experienced Programmers" later on.

2 Missions

Missions are the single-player version of Fleet Commander. As you complete the missions, you move forward in the game. Choose "Next Mission" from the Mission menu to start. It will automatically load the next mission. The mission briefing gives you information you need to design a winning fleet. Different missions may need different fleets.

The sample fleets may win the first mission or two, but very soon, you'll need to design your own fleets.

3 Designing Fleets:

3.1 Events

Fleet design in Fleet Commander is centered around events. An event is something that happens in the game that your starships can respond to. For example, your scanners may detect an enemy starship, or an enemy starship may detect you. Your starship may get hit by a missile. Many things in Fleet Commander trigger events, including "nothing." If nothing happens during a game cycle, an idle event occurs. Much of your game strategy is likely to occur while "nothing" else is happening.

Each ship gets its own events. For example, if your cruiser takes a point of damage during the same game cycle that your fighter scans an enemy, the cruiser will recieve a "damage" event and the fighter will recieve a "scan" event. If you have more than starship of the same type, the same thing applies: they each recieve their own events. If two events happen simultaneously for the same starship, the event with the highest priority occurs.

The following is the list of events in Fleet Commander, in order of priority:

`on_idle` This is the event that occurs when nothing else interesting is occurring for a particular starship. This is where missiles "guide" to their targets, where cruisers create new starships, and where your other ships carry out your overall strategy.

`on_init` Each time a new starships is created, an on_init even is triggered. Anything that you want to have happen before anything else happens should be put here.

`on_edge` Fleet Commander is played on 3600x2400 grid, no matter what resolution your computer is running in. When your starship reaches the edge of this grid, it automatically wraps around to the other side. If you want to do something when this occurs, put it here.

`on_damage` (Cruisers and Gunships only) Unfortunately, the universe is not a very safe place. There are enemies everywhere. If any of these enemies manage to hit you with a missile, you might want to do something about it. If you do, here's where you put your instructions.

`on_detect` (Cruisers, Fighters, and Gunships only) There is an obvious tactical advantage in seeing your enemy before your enemy sees you. Sometimes, however, through fortune or skill, the enemy sees you first. If they do, the on_detect event gives you a chance to react. On_detect is triggered any time your starship registers on an enemy's scanners.

`on_scan` (Cruisers, Fighters, and Gunships only) There is an obvious tactical advantage in finding the enemy before the enemy finds you. The only way to find the enemy is to scan them. When your scanners detect an enemy, this is the event that gets triggered. Put instructions here to launch missiles, perform evasive maneuvers, call for reinforcements, or anything else you think is appropriate. Once an enemy has been scanned, the scanners automatically track it, even if it goes out of range. If more than one starship is found inside a starship's target cone, the newest starship is targeted. Veteran starships are less likely to be detected than fresh starships.

`on_broadcast` (Cruisers, Fighters, and Gunships only) This is a user-defined event. Your starships can use this event to summon aide, order an attack, synchronize a maneuver, or anything else you dream of.

`on_jam` This event occurs any time your starship is jammed by an enemy missile. What you do here is completely up to you; but remember: your starship cannot scan or track enemies while being jammed.

Once an event occurs, you can either ignore it or respond to it. For example, you may not care whether or not a fighter crosses the edge of the screen, but you certainly care what happens when your cruiser scans an enemy! The first step in designing a fleet is deciding which events you want to respond to, and how you want to respond.

