ORBITER Programmers's Guide

www.medphys.ucl.ac.uk/~martins/orbit/orbit.html

17 December 2003

Contents

1 Spacecraft design

This section describes how to create a new *vessel class* for Orbiter by writing a *vessel DLL module*. Although it is possible to create simple vessel classes without a custom module, by writing a vessel configuration file, the full potential of Orbiter's custom spacecraft design capabilities can only be realised with a specialised module.

All vessels of a given class share the same DLL module. Orbiter only loads a single instance of the DLL. This means that global variables are shared between all vessels of that class. Do not store data which are specific for individual vessels in global variables, because they can be overwritten by another vessel.

1.1 Vessel module callback functions

Orbiter talks to your vessel module via callback functions. Callback functions are invoked as a result of particular events in the simulation. By implementing callback functions in your module you can react to such events and make your vessel behave in a specific way. Note that you do not need to implement all callback functions. Any callback functions which are not defined in the module are simply skipped by Orbiter. For a list of available callback functions, see section *Vessel callback functions* in the Reference Manual.

1.1.1 Vessel creation and destruction

ovcInit

This function is called whenever a *vessel instance* of your vessel class is created. It allows the module to perform all necessary initialisation steps to create the new vessel. A module should always implement this function, and should normally create an instance of the VESSEL interface class (see next section) or a derived class and return a pointer to it.

```
#include "orbitersdk.h"

DLLCLBK VESSEL *ovcInit (OBJHANDLE hvessel, int flightmodel)
{
    return new VESSEL (hvessel, flightmodel);
}
```

The VESSEL constructor requires two parameters, the vessel handle and flight model level, which are both passed by ovcInit.
The VESSEL instance is your interface to the vessel, and most other callback functions will return a pointer to it to provide access.

ovcExit

This function is called before the vessel is destroyed. It should be used for cleanup operations, including the destruction of the VESSEL instance created in ovcInit. In its simplest version it would look like this:

```
DLLCLBK void ovcExit (VESSEL *vessel)
{
    delete vessel;
}
```

1.1.2 Reading and saving vessel states

ovcLoadStateEx

Whenever a simulation is started, Orbiter loads the current status of all vessels from a scenario file. The scenario contains all information required to completely define the status of a vessel at a given time (its position, velocity, thruster levels, fuel levels, etc.) Most modules will need to save and load specific parameters of their own, which are not recognised by Orbiter's generic scenario parser. For this purpose, Orbiter will call the ovcLoadStateEx callback function to allow the module to process its own scenario data.

If the module does not require any non-standard status parameters, ovcLoadStateEx need not be defined. Orbiter will then automatically parse its own generic data. For a list of generic vessel data in a scenario file, see section *Scenario files* in the Orbiter User Manual.

If the module does implement ovcLoadStateEx, it should define a loop which reads lines from the scenario by using the oapiReadScenario_nextline function. Any lines not recogised by the module should be passed on to Orbiter by using the VESSEL::ParseScenarioLineEx function, to allow initialisation of generic data.

This is a typical implementation of ovcLoadStateEx:

```
DLLCLBK void ovcLoadStateEx (VESSEL *vessel, FILEHANDLE scn, void *vs)
{
    char *line;
    int my_value;

    while (oapiReadScenario_nextline (scn, line)) {
        if (!strnicmp (line, “my_option”, 9)) {
            sscanf (line+9, “%d”, &my_value);
        } else if (...) { // more items
            ...
        } else { // anything not recognised is passed on to Orbiter
            vessel->ParseScenarioLineEx (line, vs);
        }
    }
}
```

The vs parameter passed by ovcLoadStateEx points to a VESSELSTATUSx struct ($x \geq 2$). Currently this VESSELSTATUS2, but this may change in future versions to incorporate additional vessel properties. You don’t need to worry about a change in the interface provided you don’t use vs for anything else than passing it on to ParseScenarioLineEx. Even if the VESSELSTATUS interface changes, your module will still remain valid without re-compilation.

