Launch MFD, version 1.6.2
Orbiter module
http://sf.net/projects/enjomitchsorbit

Copyright © 2007	"Vanguard"	initial MFD author
Copyright © 2007-2014	Szymon Ender "Enjo"	author and maintainer ender-sz att go2 dot pl
Copyright © 2007	Pawel Stiasny "She'da'Lier"	contributor
Copyright © 2008	Mohd Ali "Computerex"	contributor
Copyright © 2008	Chris Jeppesen "Kwan"	PEG code
Copyright © 2008	Steve Arch "agentgonzo"	co-developer
Copyright © 2009	CJ Plooy & Tim Blaxland	KOST library

Table of Contents

1) Overview

Launch MFD calculates the appropriate launch azimuth, insertion profile (SSTO only) and time to launch and guide a spaceship to intercept a satellite or a specific orbit, for example described as parking orbit before ejection in TransX. Many physical influences are taken into account to provide you with a good accuracy of the calculations. After the launch, the MFD shows you an orientation of ship, that is needed to reach the described orbit. The orientation is adaptive in regard to the current situation in means of position and velocity. You can also use an included fuzzy logic autopilot, which follows the provided orientation smoothly (although blindly). All those features may be used simultaneously and independently for each ship in the simulation.

There's also an extra feature in beta stage of development: Direct Ascent algorithm, that, given some simplifying assumptions, will allow the pilot to launch and guide a spaceship into an immediate satellite interception course.
The Direct Ascent feature is described in section 7)

2) Installation

Unpack in your Orbiter folder, using folder names. Then, enable Modules tab in Orbiter's Launchpad and activate LaunchMFD
Alternatively, you can use this program: http://www.orbithangar.com/searchid.php?ID=2701
For Launch MFD to work, you need to install VC++ 2005 Redistributable Package if you haven't done so already.

3) Compatibility

Works with Orbiter 2010 **only**. It **will not work** with any previous versions of Orbiter. Special care has been undertaken for the MFD to work with future releases of Orbiter, by constantly updating the code to match with the recent Orbiter API, leaving out the deprecated functions.

It's also possible to use the 2006 version of the module with Orbiter 2010 and DX9 client with GDI compatibility enabled, which provides HUD drawing feature for older vessels.

Additional SDKs used:
HUDDrawer (required), ModuleMessaging (required) and OrbiterSound (optional)

4) Usage

Key summary

Shift + = Increase inclination.
Shift - = Decrease inclination.
Shift] = Increase inclination adjustment factor.
Shift [= Decrease inclination adjustment factor.
Shift T = Set target **or** inclination (equ. frame) **or** inclination LAN (ecl. frame).
Shift A = Set target altitude for PEG algorithm, or set final distance for DA program
Shift H = Switch HUD display
Shift M = Switch modes. Currently - Standard, Launch Compass, and Direct Ascent.
Shift D = Default action of the selected mode
Shift H = switch HUD display
Shift S = Change Error notification sound, or disable it completely (3 states).
Shift O = Switch off-plane correction
Shift P = Switch PID autopilot
Shift G = Switch drawing of great circles
Shift K = Switch track mode of great circles
Shift Z = Zoom in great circles
Shift X = Zoom out great circles
Shift C = Increase precision of great circles
Shift V = Decrease precision of great circles
Shift I = Switch pitch guidance
Shift E = Switch buttons page

HOT! Target inclination

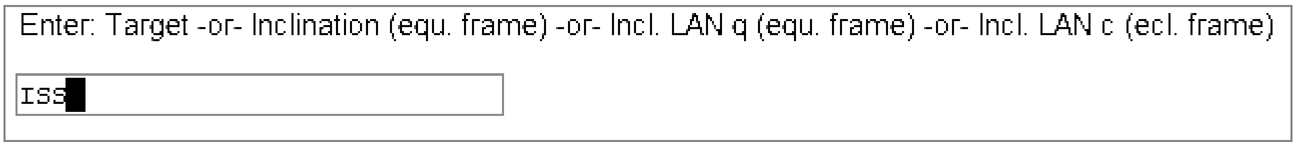
Set target's inclination by either using the **TGT** button (or **(Shift T)** combo) or use manual inclination selection:

Shift + = Increase Inclination.
Shift - = Decrease Inclination.
Shift] = Increase inclination adjustment factor.
Shift [= Decrease inclination adjustment factor.

You can use left mouse click on an MFD button responsible for increasing value to increase it, and you can click right mouse button on the same MFD button to decrease that value. The contrary works for a button responsible for decreasing a value. Notice that unfortunately this feature works only in the Glass Cockpit mode (when only MFDs are visible). This feature has been introduced to make you, the user, got used to it, because we'll have to get rid of some buttons to make some place for others eventually.

Shift T = Select target **or** inclination (equ. frame) **or** inclination LAN (ecl. frame).

If you use the TGT button or (Shift T) combo, then this input box will appear:


Enter: Target -or- Inclination (equ. frame) -or- Incl. LAN q (equ. frame) -or- Incl. LAN c (ecl. frame)

ISS

You have the following options here:

a) a target string, which can be either an artificial or natural satellite, eg. **ISS** or **Moon**, which don't need to orbit your source body at all. The target's inclination is then read and set as your target inclination - useful for intercepting that satellite.

HOT! b) “**TransX**” or “**TX**” (case insensitive), which results in obtaining the TransX' Escape Plan's settings. See below. Minimum TransX' version 2014.01.11 is required.

b) an arbitrary inclination in the equatorial frame - useful for deploying satellites at a certain inclination. The equatorial frame is used here because satellites inclinations are usually described in this frame.

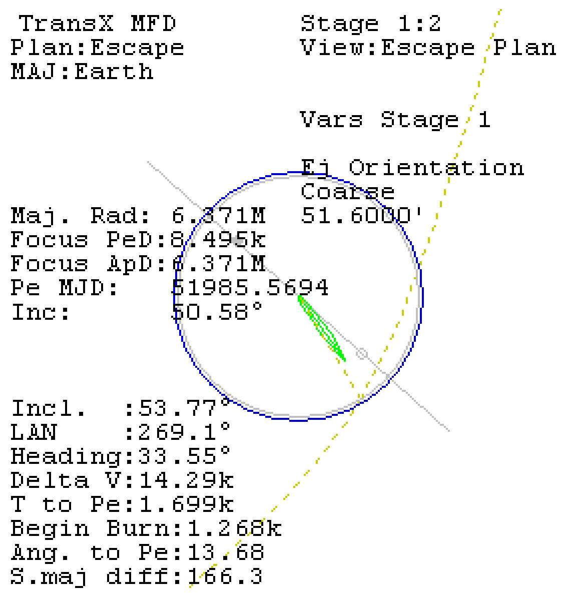
c) an arbitrary inclination and LAN in the e**q**uatorial frame, designated by additional '**q**' character – same as above but additionally gives more constraints on the orbit's orientation

d) an arbitrary inclination and LAN in the e**c**liptic frame, designated by additional '**c**' char. It is useful if you want to enter an orbit with a certain inclination, and a certain orientation, defined by LAN.

Whenever you enter a combination of inclination with LAN, as in this case (also applies to “TransX”), be aware that due to non-spherical gravity sources (Orbiter's option), the targeted dummy vessel's LAN will drift, so be sure to target the same orbital parameters again a few hours before the launch to update the dummy's position.

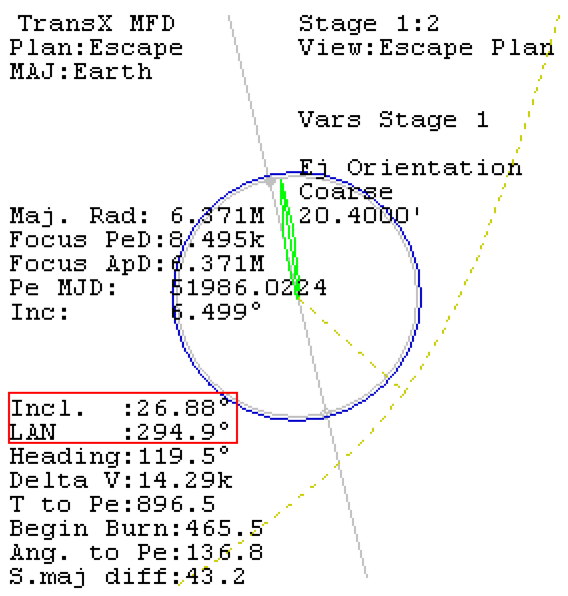
Ad. b)
TransX' outputs the necessary inclination and LAN in its first stage, displayed after setting an Ejection plan and picking a target, then setting up the transfer (velocities and date, etc.), and after that, returning to the first stage.

The proper incl. & LAN values are displayed by TransX after correctly adjusting the Ej. Orientation. The grey line, whose position depends on Ej. Orientation, must be placed so that it is some length before your position, **when your imaginary apoapsis touches the surface of the planet** in this projection. You will need to accelerate time to find out when the apoapsis is in place. The redundant distance between your position and the grey line defines your radial distance to the target orbit, thus the Time to Intersection. If you read on, you'll understand why the Time to Intersection must be greater than 0.



```
 TransX MFD            Stage 1:2
Plan:Escape            View:Escape Plan
MAJ:Earth

                       Vars Stage 1

                       Ej Orientation
                       Coarse
Maj. Rad: 6.371M       51.6000'
Focus PeD:8.495k
Focus ApD:6.371M
Pe MJD:    51985.5694
Inc:       50.58°



Incl.   :53.77°
LAN     :269.1°
Heading:33.55°
Delta V:14.29k
T to Pe:1.699k
Begin Burn:1.268k
Ang. to Pe:13.68
S.maj diff:166.3
```

Incorrect. *The grey line is before our position, but our apoapsis isn't touching the surface*



```
 TransX MFD            Stage 1:2
Plan:Escape            View:Escape Plan
MAJ:Earth

                       Vars Stage 1

                       Ej Orientation
                       Coarse
Maj. Rad: 6.371M       20.4000'
Focus PeD:8.495k
Focus ApD:6.371M
Pe MJD:    51986.0224
Inc:       6.499°



Incl.   :26.88°
LAN     :294.9°
Heading:119.5°
Delta V:14.29k
T to Pe:896.5
Begin Burn:465.5
Ang. to Pe:136.8
S.maj diff:43.2
```

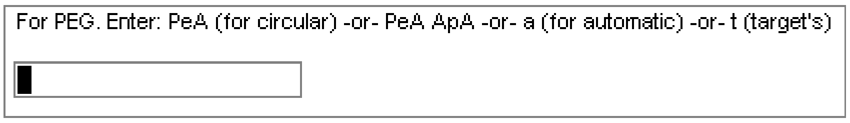
Correct. *The grey line is slightly before our position and our apoapsis is touching the surface*

You'll find very valuable TransX tutorials on Flytandem's web page:
http://www.flytandem.com/orbiter/

HOT! You can't launch a spacecraft into an inclination lower than your latitude (in equatorial frame), for example to Moon or Mir from Cape Canaveral. If you intend to intercept these targets anyway, you need to launch into inclination same as your latitude, by launching in a purely eastern direction. After reaching the orbit, you need to perform a plane change manouevre. It will cost less fuel when you launch in the correct time. Launch MFD supports such launches, by using a dummy, which has the same LAN as the target, whose name needs to be entered as usual. Note however, that such launches are limited only to the second possible heading, in case of Cape – southern, because launching into the northern one would increase your latitude to values exceeding the dummy's inclination, which confuses the MFD.
In case of selecting a celestial body as your target, you will also have to target it again a few hours before launch, because the dummy may be subject to LAN drift.
If you want to launch a winged spacecraft, like DG, instead of launching east, using a dummy, it will surely be more efficient to launch southwards and cruise in atmosphere for as long as your latitude is greater than target's inclination, and then, still in atmosphere make a turn towards the target orbit's plane to create am optimal intersection, effectively enabling Off-Plane correction. The turn will be more efficient than an in-orbit plane change, because of much smaller velocity which you need in the atmosphere, and because the velocity vector will be changed by your wings' lift, not engine power. It will make even more sense, if you use DG-S, which is equipped with SCRAM engines.