3.2 Methods

So now you know when things happen, but you're still no closer to designing a fleet, right? Take a look at this example:

```
\#name    "First Example"
\#author  "Fleet Commander"
```

```
\#version "1.0"

< anything between these brackets >
< is ignored by Fleet Commander.  >

cruiser.on_idle
{
  < This is the cruiser's idle event. This >
  < section gets executed when nothing of  >
  < interest happens to a cruiser          >

  launch.fighter  < this launches a fighter.>
}

fighter.on_init
{
  < A fighter executes this event when it   >
  < is first created. It only happens once. >

  < pick a random heading>
  nav.setheading(math.random(64))
}

fighter.on_scan
{
  < Enemy found! This event occurs when a    >
  < fighter detects an enemy. Fire a rocket! >

  launch.rocket
}
```

It won't win any battles, but it is simple enough for an adequate demonstration. The first three lines are header information-this is the information that shows up in the "Select Fleets" dialog box when you select your fleet. After this is a comment — anything inside the <> brackets is ignored by Fleet Commander. Fleet Commander also understands the C-type comment /* */ and C++ type comment //.

Below the comment are three separate event handlers, on_idle, on_init, and on_scan. In front of each of these is the type of starship for which the event is written. The on_idle is written for a cruiser, while the on_init is written for a fighter. You can have separate handlers for the same event, one for each starship type. After the name of the handler is a number of commands enclosed in curly brackets { }.

Inside each event handler (between the curly brackets) is an object call. A complete list of these can be found later. An object call is made up of an object and a method. For example, launch.fighter. Launch is the object and represents a whole group of functions (you can launch other things as well, such as launch.rocket or launch.fighter). Fighter is the method. You can call as many methods in your event handler as you wish, but making your event handler too long may make your starships less responsive. (See performance issues later).

In the above example, cruiser.on_idle calls launch.fighter. Assuming the cruiser has enough cash handy and isn't currently recharging, a new fighter will be created. (If the cruiser doesn't have enough cash, nothing happens). Since it is the last instruction in on_idle, when the launch.fighter is finished the starship waits for the next event to occur. The new fighter that was just created has it's own events, and receives the fighter.on_init event.

Remember that every starship handles its events completely independently, and runs simultaneously. If you have multiple fighters, for example, each one gets its own on_init, message when it is created, and on_idle, on_scan, etc. while it is active.

Fighter.on_init is a little more complicated. Its purpose is to pick a random heading for the newly created fighter. Many methods require parameters: additional information inside the parenthesis. For example, nav.setheading needs a new heading to steer towards. nav.setheading(0) sets a course directly east. nav.setheading(32) sets a heading directly west (there are 64 possible directions). In this case, the heading we are choosing is the result of yet another method, math.random(64), which selects a random number less than 64. If you put these two calls together, you get a new heading in a random direction.

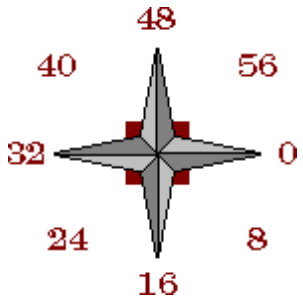

Figure 5: The compass that is in use

3.3 Math and Logic

All math, including comparisons, is handled by the math object. math.add(20, 3) would add 20 and 3, and give an answer of 23. Comparisons are handled in much the same way, except the "answers" are either 1 or 0 — true or false. math.greater(20, 3) would give an answer of 1, or true. math.less(20, 3) would give an answer of 0, or false. A more complete description of the math object can be found later.

Logic and strategy go hand-in-hand in Fleet Commander. Nearly all event handlers will need to make some sort of decisions, and the way to do that in Fleet Commander is with the 'if' statement:

```
cruiser.on_detect
{
  <event occurs when the cruiser is scanned. >

  <takes at least 350 credits to launch a gunship.>
```

```
  if (math.greater(status.balance, 350)) {
    launch.gunship
  }

  <keep rotating scanner>
  scanner.turnright
}
```

This is a handler for the on_detect event. In the above example, the cruiser reacts to being scanned by trying to launch a Gunship. Notice the 'if' in the above example. After the word 'if' is an expression in parenthesis. This is usually a math or logic function, but doesn't have to be. If the result is non-zero (not false), the block of commands after the if statement is executed. Otherwise, it is skipped. After the expression is a curly bracket, '{'. This marks the beginning of a block of commands that may or may not happen. The end of the block is marked with a close curly bracket, '}'.