There is an older version of this function available, ovcLoadState (and corresponding ParseScenarioLine). This uses the original VESSELSTATUS interface (version 1). It can still be used, but is mainly provided for backward compatibility. This interface doesn’t make use of the latest vessel capabilities, so should be avoided for new modules.

ovcSaveState
When the simulation is closed, or when the user saves by pressing Ctrl-S, a scenario file is written which contains the current simulation status, so that the simulation can be resumed from the current position. Whenever a vessel must save its state in a scenario, Orbiter will call the ovcSaveState callback function to allow the module to save any module-specific parameters. The programmer is responsible to match up the ovcSaveState and ovcLoadStateEx implementations, i.e. to make sure any parameters written by ovcSaveState can be parsed back in by ovcLoadStateEx.
ovcSaveState is not required if the vessel doesn’t need to save any specific data.
To allow Orbiter to save its generic state data, VESSEL::SaveDefaultState should be called from within ovcSaveState. For example:

```
DLLCLBK void ovcSaveState (VESSEL *vessel, FILEHANDLE scn)
{
    vessel->SaveDefaultState (scn); // write all generic data

    oapiWriteScenario_int (scn, “my_option”, my_value);
    ... // more items
}
```

The oapiWriteScenario_int, oapiWriteScenario_float, oapiWriteScenario_vec, and oapiWriteScenario_string functions provide a convenient way to write parameters to the scenario.

1.2 Creating engines

To propel your ship in space, you must equip it with engines. There exist a variety of different rocket engine types, such as liquid and solid fuel engines, or more futuristic ones such as ion or photon drives.

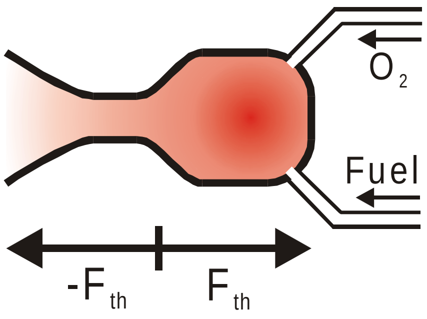
1.2.1 A bit of theory

Thrust force

Despite their very different design, all engines work by the same principle: generating a thrust force in one direction by expelling particles in the opposite direction at high velocity. A liquid-fuel engine, for example, consists of a burn chamber in which a mixture of propellant and oxydiser are ignited, and a nozzle through which the expanding gas is forced at high velocity. The force F_{th} generated by the engine is proportional to the propellant mass flow dm/dt and the velocity v_0 of the expelled gas:

$$\vec{F}_{th} = \frac{dm}{dt}(t)\vec{v}_0$$

When creating a thruster, you need to specify the maximum force F_{th} it can generate when it is driven at full power, and the propellant exit velocity v_0. (in Orbiter, v_0 is called the *fuel-specific impulse*, or Isp). The Isp value determines how much fuel per second is consumed to obtain a given thrust force. The higher the Isp value, the more fuel-efficient the engine.



Note: In Orbiter, the thrust is specified as a force, and has units of Newton [1N = 1kg m s^{-2}]. In the literature, thrust is often specified in units of kg. To convert such data into Orbiter units, multiply by 1g = 9.81 m s^{-2}. Isp is specified as a velocity in Orbiter, with units of m s^{-1}. In the literature it is often given in units of seconds [s]. To convert to Orbiter units, again multiply by 1g.

How long will my fuel last?

The burn time T_b at full thrust F_{max} for fuel mass m_F is given by

$$T_b = \frac{m_F\, Isp}{F_{\max}}$$

Pressure-dependent thrust efficiency

Most conventional rocket engines work less efficiently in the presence of ambient atmospheric pressure, because the ignited gas must be expelled through the nozzle against the outside pressure of the atmosphere. This leads to a reduction of the thrust force at ambient pressure p:

$$F(p) = F_0 - pA$$

where F_0 is the vacuum thrust rating and A has units of an area [m^2] and can be regarded as the *effective nozzle cross section*. If we know the force F_1 generated at ambient pressure p_1, then

$$F_1 = F_0 - p_1 A \quad \Rightarrow \quad A = \frac{F_0 - F_1}{p_1}$$

and therefore

$$F(p) = F_0 - p\frac{F_0 - F_1}{p_1} = F_0\left(1 - p\frac{F_0 - F_1}{F_0 p_1}\right) = F_0(1 - p\eta)$$

and likewise

$$Isp(p) = Isp_0(1 - p\eta)$$

In the literature, the pressure-dependency of engine thrust is often defined by specifying the Isp value for both vacuum and a given reference pressure (e.g. atmospheric pressure at sea level). Orbiter uses the same convention to apply pressure-dependency.