HOT! Target altitude selection for PEG

First, please note that the user interaction with old Reference Altitude is now discontinued. Modifying it made hardly any difference anyway. The new target altitude is for PEG algorithm only. When you set a target in the previous input box, both the PeA and ApA are automatically set to 20 km above the atmosphere or 50 km above surface of planets without the atmosphere. If this doesn't suit you, then after pressing **(Shift A)** or **ALT** button, this input box will appear, allowing you to change the target alt:


For PEG. Enter: PeA (for circular) -or- PeA ApA -or- a (for automatic) -or- t (target's)

You can enter for example:

250
for PeA = ApA = 250 km (a circular orbit), or
230 260
or an equivalence:
260 230
for an elliptic 230x260 km orbit, or:
a
for an automatic altitude selection. This means a circular orbit 20 km above planet's atmosphere or 50 km above surface for bodies without the atmosphere, or:
t
for setting the altitude to target's (PeA + ApA) / 2, ie. target's mean altitude.
TransX
or an equivalence:
TX
for obtaining the altitude from TransX' Escape Plan

Vessel variables

Launch MFD sometimes relies on variables stored in text files, under **Config\MFD\LaunchMFD\** directory. Most notable examples of such variables are PID autopilot settings and pitch programmes. The following strategy is applied for storing and reading the variables: variables stored during simulation are automatically exported to *VesselClass_variables.cfg* upon simulation exit. Some of these variables are read when you restart the simulation, but some are not, like the mentioned PID AP settings and pitch programmes. If you want to make these settings persistent, you have to manually move them from *VesselClass_variables.cfg* to *VesselClass.cfg* file. Then, the variables' duplicates in the *VesselClass_variables.cfg* will be ignored and the module will never overwrite your persisted settings in *VesselClass.cfg*. Both config files and subdirectories in which they exist are created automatically.

PID autopilot

The Fuzzy Logic based autopilot has been replaced by a PID based autopilot, due to poor quality of the former. Except for better quality (up to 100x time acc. in space) and more configurability, nothing has changed.

After the AP is enabled, with **(Shift P)**, or **AP** button , it will try to bank the ship so that the flight director is on top, which lets the ship use its aerodynamic surfaces better to orient itself to the marker. You may soon notice that the flight director's pitch increases greatly, especially if you use complex flight model. This is the PEG's problem and you should take this into account by not trusting the PEG and thus the AP too much. The solution is roughly pitching up manually before enabling the AP to get a proper readout first. There's also a moment when the AP switches to space operation mode, where there's obviously little or no drag. You will notice this when the AP suddenly “jumps” to a bit different pitch. After reaching orbit, the AP will deactivate itself automatically.

If you disable the autopilot manually, it will reset all the control surfaces and enable killrot autopilot. Be sure to disable it later.

There's a possibility to use the AP only for azimuth corrections, for example when you use a **SCRAM** powered ship (DG-S, XR ships), and so you want to be able to control the pitch by yourself. You can do it by pressing **(Shift I)** or **PTC** button, so that the pitch readout and guidance are disabled, and then by enabling the AP. After you reach nearly orbital velocity, I advice **against** enabling the PEG with **PTC** button, because it will provide erratic values. Instead, you'd better finish the insertion by yourself.

The AP may be ran with **time acceleration of 10x in atmosphere and 100x in space** and for **multiple vessels simultaneously**, independently of the MFD itself, which means that the MFDs can be shut down during ascent.

Shift I = Disable pitch guidance
Shift P = Switch PID autopilot

PID autopilot manual configuration

The PID autopilot is as universal as possible, meaning that its values are configured only once for DeltaGlider and hardcoded. For each different ship class, there's a calculation performed, which compares the current vessel's angular accelerations and compares them to that of the DG. This ratio lets the PID increase or decrease its values, so the AP works on the same settings.

If however they don't work, there's a possibility of defining the PID values by hand for each vessel class. The values can be adjusted in a running simulation with **[Shift 1]** or **PXY** button for X & Y axes and **[Shift 2]** or **PBN** for bank. Both types have to be adjusted for atmospheric and space PIDs. The currently used PID is displayed on the bottom of main MFD page. The same PID type would be adjusted in the moment of pressing **PXY** or **PBN.**

Once you are satisfied with your values, you'll be interested in saving them in a persistent config file, so that they can take effect after you restart the simulation. The PID definitions are searched for in **Config\MFD\LaunchMFD\** directory. The exact config file name can be obtained through the PID tuning input box with **PXY**.

The available and read tags and their expected data are:

```
SPACE_XY = (vector)
SPACE_BANK = (vector)
ATMO_XY = (vector)
ATMO_BANK = (vector)
ATMO_CONTINUOUS_CONTROL_SWITCH_ANGLE = (float)
ATMO_BANK_TARGET_WINGS_LEVEL_SWITCH_ANGLE = (float)
```

A vector consists of three space separated floating point values, while float is one floating point value. The vectors' are used to pass PIDs' values - proportional gain, or Kp, derivative gain, or Kd, and integral gain, or Ki.

`SPACE_XY` and `SPACE_BANK` PIDs are used for X and Y axes control as well as bank control, correspondingly. Both in space. The same philosophy is used for atmospheric autopilot.

`ATMO_CONTINUOUS_CONTROL_SWITCH_ANGLE` is an angle in degrees, for which the PID AP switches on, and brings its smooth control. Beyond that angle, the RCS and control surfaces work with full power, because only such behavior makes sense for winged craft. Increasing value of this variable increases the spectrum of angles, in which the AP works in its whole scale, this means that it's function of power(distance) will be stretched on this distance. For non winged craft it makes less sense to limit the continuous control spectrum, because they are already pitched to 90°, so reaching the target attitude wouldn't take much time. To disable the limit completely, a value of 180 degrees must be chosen, because the maximal angular distance is in range (-180°, 180°). Such setting would also eliminate a visible sudden jump in control when switching from atmospheric PID to space PID.

`ATMO_BANK_TARGET_WINGS_LEVEL_SWITCH_ANGLE` is an angle in degrees, for which the AP changes its bank from target marker to wings level. Before that, it tries obtain such bank, that the target marker remains directly above the ship, so that it can reach it much quicker by pitching, rather than yawing later on.

A typical definition for the DeltaGlider would be:

```
SPACE_XY = 2.5 16.0 0
SPACE_BANK = 1.0 12.0 0
ATMO_XY = 2.0 2.5 0
ATMO_BANK = 3.0 6.0 0
ATMO_CONTINUOUS_CONTROL_SWITCH_ANGLE = 10.0
ATMO_BANK_TARGET_WINGS_LEVEL_SWITCH_ANGLE = 20.0
```

Shift 1 = tune atmo/space PID for X & Y axes
Shift 2 = tune atmo/space PID for bank

Pitch program

Currently the PEG algorithm assumes that the **current** stage is the **final** stage. This means that if the vessel has more than one stage, the automatic PEG guidance will fail. What can be done about it is defining a pitch program, which guides the ship roughly during the first, booster assisted, stage. Then, after that stage runs out of fuel, the PEG would kick into action and fix the eventual errors. This can be done by defining the pitch program from the start, until the moment right after booster separation. The pitch programs should be placed in the same config file where you place your PID definitions. An exemplary pitch program could look like the following:

```
PITCH_PROGRAM_ALTITUDE
Earth
0 90
10.5 80
22.7 75.8
40 45
60 30
END_PITCH_PROGRAM_ALTITUDE
```

The first column denotes altitude (argument) in kilometers, while the second column denotes the according pitch (value). The values for arguments between the designated altitudes are linearly interpolated. Therefore if you need a more precise and oval pitch profile, you need to increase the granularity, by inserting more altitude-pitch pairs.

For each planet, defined by the first line of the programme, there can be one pitch program defined. They should be placed in the same file with their own start and end markers, for example:

```
PITCH_PROGRAM_ALTITUDE
Earth
0 90
10.5 80
22.7 75.8
40 45
60 30
```

```
END_PITCH_PROGRAM_ALTITUDE
```

```
PITCH_PROGRAM_ALTITUDE
Mars
0 90
15 70
30 50
40 35
END_PITCH_PROGRAM_ALTITUDE
```

If no planet name has been chosen, Earth is assumed by default.

Because the PEG should start to work as soon as boosters are separated, and because the booster separation time is constant, it is reasonable to define the pitch as time based, where the booster separation time (or a few seconds after it) would end the time based pitch program. Therefore, assuming that the separation occurs at T = 127s, a typical time based pitch program would look like the following:

```
PITCH_PROGRAM_TIME
Earth
0 90
20 90
130 20
END_PITCH_PROGRAM_TIME
```

If time based program and altitude based program for the same planet are found in one vessel's config file, then the time based one is preferred.

If you are able to fine tune a spacecraft's pitch program, then please share your work in this thread on Orbiter-Forum.

Roll program

There are two additional configuration boolean switches available for each craft: `ROLL_NEEDED` and `UPSIDE_DOWN`. The first one tells the AP, that roll should be performed right after launch, and the second one tells that the craft should ascend in an upside down manner, such as the Space Shuttle. Also, `UPSIDE_DOWN` implies `ROLL_NEEDED`, so an entry in Shuttle Atlantis config file could look like the following:

```
UPSIDE_DOWN = TRUE
```

Currently, the roll is performed after the craft reaches a pitch of 88.5 degrees or below, because only then the AP can be sure in which direction to roll.

Course correction (inclination)

After you have launched, the Launch MFD calculates the azimuth required to reach the given inclination continuously, subtracting your current North & East velocity from the required orbital velocity allowing to perform course corrections. The error indicator beneath tells
you exactly how many degrees (and decimal parts of them) you should turn to be right on target. Be aware that it's a pure difference between the current and necessary azimuths and is no kind of actual prediction. Because of the imposed subtraction order, the error indicator is always negative if you are faced left of the required azimuth and positive if you are faced right of it. If no target is selected, the error indicator refers to ascending azimuth if your heading is (270°, 360°) or (0°, 90°). If your heading is (90°, 270°) then the indicator refers to the descending azimuth. Same applies for retrograde orbits *(Fig. 1 below)*.