In this case, the cruiser first compares the cash in the fleet's account (status.balance) with 350. If this comparison is true, the block of code afterwards is executed — a new Gunship is launched. If the comparison is false, the block is skipped. In both cases, the cruiser then calls scanner.turnright to change its scan direction.

The other important logic statement is the else. The 'else' is what happens if the 'if' condition is false. For example:

```
cruiser.on_detect
{
  <event occurs when the cruiser is scanned. >

  <takes at least 350 credits to launch a gunship.>
  if (math.greater(status.balance, 350)) {
    launch.gunship
  }
  else {       <--- modification here >
    launch.fighter()
  }

  <keep rotating scanner>
  scanner.turnright
}
```

The above example is identical to the previous one, except for the addition of an 'else.' Now, if the cruiser doesn't have enough credits (math.greater is false), the block of commands after the 'else' is executed — the cruiser launches a fighter.

3.4 Variables

A lot of useful information is already tracked and stored for you by Fleet Commander. Information such as the current heading, the angle of the scanner, and so forth, are automatically maintained. If you want to keep information of your own, use variables.

Variables in Fleet Commander come in two forms: private and shared. Shared variables are shared among the entire fleet. Private variables are unique for each individual starship-each starship can store different values in the variable.

To create a variable, put shared or private before your first event handler, followed by the name you want to use for that variable. For example, shared direction, or private last_x. Variables, like everything else in Fleet Commander, are objects. To set the value of a variable, use the method set. For example, direction.set(16) will set the value of the variable "direction" to 16. To use the value of a variable, use the object method get. For example, nav.setheading(direction.get()) sets the starship's heading to the value of direction.

Most variables only store a single value, but variables can store up to 512 separate values if needed. To create an array of values, use the method "size." Using the above example, direction.size(32) sizes status for 32 values. You can then use the method status.getel(10) to retrieve the value of the tenth element, or status.setel(10, 3) to set the value of the tenth element to three. Resizing a variable erases whatever was there before.

Arrays are numbered arbitrarily. Asking for a size of 32 for your array, the numbers can be referred to as elements 0 to 31 (as in languages like C++) or 1 to 32, or even 101 to 132.

3.5 Credits and Energy

Two resources are required to launch missiles, rockets, and starships: credits and energy. Credits are shared among all the starships of the fleet; energy is unique to each starship. When a fighter launches a missile, it drains ten credits from the fleet account, and requires 48 clock cycles (about half a game cycle) to recharge. The complete table is listed below:

Type	Credits	Energy	
Rocket	1	48	*
Missile	10	48	*
Fighter	100	55	
Gunship	300	100	
Cruiser	1000	300	

Note the asterisks. Cruisers recharge from rockets and missiles three times faster than fighters or gunships — meaning a cruiser can fire an additional missile in only sixteen clock cycles. The status.recharge method can be used to determine whether or not a starship is fully charged and can launch again. The status.balance method can be used to determine how much money is available.

3.6 Debugging

In programming terms, a 'bug' is an error in the programming logic. If a starship doesn't do what you expect it to do, you have a bug. Careful testing is important before you use a new fleet extensively, and before you send your fleet to any internet-based Fleet Commander tournaments. Fleet Commander offers several

special keys to make debugging easier. They may also make the battle more interesting to watch.

Hold down the debug keys F1 through F5 to get a glimpse of what is really going on. Note that these keys only work if the .FC fleet file is available.

F1: Scan Cone. This key displays the scanning cone for each starship. Notice that cruisers can scan in any direction, and scan much further than fighters.

F2: Track targets. Starships in Fleet Commander automatically track the last enemy they have scanned, even if that enemy is outside of scan cone (or destroyed). F2 draws a line between starships and their targets, making it easier to determine who is following what.

F3: Damage. Use the F3 key to view the amount of damage inflicted on a Cruiser or Gunship. The number in red is the number of missiles a starship can take before being destroyed.