Thrust level

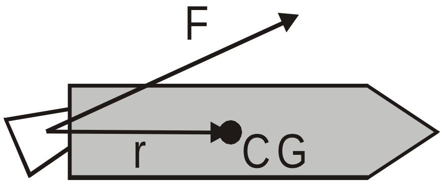
In Orbiter, thrusters can be driven at any level L between 0 (cutout) and 1 (full thrust). The actual thrust force generated by the engine is thus calculated as

$$F(p) = F_{\max}(p) \cdot L$$

In reality, thrusters can often only be driven at maximum, or within a limited range below maximum. This is not currently implemented in Orbiter, but may be introduced in a future version.

Thruster placement and thrust direction

The effect of a thruster depends on its placement on the vessel, and the direction in which the thrust force is generated. In the most general case, a thruster will produce both a linear acceleration (due to a force) and an angular acceleration (do to torque).
Torque is generated if the force vector does not pass through the vessel's centre of gravity (CG)



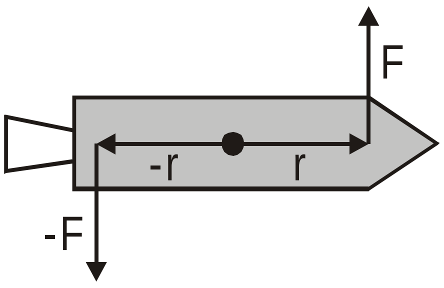
The torque is then given by the cross product

$$\vec{M} = \vec{F} \times \vec{r}$$

(remember that Orbiter uses a left-handed coordinate system!) To avoid uncontrollable spin you should design your ship's main engines so that their force vector passes through the CG. Vessel coordinates are always defined so that the CG is at the origin (0,0,0). Therefore, a thruster located at (0,0,-10) and generating thrust in direction (0,0,1) would not generate torque.

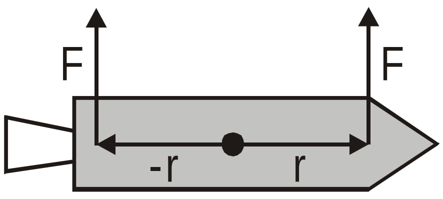
Attitude thrusters: Rotation

Sometimes generating torque is desired in order to rotate the spacecraft. For controlled attitude manouevres one then usually wants to change *only* the angular moment, without also inducing a linear acceleration. This requires the simultaneous operation of at least 2 thrusters so that their linear moments cancel.



Attitude thrusters: Translation

In order to provide small linear accelerations in various directions (for example, to line the ship up with the docking port of a space station), thrusters must be driven single or in groups so that they don't generate torque. Sometimes it is possible to re-use the rotational attitude thrusters for this task, but it is equally possible to add separate linear thrusters.



Engine gimbal and thrust vectoring
Using attitude thrusters in a launch vehicle during the burn phase of the main engines is usually not practical. Instead, attitude control is performed by tilting the main engines and thereby generating a torque as described above. In practice this may be done by suspending the engines in a gimbal system which allows rotation around one or two axes. In Orbiter, this can be implemented by modifying the thrust direction of the engine.
Another way to change the thrust direction is by inserting deflector plates into the exhaust stream.

Torque, angular momentum and angular velocity
The relationship between torque M and angular velocity is given by Euler's equations for a rotating rigid body:

$$J_x\dot{\omega}_x = M_x - (J_z - J_y)\omega_y\omega_z$$
$$J_y\dot{\omega}_y = M_y - (J_x - J_z)\omega_z\omega_x$$
$$J_z\dot{\omega}_z = M_z - (J_y - J_x)\omega_x\omega_y$$

where (J_x, J_y, J_z) are the principal moments of the inertia tensor (PMI), (M_x, M_y, M_z) are the components of the torque tensor, and (ω_x, ω_y, ω_z) are the angular velocity components around the x, y, and z-axes. In Orbiter, this system of differential equations is solved by a trapeziod rule.