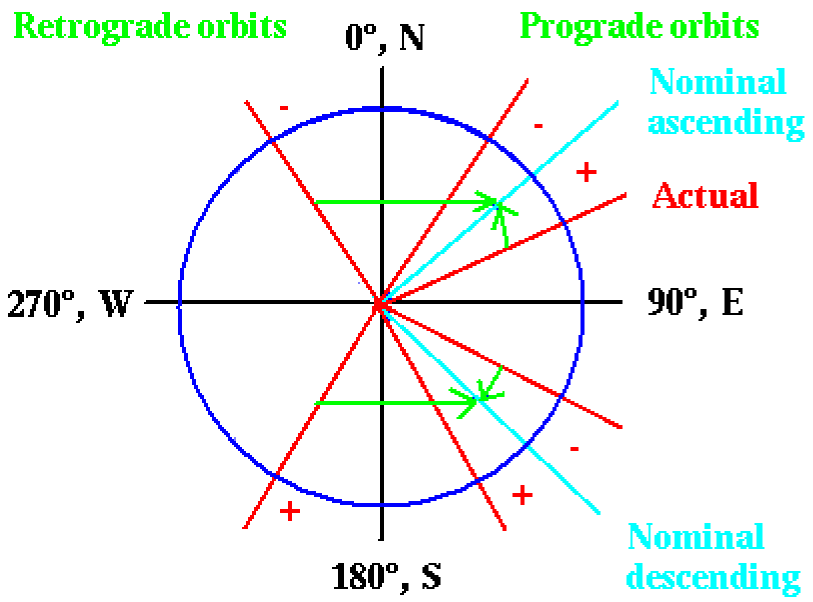
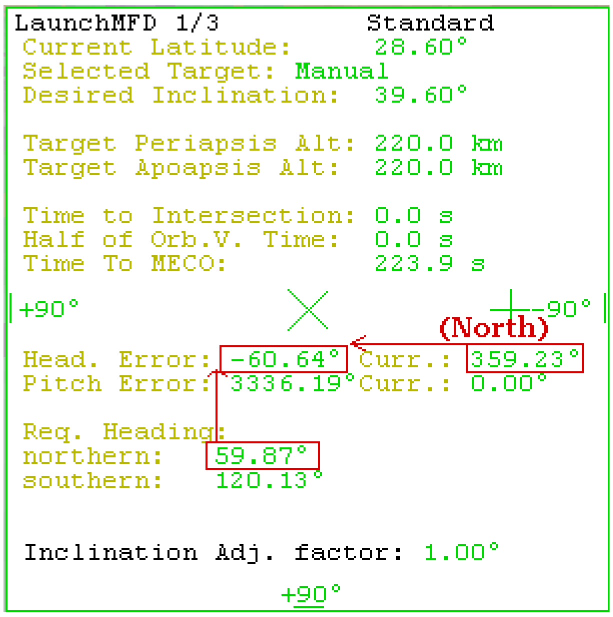


Fig. 1. Actual azimuth referencing when no target is selected and Error's sign.

In the example on the right our actual azimuth is 359° which is between 270 and 360 so is the northern halve, thus the error refers to the calculated, ascending azimuth which is 59.87° for the given inclination, so the difference is -60.64°, meaning that we are 60.64° left of the nominal azimuth.

If, on the other hand you have selected a target, then the error variable will always refer to the appropriate azimuth, which means that you don't have to watch Map MFD to know in which direction to launch any more.



```
LaunchMFD 1/3            Standard
 Current Latitude:        28.60°
 Selected Target: Manual
 Desired Inclination:     39.60°

 Target Periapsis Alt:    220.0 km
 Target Apoapsis Alt:     220.0 km

 Time to Intersection:    0.0 s
 Half of Orb.V. Time:     0.0 s
 Time To MECO:            223.9 s

+90°                              -90°
                            (North)
 Head. Error:  -60.64° Curr.: 359.23°
 Pitch Error:  3336.19°Curr.: 0.00°

 Req. Heading:
 northern:    59.87°
 southern:    120.13°


 Inclination Adj. factor: 1.00°

              +90°
```

Fig. 2. Example of Error referencing when no target is selected.

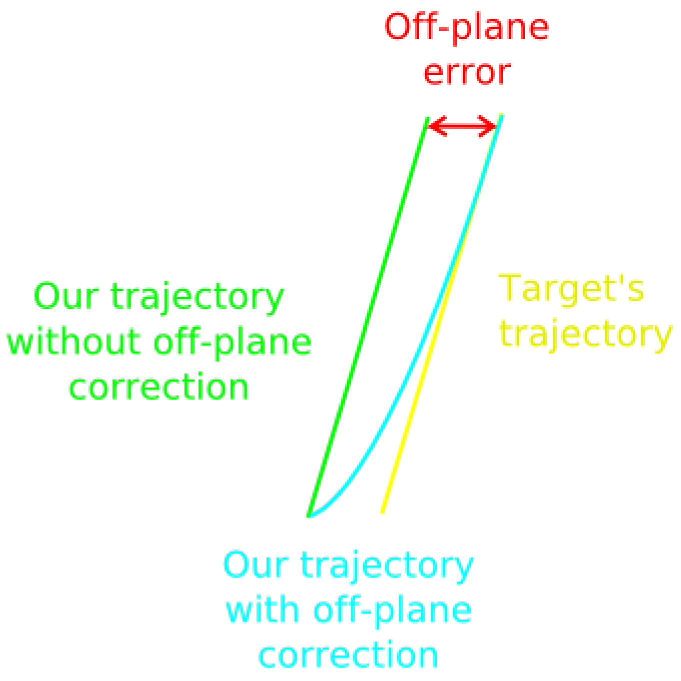
As you begin your ascent and gaining speed, the error indicator becomes more and more sensitive. Your job is to keep it as close to 0.00° as possible. If you won't be doing so, and will keep it eg. at 1.00°, you'll notice very soon that it begins to increase faster and faster, because you will be applying the velocity in a wrong direction, which makes a need of bigger course corrections later. A thing to remember is that it's more fuel efficient to turn your orbit when your velocity is smaller, so point yourself at the proper direction as soon as possible.

To understand it, imagine that you are already on low Earth orbit (about 7776 m/s) and you want to align planes with another target. You'd have to use some amount of fuel. On the other hand if you are sitting on pad and your orbital velocity is mere 408 m/s (as on Cape), switch on Map MFD, accelerate in any direction other than east and see your semi-orbit (red line) changing ultra-easy, and that's what it was – changing inclination. It was performed so easy because your orbital *velocity was so small*

Note however, that because the Error variable becomes more sensitive when you get closer to a specified orbit, an Error of 0.10° is a lesser deviation when at 3/4 of orbit velocity than 0.10° shortly after launch, so you don't need to worry about getting Error to exactly 0.00° during final stages of your ascent. A second thing worth knowing is that because Launch MFD constantly checks your current velocity and chooses the appropriate course accordingly, so even if you swerve, you'll always be directed towards a heading allowing to reach an orbit with the specified inclination, but swerving also makes an unpleasant effect: because you were accelerating in the wrong direction, one velocity vector component is over magnified, so you'll need to use some of your thrust to counter the component, which results in extending your insertion time. This is bad for flight time measurement and earlier flight profile imitating. Read the subsection titled **Time to Intersection.**

Course correction (LAN)

Launch MFD normally corrects only your inclination on the fly. Agentgonzo has implemented an off-plane correction, which means that not only your inclination will be corrected, but also your LAN. The correction can be explained with the following graph:


Off-plane error
Our trajectory without off-plane correction
Target's trajectory
Our trajectory with off-plane correction

Top-down view of situations when you are using and not using off-plane correction, assuming there's no body's rotation

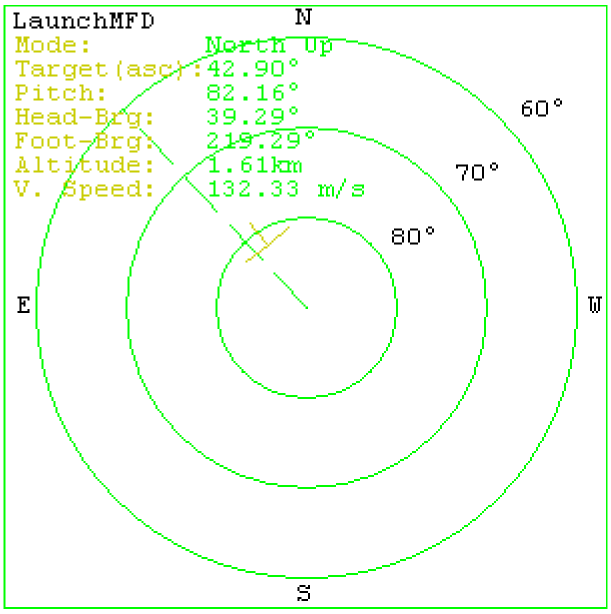
The correction can be switched on with **Shift O** combination or with **COR** button. It will be activated once your position on pad is close enough to intersection with target's orbit. The correction deactivates itself in the last seconds of the flight, because its guidance is meaningless then. The correction also doesn't work with staged vessels currently, as it assumes constant thrust of the vessel.

Despite that the off-plane course correction should help you getting the LANs right, it's not an alternative of timing the launch correctly. The worse you time the launch, the more fuel you'll waste, although your final orbital parameters should be correct in both inclination and LAN. For timing the launch, see **Half of Orb. Vel. Time** subsection, and **Tutorial** section.

Shift O = Switch off-plane correction

Launch Compass

Thanks to Agentgonzo, we now have the Launch Compass inside Launch MFD. The Launch Compass is especially useful for vertically launched crafts, like the Shuttles, allowing you to perform perfect roll manoeuvres. It shows the heading, resulting from your targeted inclination, and your current heading versus pitch in a 2D display, and other info. This information is quite hard to interpret, using the standard HUD display while launching vertically.



```
LaunchMFD
Mode:          North Up
Target(asc):42.90°
Pitch:         82.16°
Head-Brg:      39.29°
Foot-Brg:      219.29°
Altitude:      1.61km
V. Speed:      132.33 m/s
```

The Launch Compass has several view modes , that you can switch them using the default action button - **DEF** or **Shift D** combination. The modes are:

a) North up - the compass is static, and your heading and direction are represented by the moving T marker
b) Head up - your direction representation is static, and the compass rotates depending on your direction
c) Heading up - your heading representation is static, and the compass rotates depending on your heading

Notice that east switched places with west on the compass. There are two reasons: First is that sky on astronomical charts is displayed this way, and second is that if they were placed the "normal" way, then the visual representation of your roll manoeuvre would be counter intuitive, ie. you'd see the marker rotate in a direction contrary to your commands.

Shift D = Default action: Switch display mode.

PEG algorithm

Thanks to Kwan and Agentgonzo, we now have PEG algorithm integrated with Launch MFD. It allows to guide your ship along an insertion trajectory to an orbit specified by periapsis and apoapsis altitudes with **Shift A** combination or **ALT** button. The PEG is enabled automatically, without user's interaction, and displayed on HUD as a cross marker. The guidance marker also takes into account the calculated heading, needed for inserting into an orbit with a specified inclination. This means, that you'll be guided by hand into your orbit. The PEG algorithm is adaptive, just like the standard heading calculation, which means that it allows to perform course corrections, instead of giving you only one chance to get your orbit right.
If the guidance marker is out of your view, an arrow marker will be pointing toward it.