F4: Recharge. Use the F4 key to view the amount of time before a starship will be able to launch a new missile/starship, or the amount of time until a missile or rocket self-destructs.

F5: Launch Code. Use the F5 key to view the value assigned to a starship at launch time. These values are usually used to assign special tasks or missions. See the object reference for more information.

3.7 Performance Issues

Each starship is given a specific and finite amount of processor time. Currently, this is enough for twelve instructions, or roughly six object calls. This number may change. If you have more code than can be executed in allotted time, the game engine continues executing where it left off the cycle before.

This means that a very long or complicated event handler may require several cycles to execute. During this time, other events are not processed. When the complicated event handler is finished, the pending event with the highest priority is processed. The other events are ignored.

3.8 Note to Experienced Programmers

Fleet Commander was designed as a game for all experience levels, from complete novice to advanced professionals. It was also designed as a teaching aid for C++ or Java programming. Because of this, the language used in Fleet Commander is very similar to the languages C++ and Java.

Programmers with experience in C++ or Java will have an advantage over those who have not used those languages, initailly. This advantage is brief and only applies during the first stages of the learning process. Once a novice has mastered the first few missions and understands the Fleet Commander language, he/she should be able to compete on fair terms with even the most advanced "real" programmers.

As an aid for those who do know C++ or Java, the following options exist:

Semicolons: In C++, all statements must end in semi-colons. C++ programmers are used to putting semi-colons everywhere. Fleet Commander ignores semi-colons, but does not give an error. Feel free to use them if you want your source to look more like C++.

Parenthesis: Functions that require no parameters do not require parenthesis in Fleet Commander. You may add an empty set of parenthesis if you like, such as status.getx();

Case Sensitivity: Unlike C++ and Java, case does not matter in Fleet Commander.

If you are using Fleet Commander as an aid in learning C++ or Java, it is recommended that you use empty parenthesis and semicolons.

3.9 Predefined Object Quick Reference

The objects and their methods are listed below. Methods that take parameters are indicated. For more detailed information and examples, see the Detailed Object Reference (seperate).

3.9.1 status

`status.x`	Current X coordinate (0-3600).
`status.y`	Current Y coordinate (0-2400).
`status.heading`	Current heading (0-63).
`status.color`	Team ID (0-5).
`status.life`	Points of damage left before starship is destroyed. (Always 1 for fighters). This value can be displayed on the screen using the F3 debugging key.
`status.code`	Parameter from the launch method, if used. This value is set by the launching starship, and is intended for assigning missions to craft. This value can be displayed on the screen using the F5 debugging key. This value is undefined if no launch code was used. If no launch code was used, this value is undefined (it could be anything). This method can be used to create different "styles" of starship. For example, you may choose to use fighters with a launch code of zero as scouts, while fighters with a launch code of one remain near the cruiser. You could also pass other information such as a starting heading, or use this launch code to break a fleet up into wings.
`status.SetCode(C)`	Writes C to status.code.
`status.recharge`	Number of clock cycles until starship can launch again. For missiles, indicates the number of clock cycles until detonation. This value can be displayed on the screen using the F4 debugging key.
`status.balance`	Returns the total credits available to the fleet.

3.9.2 target

`target.life`	Points of damage left before last scanned target is destroyed, or zero if target is destroyed, invalid, or cleared. The F2 debugging key can be used to view the current target.
`target.x`	X coordinate of last scanned target.
`target.y`	Y coordinate of last scanned target.
`target.heading`	heading of last scanned target.
`target.color`	Team ID of last scanned target (0-5).
`target.type`	The type of the target. 1 for Fighter, 2 for Gunship, 3 for Cruiser.
`target.clear`	Clears the current target.