1.2.2 Putting it all into the module

Now that you know how thrusters work, it is time to add a few to your new ship. As with other vessel capabilities, thrusters should usually be designed in the ovcSetClassCaps callback function, for example like this (assuming that MyVessel is a class derived from VESSEL):

```
void MyVessel::SetClassCaps (FILEHANDLE cfg)
{
    // vessel caps definitions
}

DLLCLBK void ovcSetClassCaps (VESSEL *vessel, FILEHANDLE cfg)
{
    ((MyVessel*)vessel)->SetClassCaps (cfg);
}
```

Propellant resources
Thrusters can only be operated if they are connected to propellant resources (e.g. fuel tanks). To create a propellant resource:

```
class MyVessel: public VESSEL
{
    ...
    PROPELLANT_HANDLE ph_main;
}

void MyVessel::SetClassCaps (FILEHANDLE cfg)
{
    ...
    const double MAX_MAIN_FUEL = 1e5;
    ph_main = CreatePropellantResource (MAX_MAIN_FUEL);
    ...
}
```

which creates a fuel tank of capacity 10^5kg. CreatePropellantResource returns a handle to the new tank, which is used later to connect thrusters to the tank.
CreatePropellantResource accepts two further optional parameters: the initial fuel mass, and a fuel efficiency factor *eff* between 0 and 1. By default, the tank is full, with fuel efficiency 1. If an *eff* < 1 is specified, then the thrust force generated by all connected thrusters is modified by

$$F' = F \cdot eff$$

Creating thrusters
To add a new thruster, use the CreateThruster command:

```
class MyVessel: public VESSEL
{
    ...
    THRUSTER_HANDLE th_main;
}

void MyVessel::SetClassCaps (FILEHANDLE cfg)
{
    ...
    const double MAX_MAIN_THRUST = 2e5;
    const double VAC_MAIN_ISP = 4200.0;
    th_main = CreateThruster (_V(0,0,-8), _V(0,0,1), MAX_MAIN_THRUST,
                              ph_main, VAC_MAIN_ISP);
    ...
}
```

This adds a thruster at position (0,0,-8) with a thrust vector in the positive z-direction, with the specified max. thrust and Isp values, and connected to the tank we added earlier. In this configuration, the engine efficiency is assumed not to be affected by atmospheric pressure. For increased realism, we could introduce pressure-dependency by adding an additional Isp value at a reference pressure, and the reference pressure itself:

```
void MyVessel::SetClassCaps (FILEHANDLE cfg)
{
    ...
    const double MAX_MAIN_THRUST = 2e5;
    const double VAC_MAIN_ISP = 4200.0;
    const double NML_MAIN_ISP = 3500.0;
    const double P_NML = 101.4e3;
    th_main = CreateThruster (_V(0,0,-8), _V(0,0,1), MAX_MAIN_THRUST,
                              ph_main, VAC_MAIN_ISP, NML_MAIN_ISP, P_NML);
    ...
}
```

This reduces the Isp value at sea level to 3500 and performs a linear interpolation to obtain the Isp at arbitrary pressures. Note that we could have omitted the last parameter, P_NML, because the reference pressure defaults to 101.4 kPa (atmospheric pressure at Earth sea level).
If you descend into a very dense planetary atmosphere, Orbiter will exprapolate the Isp value beyond sea level pressure, until Isp drops to zero. At this point, the thruster will stop working altogether.

Grouping thrusters
Although it is possible to address thrusters individually in your module, it is often easier to engage them in groups. Groups are also required to activate manual user thruster control via the keyboard or joystick, and the automatic navigation modes such as *killrot*.

Orbiter has a number of standard thruster groups, such as THGROUP_MAIN, THGROUP_RETRO, THGROUP_HOVER, and a full set of attitude thruster groups. For a full listing, see VESSEL::CreateThrusterGroup in the Reference Manual.
It is the responsibility of the vessel designer to make sure that thrusters are grouped in a sensible way. For example, whenever the user presses the "+" key on the numerical keypad, all thrusters in THGROUP_MAIN will fire. If the thrusters grouped in THGROUP_MAIN behave in an unexpected or non-intuitive way it will be confusing to the user. Furthermore, if attitude thrusters are not appropriately grouped, some or all of the navigation modes may fail.
To group thrusters, use the CreateThrusterGroup command:

```
void MyVessel::SetClassCaps (FILEHANDLE cfg)
{
    ...
    thg_main = CreateThrusterGroup (th_main, 2, THGROUP_MAIN);
    ...
}
```

(this assumes that th_main is an array of 2 thruster handles which have been created previously). The function returns a handle to the group which can be used later to address the group.
Apart from the standard groups, Orbiter allows to create custom groups by using the THGROUP_USER label. Custom groups are not engaged by any of the standard manual or automatic control methods, therefore the module must implement a suitable control interface for these groups.