Warning: PEG in its original form is designed for Single Stage To Orbit craft only, which for example the Shuttle is not. Nonetheless the PEG will activate itself on the final stage (after SRBs separation), but don't be fooled, because before that the guidance marker will be exactly on your zenith, so you'll have to rely on your experience for a proper launch profile during the first stage.

Graphical eycandies

She'da'Lier coded an initial graphical representation of the error that allows to get an idea of how fast you change your error. I modified it a bit so that now it switches its scale, ie. It gets more sensitive as the error decreases thus allowing to eyeball the 0.00° state. Particularly, the scale is ±180 if your Error is (180,90), ±90 if it's (90,5) and ±5 if below 5

You will notice that Error value's colour is grey if the value is smaller than ±0.20° and turns green if it goes beyond that, so it's easy to spot that you're off course.

Because the HUD display requires your ship to be derived from VESSEL2 class, the HUD display won't work for very old ships, which are derived from VESSEL class. This is why I've implemented the same functionality as HUD gives in the basic MFD area. You'll see that the graphical error's representation is now two-dimensional, representing heading error and pitch error. You may not always want the pitch error to be included, so it can be switched with **Shift I** combination, or **PTC** button. For the same reason, the PEG guidance marker is now displayed also in the Launch Compass mode.

While actually using the Launch MFD, I noticed that one can easily forget about orbit circulation and go elliptic if he keeps watching the error indicator too closely. This is why when both of your actual velocity components (North & East) exceed calculated ones for the specified orbit, you'll get a red note saying: “Cut your engines!” under the error indicator. It will still be displayed even if you cut them but your velocity still exceeds the calculated one. It's one of those “it's not a bug, it's a feature” things. After all you may want to decelerate then a bit to make the orbit circular. This is when the note disappears. As mentioned, because **both** components must exceed the calculated ones, the feature will not work if you were accelerating in a wrong direction in the final parts of your ascent thus made only one velocity component exceed the required one.

Sound configuration

MFD sound can be permanently turned off using the Launch MFD config file, whose exact path is **Config\MFD\LaunchMFD\LaunchMFD.cfg**

Time to Intersection

After selecting a target, Launch MFD now shows an absolute amount of seconds to next launch window, ie. to a moment in time, when your and the target's orbits crossing will be placed on the position of your launch site. Thanks to this variable, you don't need to watch Align MFD any more to know approximately when the launch should occur.

Half of Orb. Vel. Time variable

Another feature is a Half of Orb. Vel. Time variable. It calculates how long it takes from a moment of taxiing/liftoff to reaching half of orbital velocity, during a **current launch**, so you will get a picture when you should launch **next time**, to not only equalize inclinations but LANs as well (check Basic Theory). To use the value properly, **launch when Time to Intersection equals Half of Orb. Vel. Time**. It turned out that the ~10s error in Shuttles which I mentioned before, was caused by the fact that Align MFD doesn't display exactly seconds when you're grounded. Just try to accelerate time to make Time to Intersection = 313 (the proper Half of Orb. Vel. Time for Shuttles) and compare it with Align MFD. You should see about 300 s there.
An obvious negative of the Half of Orb. Vel. Time variable is that if you launch a manually controlled spacecraft, your ascent profile will have to resemble the previous one, so it is performed in a similar time. If you are on a second ascent and your NLTime for this launch is close to the

previous one, you can be almost sure that you'll get the LANs to to close values.

Direct ascent

Direct ascent mode can be selected by pressing **(Shift M)** combination or **MOD** button. After activating direct ascent autopilot with **(Shift P)** or **AP** button, it works by automatically getting ship into a suborbital trajectory directly in the target orbit's plane, cruising, preferably with SCRAM engines, while waiting for the target to catch up on us. Once an injection window is available, the AP will bring you very close to the target.
There are two modes of direct ascent:

1) Regular – after injection, the ship immediately reaches the target orbit
2) Synchronization – the ship creates a synchronization orbit and the interception occurs at the orbits' intersection

The Synchronization mode should be used when the suborbital velocity is so high, that it's impossible to drastically change the orbit's shape as in the Regular mode. This is a typical case for a full throttle XR SCRAM ascent. After performing autoburn, you should use Align planes MFD and Synchro MFD to fine tune the rendezvous time. You must make sure that the plane alignment is performed before reaching apoapsis. If a node is positioned behind apoapsis, you must accelerate orbit (ant-)normal, to place the node before the apoapsis.

The modes are switched with **(Shift D)** or **DEF** button on the direct ascent page.

The Playback folder of LaunchMFD installation contains examples of direct ascent usage with descriptions.

HOT! Final distance selection for Direct Ascent

When the autopilot for regular DA is enabled, using **(Shift A)** or **ALT** button, you may enter the final distance to the target **in meters**. Positive values mean ending *n* meters above target and negative mean ending below the target. Notice that for reasons of realistic spaceflight, the positive values make less sense than negative, because you always approach the target from below, firing thrusters in the direction of target. If you intend to end up above the target, there would be a moment when you cross the altitude of the target, firing your exhaust right at the target.

Great Circles (a'la Map MFD)

To help timing your launches visually, without the need of having Map MFD opened, great circles of your ship and of the target are being displayed. They can be controlled only via keyboard, and the controls are the same as for Map MFD. However, because currently there's no moving feature, zooming work only in the tracking mode.

Shift G = Switch drawing of great circles
Shift K = Switch track mode
Shift Z = Zoom in
Shift X = Zoom out
Shift C = Increase precision of great circles
Shift V = Decrease precision of great circles

5) Basic theory

The problem of calculating an exact launch azimuth appears when you want to meet with a target in orbit and you have relatively low amount of fuel and error margin, or you want to save as much of the fuel as possible. By launching with the azimuth, calculated earlier, ie, such that will allow you to insert yourself in an orbit with the same inclination as target, you save the fuel which you'd have to use for a plane change manoeuvre otherwise.

Some basic information:
***Inclination** is in this case the angle between a reference plane (eg. ecliptical or equatorial. We refer to the second here.) and your current orbit's plane.*
An orbit with inclination = 0° passes directly over equator, from west to east. Orbit with 90° inclination passes from south to north pole. Orbit with 180° inclination is also placed over the equator but goes from east to west. Orbits with inclinations between 90° and 180° are called "retrograde orbits".
***Launch azimuth** is the angle between north direction and the projection of the initial orbit plane onto the launch location. It is the compass heading you head for when you launch. (from OrbiterWiki)*
***Azimuth** = 0° is north, 90° is east, 180° is south and 270° is west*
If you are interested in how to calculate the launch azimuth, visit this page:
http://www.orbiterwiki.org/wiki/Launch_Azimuth
and check OrbiterSDK\Samples\LaunchMFD\Math\AzimuthMap.cpp file for the implementation.

For input values of Cape Canaveral (it's latitude), ISS (it's inclination) and Earth's physical parameters the equations give a result of about 42.85° (north east) or (180°-42.85°) = 137.15° (south east), so you should launch in one of this directions depending on ISS' orbit projection on Map MFD (yellow line) "ascends" (42.85° – on the left picture) or "descends" (137.15° on the right picture), w.r.t. your launch site.

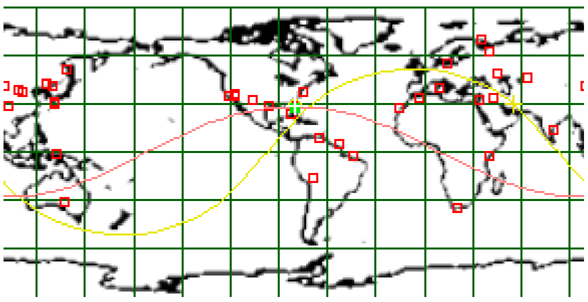
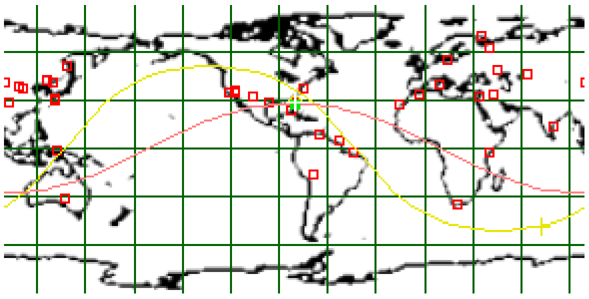



Fig. 4 Choosing a proper heading, depending on situation.
Left - choosing northern, right - southern azimuth

During your ascent the azimuth will be changing despite you're going straight. Don't worry about it because it's normal and results from one of the basic equations (check OrbiterWiki) from which we can see that an inertial azimuth is a function of your current (thus changing) latitude. The other reason is Earth's rotation.
Assuming that you are able to use Launch MFD to insert into an orbit with the inclination identical as the target's (which is represented on Map MFD as your semi-sine amplitude), there's also another problem you'll face: **timing the launch** (represented as phase shift).

If you targeted ISS and accelerated time, you would see ISS' orbit moving left (from east to west). The movement results from compensation of Earth's rotation on a static 2d map, just as Sun moves from east to west. Your objective is to launch some small amount of time before ISS' orbit passes almost exactly overhead. On the left picture below we see how your orbit would look like if we launched too early and on the right – too late. Notice that inclinations are already the same.

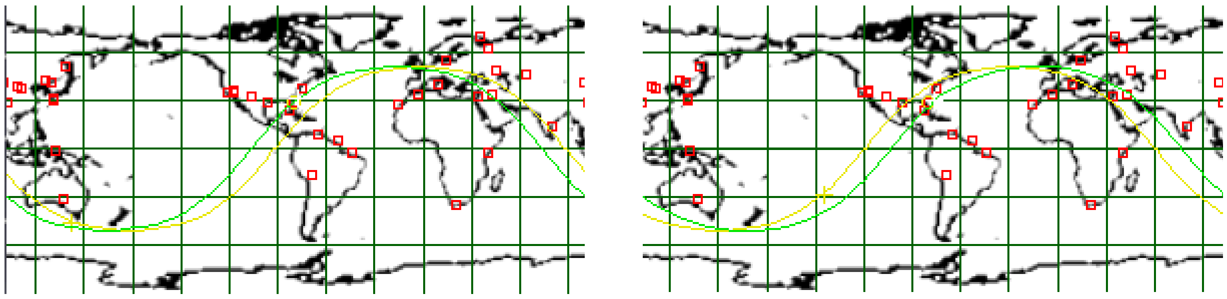
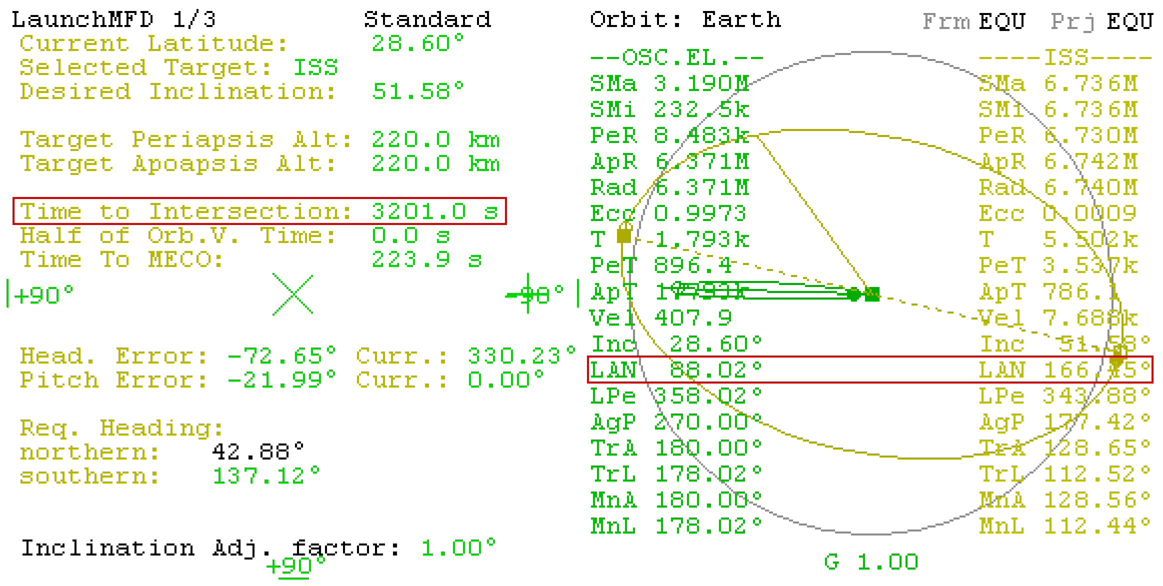