3.9.3 nav

`nav.setheading(H)`	Sets the desired heading for a starship to any specified direction. Actual turning takes place later.
`nav.turnright`	Sets the desired heading one mark to the right of the current heading. Actual turning takes place later.
`nav.turnleft`	Sets the desired heading one mark to the left of the current heading. Actual turning takes place later.
`nav.stop`	Stops the starship. Not valid for fighters. Converts missiles into mines.
`nav.start`	Starts the starship moving forward. Not valid for fighters or missiles.

3.9.4 launch

`launch.rocket`	Launches a rocket. Not valid for missiles.
`launch.missile(C)`	Launches a missile. Not valid for fighters or missiles. Missiles are automatically assigned any target the launcher is tracking. Add a launch code if desired. See status.code for more information.
`launch.fighter(C)`	Creates a fighter. Only valid for cruisers. Add a launch code if desired. See status.code for more information.
`launch.gunship(C)`	Creates a gunship. Only valid for cruisers. Add a launch code if desired . See status.code for more information.
`launch.cruiser(C)`	Creates a cruiser. Only valid for cruisers. Add a launch code if desired. See status.code for more information.

3.9.5 do

`do.jam`	Only valid for missiles. Temporarily prevents any target from scanning or tracking enemies. This can completely paralyze a target, preventing any event handlers other than on_jam from executing. To be effective, this must be called repeatedly and frequently. Missiles that use this method do not destruct when they hit their targets. Only valid for missiles.

`do.broadcast` Sends an on_broadcast event to all starships in the fleet.

3.9.6 scanner

`scanner.heading`	Only valid for cruisers and gunships. retrieves the current orientation of the scanner, relative to the starship's heading. A scanner heading of zero indicates the scanner is pointed directly forward. The debugging key F1 can be used to display the current scanning cone.
`scanner.setheading(H)`	Only valid for cruisers and gunships. Sets the relative orientation of the scanner to any specified direction.
`scanner.turnright`	Only valid for cruisers and gunships. Moves the relative orientation of the scanner to the right.
`scanner.turnleft`	Only valid for cruisers and gunships. Moves the relative orientation of the scanner to the left.

3.9.7 math

`math.add(A, B)`	Adds two numbers.
`math.sub(A, B)`	Subtracts two numbers.
`math.mult(A, B)`	Multiplies two numbers.
`math.divide(A, B)`	Divides two numbers.
`math.mod(A, B)`	Returns the remainder of A/B. Useful for forcing A into the range zero to B. Ex: mod(124, 64) computes a number between 0 and 63.
`math.not(A)`	Returns 1 if A is 0, otherwise returns 0.
`math.greater(A, B)`	Returns 1 if A is greater than B, otherwise returns 0.
`math.less(A, B)`	Returns 1 if A is less than B, otherwise returns 0.
`math.equal(A, B)`	Returns 1 if A is equal to B, otherwise returns 0.
`math.random(R)`	Returns a random number less than R, including zero.
`math.turn_xy(X,Y)`	Calculates the best turn direction to reach point X, Y: positive for right turn, negative for left turns, zero for directly ahead or behind.
`math.turn_targ`	Calculates the best turn direction to reach the target: positive for right turn, negative for left turns, zero for directly ahead or behind.

3.9.8 clock

`clock.time`	Returns the value of the starship's internal clock. Initially, this is the number of clock cycles since the starship was created.
`clock.reset`	Resets the starship's internal clock.
`clock.fleettime`	Returns the elapsed clock cycles since the fleet was created.

4 Finally...

Remember that this is a Beta-and the very first beta at that. Please give us feedback about the game, the documentation, the idea, etc. If you see something you don't like, tell us! We can't fix it if we don't know. The best way to reach us is at http://fcomm.sourceforge.net

4.1 So, who are "we"?

(Or the Fcomm team.)
lodsw
Team leader, lead programmer
calash
Real name - Jason

Send the money to P.O. box :)

As for my role here.... Lurker :).
Mission Files
Testing
Mission development (Once i understand the language a bit more)
End user support (as needed)

I am avaialble via ICQ and MSN Messanger.

ojones

folk
Installation task
transients

Nilsarne
Documentation