Fig. 5. Results of incorrect launch timings.
Left - launched too early, right - too late

After selecting a target, Launch MFD gives you almost an exact number of seconds to the moment when a target's orbit passes your launch site *(not entirely exact because LAN drift resulting from non-spherical gravity sources, hasn't been coded as according to Orbiter's API Reference, Orbiter sometimes disables the non-sph. grav. src. under high time accelerations, which would render long time predictions useless).*
If you are sitting on a **prograde** rotating body (every planet in Solar system, except Venus), your LAN will be **increasing,** while LANs of bodies in orbit will be relatively stable (not entirely due to non-spherical gravity sources but let's forget about it for now). We want to launch so that upon insertion our LAN would have grown big enough to match that of target. However, note that our LAN won't stop increasing right after launch, but only when we're in orbit. If we launched exactly when ISS was overhead (Time to Intersection = 0), we wouldn't create our orbit in time and ISS' orbit would move relatively west of us.



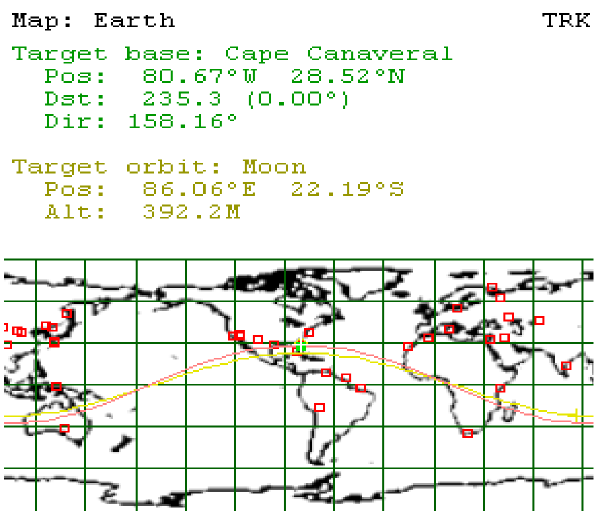
```
LaunchMFD 1/3              Standard
 Current Latitude:          28.60°
 Selected Target: ISS
 Desired Inclination:       51.58°

 Target Periapsis Alt:      220.0 km
 Target Apoapsis Alt:       220.0 km

 Time to Intersection:      3201.0 s
 Half of Orb.V. Time:       0.0 s
 Time To MECO:              223.9 s
|+90°                                  -90°|

 Head. Error: -72.65°  Curr.: 330.23°
 Pitch Error: -21.99°  Curr.: 0.00°

 Req. Heading:
 northern:    42.88°
 southern:    137.12°

 Inclination Adj. factor: 1.00°
                    +90°
```

```
Orbit: Earth                    Frm EQU  Prj EQU
--OSC.EL.--                     ----ISS----
SMa 3.190M                      SMa 6.736M
SMi 232.5k                      SMi 6.736M
PeR 8.483k                      PeR 6.730M
ApR 6.371M                      ApR 6.742M
Rad 6.371M                      Rad 6.740M
Ecc 0.9973                      Ecc 0.0009
T   -1.793k                     T   5.502k
PeT 896.4                       PeT 3.537k
ApT 1.793k                      ApT 786.1
Vel 407.9                       Vel 7.688k
Inc  28.60°                     Inc  51.58°
LAN  88.02°                     LAN 166.45°
LPe 358.02°                     LPe 343.88°
AgP 270.00°                     AgP 177.42°
TrA 180.00°                     TrA 128.65°
TrL 178.02°                     TrL 112.52°
MnA 180.00°                     MnA 128.56°
MnL 178.02°                     MnL 112.44°
                G 1.00
```

Fig. 6. Left: Time to intersection,
Right: your LAN should increase up to less than 166.45°

The solution is launching when Time to Intersection = time after which your craft reaches half of its final orbital velocity, that is Half of Orb. Vel. Time. You can read the Orbital Speed from Surface MFD after switching it to Orbital Speed mode (OS). So if the final orbital velocity of a Shuttle inbound for ISS is 7776 m/s then you should start when the Shuttle gains velocity equal half of the orbital velocity = 7776 / 2 = **3888 m/s**, which the Shuttle achieves in **310 seconds.** In practice, knowing the exact time for a manually controlled vehicle is impossible, and even knowing this time from the previous launch using the MFD (**Half of Orb. Vel. Time**), you're not able to make your profile completely resemble the previous one.

Also notice that you can't use these equations for orbits whose inclinations amounts are less than your latitude. Explaining mathematically, because the argument of ArcSin can't be bigger than “1” (Check OrbiterWiki). Speaking in a human language: If you're starting from Cape Canaveral at current dates, and you target the Moon, then even with maximal effort from your side (launching on heading = 90° = east) you won't align the plane of your orbit with Moon's in a way other than the standard one and its derivatives. It must happen either on LEO or between trans-lunar injection (TLI) and Moon interception, inclusive.

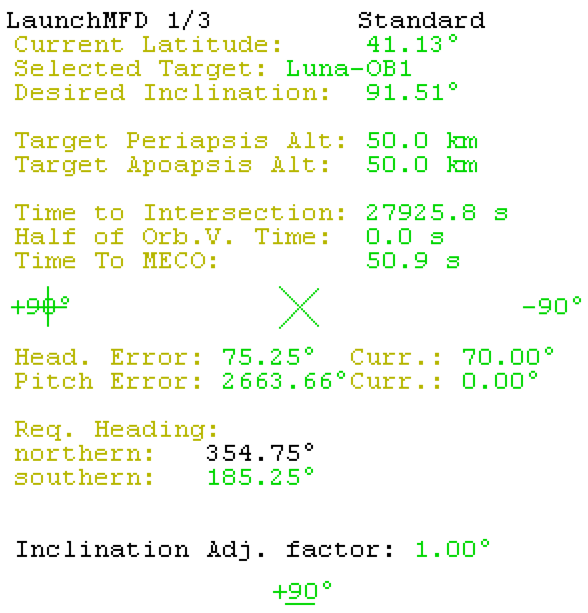


```
Map: Earth                              TRK
Target base: Cape Canaveral
  Pos:  80.67°W  28.52°N
  Dst:  235.3  (0.00°)
  Dir: 158.16°

Target orbit: Moon
  Pos:  86.06°E  22.19°S
  Alt:  392.2M
```

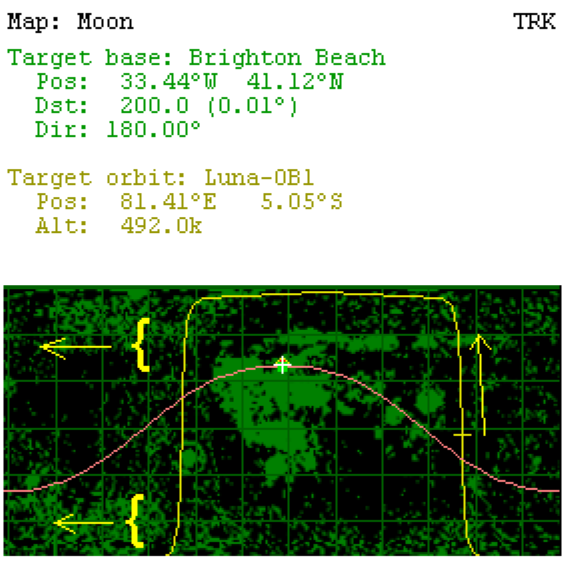
Fig. 7: Inability to align planes with a target whose inclination is lower than your latitude by launching with any heading.

6) Tutorial

In this example I chose to launch to rendezvous with Lunar station Luna-OB1



```
LaunchMFD 1/3           Standard
 Current Latitude:      41.13°
 Selected Target: Luna-OB1
 Desired Inclination:   91.51°

 Target Periapsis Alt:  50.0 km
 Target Apoapsis Alt:   50.0 km

 Time to Intersection: 27925.8 s
 Half of Orb.V. Time:   0.0 s
 Time To MECO:          50.9 s

+90°                                -90°

 Head. Error: 75.25°  Curr.: 70.00°
 Pitch Error: 2663.66°Curr.: 0.00°

 Req. Heading:
 northern:    354.75°
 southern:    185.25°


 Inclination Adj. factor: 1.00°

                  +90°
```



```
Map: Moon                         TRK
Target base: Brighton Beach
   Pos:  33.44°W  41.12°N
   Dst:  200.0 (0.01°)
   Dir: 180.00°

Target orbit: Luna-OB1
   Pos:  81.41°E   5.05°S
   Alt:  492.0k
```

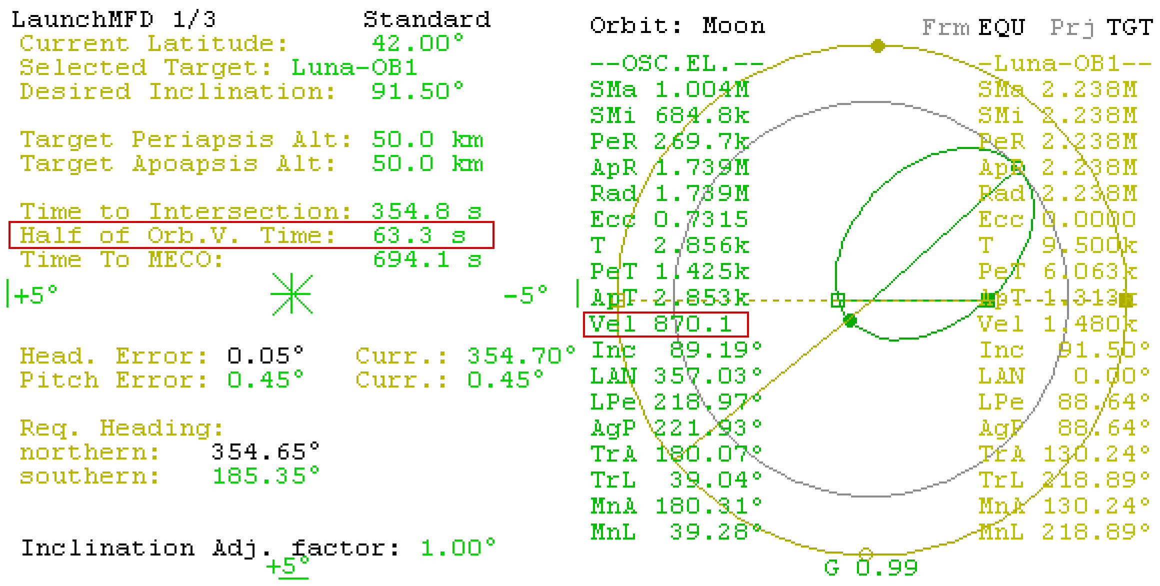
Fig. 8. Planning the launch

Planning

Look at the picture above. I have targeted Luna-OB1 in both Launch MFD and Map MFD. Luna-OB1 is the yellow cross moving on a yellow orbit in Map MFD. The station's inclination readout says 91.51°, which is greater than 90°, thus the station moves on a retrograde orbit. You can also judge it by observing the yellow cross' movement on its orbit with a help of time acceleration. Luna-OB1 (Luna for short) moves as the vertical arrow points. Moon is a prograde rotating body as well, so Luna's orbit (as well as any other) will be shifting west on a static map. This knowledge helps us to determine:
a) which crossing of Luna's orbit and our position will reach us first – i.e. the eastern one
b) which azimuth of two to choose – i.e. Luna will be passing us from south to north on that crossing, so we'll choose the ascending = 354.75° (northern) azimuth to get us on the right orbit.

Defining Half of Orb. Vel. Time

Knowing everything we need, we can make a test launch to help us determine when it will be a good time to perform the actual ascent (to get a proper Half of Orb. Vel. Time). I don't think that things like that are made in reality, but manual ascents also aren't performed. You can now launch in any heading of the two and try to keep your Error relatively small. Time is calculated from the moment you activate thrusters, so if you need to yaw your ship in a proper direction for some amount of time, you'll have to take this time into account during the second launch either by making the same yaw or subtracting it from time or waiting the same amount of time above the ground. When you reach half of orbital velocity, you'll be informed about it by a male voice. In the picture below, I've overdone with the velocity a bit, but it doesn't matter, as Half of Orb. Vel. Time var. stays persistent up to a moment when you land, and reach the half of orbital velocity again.



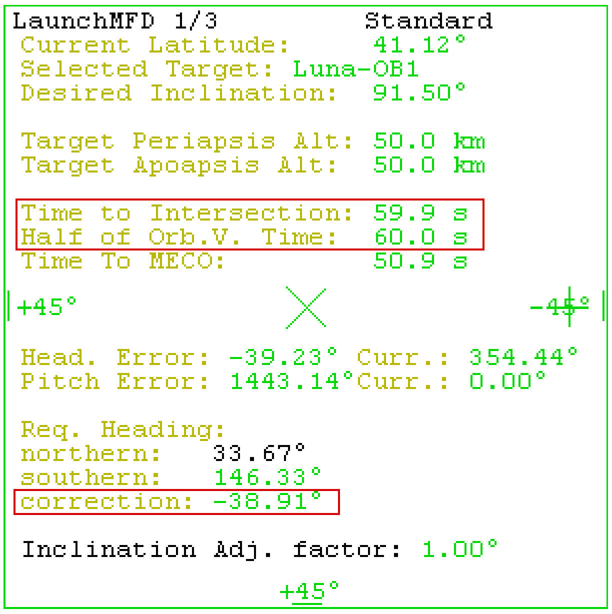
```
LaunchMFD 1/3            Standard
 Current Latitude:        42.00°
 Selected Target: Luna-OB1
 Desired Inclination:     91.50°

 Target Periapsis Alt:    50.0 km
 Target Apoapsis Alt:     50.0 km

 Time to Intersection:   354.8 s
 Half of Orb.V. Time:     63.3 s
 Time To MECO:           694.1 s
|+5°                               -5° |

 Head. Error: 0.05°    Curr.: 354.70°
 Pitch Error: 0.45°    Curr.: 0.45°

 Req. Heading:
 northern:     354.65°
 southern:     185.35°

 Inclination Adj. factor: 1.00°
                  +5°
```

```
Orbit: Moon                Frm EQU  Prj TGT
--OSC.EL.--                -Luna-OB1--
SMa 1.004M                 SMa 2.238M
SMi 684.8k                 SMi 2.238M
PeR 269.7k                 PeR 2.238M
ApR 1.739M                 ApR 2.238M
Rad 1.739M                 Rad 2.238M
Ecc 0.7315                 Ecc 0.0000
T   2.856k                 T   9.500k
PeT 1.425k                 PeT 6.063k
ApT 2.853k                 ApT -1.313k
Vel 870.1                  Vel 1.480k
Inc  89.19°                Inc  91.50°
LAN 357.03°                LAN   0.00°
LPe 218.97°                LPe  88.64°
AgP 221.93°                AgP  88.64°
TrA 180.07°                TrA 130.24°
TrL  39.04°                TrL 218.89°
MnA 180.31°                MnA 130.24°
MnL  39.28°                MnL 218.89°
              G 0.99
```

Fig. 9. Half of orbital velocity reached, NLT variable set.

Performing the launch

You may now use Scenario Editor to place your ship back in a base and await for a launch window. If you are using a realistic craft or such that can't be simply moved by the Editor, you'll need to remember the Half of Orb. Vel. Time and restart the scenario. Accelerate time (T) and wait for Luna's orbit to nearly cross your position, indicated by a small time to Launch Window. Launch when Time to Intersection = Half of Orb. Vel. Time from Launch MFD (or the value that you had to remember before restarting the scenario), and try to imitate your previous launch as much as you can. This time, though, you can chose only one of the two headings, i.e. the one flashing in grey.



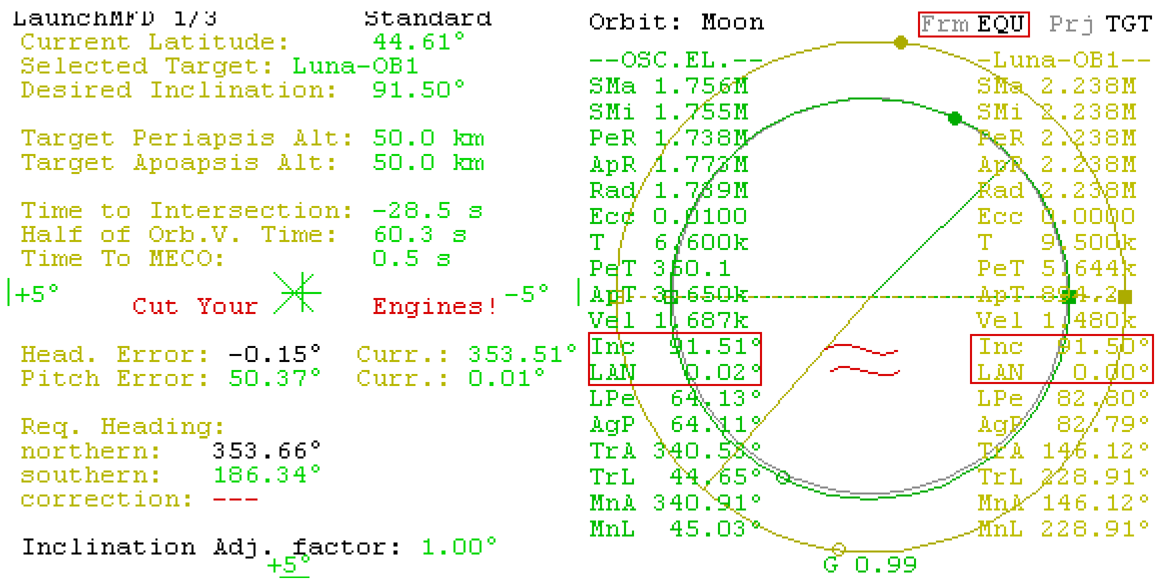
```
LaunchMFD 1/3             Standard
 Current Latitude:         41.12°
 Selected Target: Luna-OB1
 Desired Inclination:      91.50°

 Target Periapsis Alt:     50.0 km
 Target Apoapsis Alt:      50.0 km

 Time to Intersection:     59.9 s
 Half of Orb.V. Time:      60.0 s
 Time To MECO:             50.9 s
|+45°                              -45° |

 Head. Error: -39.23° Curr.: 354.44°
 Pitch Error: 1443.14°Curr.: 0.00°

 Req. Heading:
 northern:     33.67°
 southern:     146.33°
 correction:  -38.91°

 Inclination Adj. factor: 1.00°

                 +45°
```

Fig. 10. These variables' equality indicates a good time for launch. Note off-plane correction switched on with COR button

If your second launch is performed faster or longer so that half of orbital velocity is reached in a different time, there will be LAN difference upon reaching orbit, depending on how fast the body rotates. On the Moon it won't be much of a difference because it hardly rotates. On Earth on the other hand, your LAN would be about 0.05° different for a 10 second shift in launches.

Correctness checks

When you are in Orbit, you can check which thing you will have done wrong, ie:
a) not following the Error's directives or
b) launching at incorrect time
by must switching Orbit MFD to Equatorial Frame, and comparing your inclination and that of target for point **a)**, and compare LANs for **b)**. If your LAN is greater on a prograde rotating body, then you launched too late, and vice versa. Switching Orbit MFD to the equatorial frame is mandatory, as this is the frame which LaunchMFD uses. There's not much difference in Moon's equ/ecl frames but for Earth you could be misinformed that your inclinations were different if your LANs were very different in the ecliptic frame, while in the equatorial frame inclinations could be the same.



```
LaunchMFD 1/3            Standard
 Current Latitude:        44.61°
 Selected Target: Luna-OB1
 Desired Inclination:     91.50°

 Target Periapsis Alt:    50.0 km
 Target Apoapsis Alt:     50.0 km

 Time to Intersection:   -28.5 s
 Half of Orb.V. Time:     60.3 s
 Time To MECO:            0.5 s
|+5°       Cut Your         Engines! -5° |

 Head. Error: -0.15°  Curr.: 353.51°
 Pitch Error: 50.37°  Curr.: 0.01°

 Req. Heading:
 northern:    353.66°
 southern:    186.34°
 correction:  ---

 Inclination Adj. factor: 1.00°
                   +5°
```

```
Orbit: Moon                 Frm EQU  Prj TGT
--OSC.EL.--                 --Luna-OB1--
SMa 1.756M                  SMa 2.238M
SMi 1.755M                  SMi 2.238M
PeR 1.738M                  PeR 2.238M
ApR 1.773M                  ApR 2.238M
Rad 1.759M                  Rad 2.238M
Ecc 0.0100                  Ecc 0.0000
T   6.600k                  T   9.500k
PeT 360.1                   PeT 5.644k
ApT 3.650k                  ApT 894.2
Vel 1.687k                  Vel 1.480k
Inc  91.51°                 Inc  91.50°
LAN  0.02°                  LAN  0.00°
LPe  64.13°                 LPe  82.80°
AgP  64.11°                 AgP  82.79°
TrA 340.52°                 TrA 146.12°
TrL  44.65°                 TrL 228.91°
MnA 340.91°                 MnA 146.12°
MnL  45.03°                 MnL 228.91°
                G 0.99
```

Fig. 11. Despite a different insertion time, our final parameters are nearly perfect.
Notice the Equatorial Frame in Orbit MFD, as this is the frame which is used by Launch MFD

I'm sorry, I was wrong before, pointing you to compare the four variables in Align MFD, which only uses ecliptic frame, in which inclination is also LAN dependent.
If for some reason you forgot to watch Orbit MFD and to circularise your orbit, you'll be informed by Launch MFD visually and verbally by the necessity of cutting your engines.

7) Known issues & to do

– PEG works only for SSTO ships. Multistage PEG is the next target

8) Author's notes

a) Support - I'll always try to provide support for my addon when I have time. If you have found a bug in Launch MFD, then write a private message to me at http://orbiter-forum.com/ (nick - Enjo) or send me an e-mail. It's very possible, that if you start a thread, I might not be able to read it.

b) Localisation - If you want to make you own language version of Launch MFD, then do the same as above. Check the file OrbiterSDK/samples/LaunchMFD/localisation.h to measure the necessary amount of work. By localising Launch MFD, you agree with GPL v3 license statements (nothing that hurts you, but protects me instead).

9) Credits

The following people somehow influenced the current state of the add-on (loose order):
- **Jerry Jon Seller** (book "Understanding Space" from where the new equations come. I've modified them a bit)
- **Andy44** (the above equations and launch compass idea provider)
- **Jaggers, R. F.** (The original PEG algorithm author)
- **Urwumpe** (sharing info about timing the launch and orbits' crossing)
- **Vanguard** (dressing into a MFD and coding basic functions)
- **Martin Schweiger, PhD**. (providing ScnEditor's sources which I use here and giving Orbiter :))
- **Auditek (Sim4Lin)** (testing, bug reports, and pointing me to a good voice synthesizer - Festival)
- **David** (bug reports)
- **She'Da'Lier** (help with saving values, coding graphical representation and much patience :))
- **Jarmonik,gp** (inspiration for the current MFD's functions)
- **Dan Steph** (OrbiterSound 3.5 SDK which allowed Launch MFD to use sound)
- **Chaps @ OrbitHangar IRC** (programming help)
- **Gary** (suggestions about sound)
- **Daver** (sharing some Public Domain code)
- **X-Viila** (nicely whitening Earth's map used in documentation)
- **Computerex** (sharing code that allows using multiple vessels)
- **Kwan** (PEG algorithm implementation, which we borrowed)
- **Face** (helping Agentgonzo with Flight Director)
- **Agentgonzo** (help with targeting celestial bodies, PEG integration, HUD display, source cleanup)
- **C3PO** (providing a dummy mesh for LaunchMFD-Probe)
- **Zachstar**, and others from Orbiter-Forum (testing)
- **Face, Fizyk** (trigonometry help, which resulted in correctly calculating time to launch)
- **Ed Williams** (awesome page about aviation - http://williams.best.vwh.net/avform.htm)
- **Chris Veness** (another aviation page - http://www.movable-type.co.uk/scripts/latlong.html)
- **Prof. Edvard Sazonov** (Fuzzy Engine for Java)
- **Agnieszka Nowak-Brzezińska, PhD.** (my MSc. thesis advisor)
- **BrianJ, RisingFury** (help with maximal angular acceleration of a vessel)
- **Cras** (testing)
- **turtle91** (great bug report about CTDs with incompatible vessel classes (VESSEL2))
- **orb** (help with the problem above)
- **boogabooga** (bug report – CTD when switching to Direct Ascent page)
- **blixel, SolarLiner** (suggestion about setting the final distance to target in Direct Ascent program)

Sorry if I forgot anybody. If you think you should be in that list, then send me a private message at orbiter-forum.com or e-mail me.

10) Directories / files list

Config\MFD\LaunchMFD
Config\Vessels\LaunchMFD-Probe.cfg
Doc\LaunchMFD
Flights\Direct ascent DG-S - phase 1
Flights\Direct ascent DG-S - phase 2
Flights\Direct ascent DG-S - phase 2 - synchro
Meshes\LaunchMFD-Probe.msh
Modules\Plugin\LaunchMFD.dll
Orbitersdk\samples\LaunchMFD
Scenarios\LaunchMFD
Scenarios\Playback\LaunchMFD
Sound\LaunchMFDEnjo

11) Informations for developers

In the process of cleaning up the code of Launch MFD, once I finally learned some practical Object Oriented Programming, I've separated many reusable basic classes, that can be used in your MFD or even other projects. The classes' licenses are quite permissive. The classes which contain basic math, which I classify as Human Legacy, are licensed under BSD, which means that you can use them even in a closed sourced / commercial project as long as you provide copyright note (I didn't invent these equations, but I've put them together). The more advanced ones are licensed under LGPL, which means that you either GPL/LGPL your project (thus sharing your code) if you use these specific classes, or you link them dynamically (via .dll). The LGPL license has been chosen to at least try induce some more code sharing from other developers. Aside from having a better informed community this way, note that it helps maintaining the developers' addons, even after they leave the scene. There are too many examples of orphaned, thus currently entirely junk addons, just because somebody didn't want to share their thoughts for some unknown reason.

Let's start from the most interesting package, which is:

Multiple Vessels Support Library (LGPL)

Introduction

Normally the MFDs in Orbiter are constructed as one instance. Imagine opening an MFD in a vessel. The MFD's constructor is called which creates the instance and its variables are initialised. Now you make some changes in the MFD, thus modifying the internal variables. Next, you close the MFD (destructor is called) and reopen it (constructor is called). This causes all the modified variables to be reinitialised. Martin's solution to this (see Custom MFD in OrbiterSDK) is storing the variables in static member variables (meaning: common for all class instances) by using method `MFD::StoreStatus()`, which is called just before MFD destruction. This way, after MFD reconstruction, the then called `MFD::RecallStatus()` allows to read the static data and put it back where it were. An amateur's solution would be to even avoid `MFD::StoreStatus()` and `MFD::RecallStatus()` altogether, and store the values in global variables. Both solutions would solve the problem of closing and opening an MFD in one vessel. However, they obviously have negatives as they present a lousy approach to Object Oriented Programming, especially the global variable approach. The Martin's solution, although more encapsulated, demands that you keep track of all the variables, and store and recall each one of them – note that you have to **remember** to do this for each new variable, which is quite a tedious and human error-prone work. It makes you wonder if other than some encapsulation, there's any positive compared to the global variables approach. There's also one big common flaw in both of these approaches: the choices made in the MFD will be shared between all vessels.

Fortunately, the MFD destruction / reconstruction scheme is consequently exercised during switching vessels, and the MFD's constructor is being passed a pointer to the currently operated vessel (treat it as a vessel's ID). This feature can be used to cheat the system a bit and use the pointer as a key of a structure, which contains sets of each vessel's individual variables. This way, whenever an MFD is constructed, the passed pointer can be used to search for the particular vessel's set of variables, thus “restoring” the values, or if the set of variables didn't exist yet, a new set is created.

AFAIK, the idea first came from Computerex and Face, although Face uses hashtables, which in my opinion is a bit too complicated approach compared to what can be done, so I engineered the Computerex' solution, by making it less global and more Object Oriented (I did need some time and experience for this myself!). The OOP approach “hides”, from the initially inexperienced MFD coder, details of the multiple vessels support, and leaves him the total joy of MFD coding itself. Of course, an interested coder can still have a look at the library to see its simplicity :)

There's also one thing to consider – an amateur's approach to coding MFDs would be to place all the MFD's cyclical logic in `MFD2::Update()` method. The logic inside might then include, for example: printing values on the MFD's drawing context, routines for playing sound on certain events and for of course checking for these events, as well some guidance routines possibly. There are two problems with this approach – first, the `MFD2::Update()` method is called only as often as it's defined in the Orbiter Launchpad, and not as often as possible, so any guidance routines, which require precision resulting from frequent flight data updates, would not work properly. Second problem is that the `Update()` method is called only when the MFD itself is opened, which is logical, when you think about what is supposed an `Update()` method, belonging to MFD2 class do – update the MFD, and nothing more :) This means, that all your guidance and event routines would not be called at all if you switched to external view or close the MFD after enabling the guidance. The proper place to put such routines is in `oapi::Module::clbkPreStep()`, which is called independently of the MFD. A problem arises though, how to make your Module derived class communicate with the MFD and its variables other way than by getting more global...

The Multiple Vessels Support Library solves all the mentioned problems. To sum up, the library's features are:

- storing altered variables between MFD reconstructions
- storing the variables for each vessel independently
- ability to execute precision routines from `oapi::Module::clbkPreStep()`, while communicating with the MFD's variables in an encapsulated fashion (without global variables except for the Module derived class itself)

Although the library works perfectly and is proven to be CTD resistant even in such uncertain events like vessel deletion, its interface may change (evolve) some time, because it's still being developed. This means that I'd currently rather advice against dynamic linking with your closed source project, but link the classes directly, thus having to GPL, or at least LGPL your project, or risk broken API upon updating the library some day otherwise. This is not an Open Source black mail at all – I'm just being honest.

Usage

There are just a few things that you need to do to use the lib's features, and I will cover them in this paragraph. The package contains Doxygen documentation. You should open it now and keep it as a reference, to view specific classes that I'll be talking about. The necessary steps are the following:

1) Derive a class from `MFDData` to add any additional members and functionalities that you want your MFD to contain. In this example, it will be called `MyMFDData`.
2) Derive from `PluginMultipleVessels`, where the method `ConstructNewMFDData` should return a new `MFDData` derived object pointer (`MyMFDData *`). It will be stored and managed internally. In this example, the Plugin derived class will be called `PluginMyMFD`. For instance:

```
MFDData * PluginMyMFD::ConstructNewMFDData( VESSEL * vessel )
{
   return new MyMFDData( vessel );
}
```

3) Declare a global pointer to the derived Plugin class, initialize it and register it in Orbiter's InitModule() along with the usual MFD inits.

```
PluginMyMFD * pPluginMyMFD;

// Called on module init
DLLCLBK void InitModule (HINSTANCE hDLL)
{
   // init spec and register MFD mode
   pPluginMyMFD = new PluginMyMFD(hDLL);
   oapiRegisterModule (pPluginMyMFD);
}
```

4) Declare a specific constructor that accepts the derived PluginMultipleVessels class and Declare the derived MFDData in the main MFD class, for instance:

```
class MyMFD : public MFD2
{
  public:
    /// Default constructor
  MyMFD( DWORD w, DWORD h, VESSEL * vessel, PluginMyMFD * pluginMyMFD );
    /// Returns MFDData
    MyMFDData * GetData() const;

  private:
    MyMFDData * m_data;
}
```

5) Pass the Plugin to MFD's constructor, initialize the MyMFDData and perform a simple check

```
// Constructor
MyMFD::MyMFD(DWORD w, DWORD h, VESSEL *vessel, PluginMyMFD * pluginMyMFD )
// Initialisation list
: MFD2 (w, h, vessel)
/* init m_data with PluginMultipleVessels's return value, depending on the vessel pointer.
If dynamic cast fails, the m_data member becomes NULL. */
, m_data(dynamic_cast<MyMFDData *>(pluginMyMFD->AssociateMFDData(vessel)))
{
   if ( m_data == NULL ) // Programming error
      sprintf_s(oapiDebugString(), 512, "m_data pointer type is not compatible with the
pointer that was being assigned to it in Ctor");
}
```

6) Place all vessel state updates in MyMFDData::Update() and call it in MFD2::Update().

```
// Repaint the MFD
bool MyMFD::Update ( oapi::Sketchpad * skp )
{
   if ( m_data == NULL )
      return false;

   // Update all ship's variables
   m_data->Update();
   // Draws the MFD title
   Title (skp, "My Multiple Vessels MFD");

   PrintResults(skp);

   return true;
}
```

7) If it's necessary, you can update each of the vessels' MFDData in your derived plugin's UpdatePreStep() (or UpdatePostStep() if needed) that runs even if the MFD itself is disabled and for all vessels simultaneously. Exemplary usage would be autopilots.

```
void PluginMyMFD::UpdatePreStep( MFDData * data )
{
   // Normally you'd use dynamic_cast and check if the returned value is NULL,
   // but this method is supposed to work fast enough.
   MyMFDData * myMFDData = static_cast<MyMFDData *> (data);
   AutopilotBase * apBase = m_apMan.GetAP(myMFDData->GetAutopilotType());
   if (apBase != NULL)
   {
      // Only now it's reasonable to call data->Update() not to waste resources in case it's
actually not needed.
      myMFDData->Update();
      apBase->Guide(myMFDData);
   }
}

void PluginMyMFD::UpdatePostStep( MFDData * data )
{
  // Must be implemented
}
```

The package contains a very simplified, working example of the library's usage. Good luck!

MFDButtonPage (LGPL)

Allows to create multiple MFD button pages, with little extra logic. To use it, derive from the class, adding your MFD as a template argument. In the child class' constructor, register menus and functions. You also need to add handlers of the button/key events into your MFD. The class' instance could be stored in the oapi::Module class and passed to the MFD as a reference, or it could be global. Check the Doxygen documentation for more details.

Math and Systems libraries (BSD)

The Math library and its dependency – (Coordinate) Systems library, contain some useful, organised math for any space related project. They are licensed under BSD. Let's name the Math modules:

- AzimuthMath – calculates azimuth required to reach a specified orbit. Both in rotating and non rotating frame of reference
- BinSearchArg – searches for an argument of function, that produces a given reference value of the provided function (an opposite of y = f(x)). It's very fast and has a complexity of 2*log(maxArgument/precision)

- GreatCircleMath – calculates position of a ship in the Great Circle environment, given percentage of distance passed between two defined geographic points. Normally such calculations are valid for a quarter of the Great Circle. The extension inside allows for 360° calculations with good precision and small execution time.
- GreatCircleAdvMath – helps drawing the Great Circle on a graphical context, like MFD.
- SpaceMath – some currently specific space related math, like calculating orbit's momentum
- SpaceMathBody – containts equations that depend on planetary mass and radius (or altitude above surface)
- VectorMath – manipulations of vectors, containing methods similar to OrbiterSDK – dot and cross product, as well as calculation of angle between two vectors via atan2.
- GeneralMath – everything that doesn't fit into above categories. Currently contains numerical Simpson integration, which accept any function pointer, having argument and arbitrary data

The Systems library is a set of the following coordinate systems:

- Geo – geographic point
- Spherical
- Vect3 – Cartesian
- Point – a 2d point

as well as a class which converts one into another – SystemsConverter, along with its derived version – SystemsConverterOrbiter, used to convert the systems into Orbiter's VECTOR3 and vice-versa. Additionally the Vect3 contains utility members like len() (vector's length), norm() (normalised vector) and a set of intuitive overloaded operators, just like those in OrbiterSDK.

A question may arise – why do we need to break open doors by doubling the OrbiterSDK? The answer is – because they're not entirely open. The VECTOR3 is a C struct, that can't have its own length and normalise methods, nor proper constructors. Each time you want an empty VECTOR3, you have to initialise all the data to 0 individually, nor can you init it with other data. This is not Object Oriented Programming. Another problem is, that the VECTOR3 is an integral part of the SDK. If you wanted to use the SDK math in other, Orbiter-unrelated project by including the SDK header, you're technically able to do it, but only as long as you operate on Windows, which is not what you want as a free human being. An OOP note: For the same reason the SystemsConverterOrbiter functionality has been placed in a separate, derived class. This way, when you're developing some Orbiter-unrelated software based on the Systems package, you don't need to include any big dependencies, such as OrbiterSDK in this case, just to have the definition of the VECTOR3 structure.

MFDSound++ (LGPL)

MFDSound++ is Dan's MFD sound C++ wrapper. If you've ever used Dan's MFD sound library, you'd know that it has an annoying limit of 10 sound samples per registered MFD, which is probably a result of static data structure declaration inside the library. The remedy for this is reserving one or a few sample slots for dynamic loading of the additional samples, after freeing the memory of the slots in question. Assuming that more than 10 samples could lead to lack of memory problems on today's machines, as if just one big sample couldn't, I'd like to only know the limitation, but hide the remedy from the essence of my own code. Therefore I've created a wrapped over the library, which requires you only to define only once all the samples you'll ever need, no matter how many, and just signal the wrapper to play any of these samples, by passing their IDs, and that's it. All you need to remember is that the first 9 samples will not need to be loaded dynamically, therefore the first 9 samples should be the ones that are played the most often.

I encourage you to see the details in the a well documented example, located in Multiple Vessels Support Library.

PID controller (BSD)

A very small basic PID controller class has been also isolated. I've heard of some PIDs floating around the community, but the one that I got my hands on lacked the formal form, and as far as I could see, was simply incorrect. You can use the PID class for example to build your own autopilots using your own creativity. A practical example can be found in the `Autopilot` package. You can also find the following thread very beneficial for controlling a ship in space:

http://orbiter-forum.com/showthread.php?t=23368

Here is the summary of this conversation:

Assume that you are in space, and have your current ship's attitude and the target attitude, both given as `VECTOR3`. The rotation axis between the two vectors is perpendicular to them, so can be obtained by calling `cross(v1, v2)` (cross product) and by normalising the result. Orbiter core would accept the rotation axis as VECTOR3 and fire the RCS into the target direction at full power. The method for this is `VESSEL::SetAttitudeRotLevel( const VECTOR3 & th ) const`. What you need now is to obtain the magnitude of the rotational acceleration (power), and here's what you need the PID for. The PID needs the definition of current error, that is generally the difference between what you need to have and what you currently have → error. Unlike *evilfer*, I'm convinced that it's enough for any PID to operate on floating point data (for input/output), while *evilfer* chose to operate on vectors for this purpose. So what's the error in this case? It's simply the angle between the two vectors. You may remember, that I have a method in `VectorMath` class just for that. I also think that it's a nice practice to limit the error to a range of (-1, 1), only if it's possible to tell the maximal and minimal values of the error. For an angle, using `atan2` function, we get the result in a range of (-180°, 180°), therefore you can divide the error by Π (180°). What you need to do now, is to adjust the PID's gains so that the ship reaches the target relatively quick, and without oscillations. Picking the gains is a separate subject tough. For now, let's assume, that you know how PID's gains work, to finish up one thing. Remember that we've normalised the rotation axis? This means that its length is 1, which for Orbiter core will mean full power in the given direction. The PID must be adjusted so that it keeps shortening the length as the vessel reaches the given attitude, therefore it's wise to reserve the output of 1 to situations like moving away from target, and use output values significantly smaller than 1, like 0.7, for situations like moving towards the target. So assuming that the PID returns a value of 0.4, you must shorten the rotation axis `VECTOR3`, by multiplying it by the returned power. Of course, you can't guarantee, that the output power will always be smaller than 1. Sometimes it may exceed it. What happens if the rotation axis vector's length becomes greater than 1? Orbiter core will rotate the vessel in a very ugly, desynchronised manner. What you need, is to make sure, that if the power is greater than 1, then it stays equal to 1.

Let's return to the discussion about how the PID works. The PID consists of three parts: Proportional, Derivative and Integral. I can fully explain only the first two parts. In a formal form, the PID's output is defined as:

$$PID = Kp \cdot e + Kd \cdot \frac{de}{dt} + Ki \cdot \int e\, dt$$

where:

Kp – proportional gain

Kd – derivative gain
Ki – integral gain
e – error signal

The specific parts' gains define how much of influence the given error signal variation has on output and together define the value of the output. The proportional part measures the current distance from the required position, the derivative part measures the change of that distance in time (velocity), and the Integral part measures the history of error changes. Now imagine a situation where Kp > 0, Kd = 0 and Ki = 0. This means that the output will be solely the error, multiplied by the Kp. So if the output is used to steer the ship towards the target, on the next frame, the error would be smaller. Because of that, the output will also be smaller. Note however what is the output – it's the acceleration. So even if the output is getting smaller, you are still accelerating towards the target, and not decelerating at all, even if you are closer to the target, because you are in space, where there's no drag to slow your motion. You'd start decelerating only after you've overshot the target and the error sign changed. This would leave you constantly oscillating around the target. This is where the derivative gain is supposed to kick in. Let's make the Kd > 0. Now, if you are moving towards the target, the velocity, defined as (e – e_prev) / dt will give you a small, but negative number. You increase its value by Kd and notice what happens to output: as the derivative part's value is negative, the overall output is smaller than it would be without the derivative part, and this is what makes the PID decelerate gracefully as it approaches the target. There's even some more – imagine that you're moving away from the target. The proportional part reacts blindly to the current error, but the derivative part evaluates to now a positive number, which in effect increases the output, thus breaking the undesired movement faster. These are the things that you need to understand when adjusting the PID. Unfortunately I can't explain the integral part. My colleague explained to me that the integral part helps in reaching the desired position faster, although some oscillations are involved.

Once you've adjusted the PID for the given ship, a problem arises when you want to use the same adjustments for a different ship, which undeniably has different characteristics – the PID adjustments will not work for this ship if its characteristics are much different. What you can do is to compare these characteristics with the characteristics of the ship that the PID has been adjusted for, use the same PID, and adjust its output by the ratio of values of the two sets of characteristics. The characteristics in question are generally:

- power of the machine that acts on a controlled object (RCS)
- inertia of the controlled object (vessel)

In Orbiter both can be combined into one value – it will be the maximal rotational acceleration in all three axes separately (pitch, yaw, bank – so in practice three values with similar meaning). Obtaining the accelerations is a matter of solving the following equation:

$$\varepsilon = \frac{T}{I}$$

where:
T – total torque (momentum of force) of a thruster group
I - momentum of inertia in a given axis

All the detailed math required for this calculation is placed in the `Utils\AngularAcc` class. The characteristics have to be obtained constantly, because as your mass changes with the burned fuel, the momentum of inertia changes along. After obtaining the accelerations, you have to compare them to the reference accelerations, by dividing the reference accelerations by the obtained ones in The resulting ratio should be used as a multiplier of the PID's output value. Remember to artificially limit the output power to 1 after multiplying it by the ratio! A good question would be which of the three obtained ratios (3 axes) should be used as the PID's output multiplier. My guess is that both pitch (oX axis) and yaw (oY axis) could be used with success, but you have to make sure. You have to do it only once though.

12) Disclaimer