1.2.3 Defining exhaust flames

When you define a thruster with CreateThruster, Orbiter will not automatically generate visuals for the exhaust flames when the thruster is engaged. Sometimes exhaust flames may not be appropriate, or, more importantly, you may want to detach the *logical* thruster definition from the *physical* definition (more about this below).

To create an exhaust flame definition use the AddExhaust function. AddExhaust comes in two flavours:

- `UINT AddExhaust (THRUSTER_HANDLE th, double lscale, double wscale, SURFHANDLE tex = 0) const`
- `UINT AddExhaust (THRUSTER_HANDLE th, double lscale, double wscale, const VECTOR3 &pos, const VECTOR3 &dir, SURFHANDLE tex = 0) const`

Both versions require a handle to the logical thruster they are linked to, and two size parameters (longitudinal and transversal scaling), but while the first version takes exhaust location and direction directly from the thruster definition, the second version gets location and direction passed as parameters.

Here is an example demonstrating how you would use the second version of AddExhaust: Let's assume you build a rocket propelled by 4 main engines arranged in a regular square pattern. The engines have fixed orientation (no individual gimbal mode) and all thrust force vectors are parallel. In addition, the engines produce identical thrust magnitudes at all times. Then the 4 engines can be represented by a single logical thruster, whose magnitude is the sum of the 4 actual engines, and positioned in the geometric centre. This simplifies the code, and is more efficient, because Orbiter does not need to add up 4 individual force vectors. However, you still want to see exaust flames for each of the 4 engines, so you would use the second version of AddExhaust to define 4 exhaust flames at the correct positions.

The disadvantage of the second version is that changes in the position or orientation of the thruster (for example as a result of SetThrusterPos or SetThrusterDir) are not automatically propagated to the exaust flames. Therefore, if you plan to move or tilt the thrusters, you should create them individually and use the first version of AddExhaust.

Custom exhaust textures

By default, Orbiter uses a standard texture to render exhaust flames. If you want to customise the exhaust appearance on a per-thruster basis, you can pass a nonzero surface handle tex to both of the AddExhaust versions. To obtain a surface handle for a custom texture, use the oapiRegisterExhaustTexture function.

```
...
SURFHANDLE tex = oapiRegisterExhaustTexture (“MyExhaust”);
AddExhaust (th_main, 10, 2, tex);
...
```

The texture file must be stored in DDS format in Orbiter’s default texture directory. Note that oapiRegisterExhaustTexture can be safely called multiple times with the same texture.

1.3 Rendering re-entry flames

To visualise the friction heat dissipation during atmospheric reentry, Orbiter supports the rendering of “re-entry flames”. To calculate the amount of heat generated per surface area and time (and to scale the exhaust flames) Orbiter uses this formula:

$$P = \frac{1}{2}\rho v^3$$

where ρ is the atmospheric density, and v is the vessel’s airspeed. Orbiter renders exhaust flames if $P > P_0$ where P_0 is a user defined limit. The size and opacity of the reentry flames is scaled by

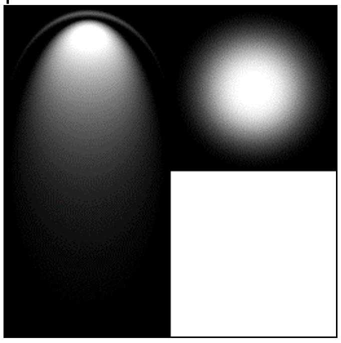
$$s = \min\left(1, \frac{P - P_0}{5P_0}\right)$$

In addition, the user can specify scaling factors for length and width of the reentry texture, as well as the texture itself.

Orbiter by default uses its own texture to render reentry flames. If you want to change the texture globally, you need to replace reentry.dds in the Textures subdirectory. If you only want to modify the texture for a specific vessel class, you need to load a custom texture, and then set your render options:

```
ovsSetClassCaps (VESSEL *vessel, FILEHANDLE cfg)
{
  ...
  SURFHANDLE tex = oapiRegisterReentryTexture (“MyReentryFlame”);
  vessel->SetReentryTexture (tex, my_plimit, my_lscale, my_wscale);
  ...
}
```

Reentry textures require a specific layout. They consist of an elongated part in the left half of the texture map, and a circular part in the upper right corner. The lower right corner is not currently used. This is how the alpha channel of the default reentry.dds looks like:



Note that Orbiter automatically adds a colour component to the texture depending on the value of s, from red to white. If this is sufficient for your custom reentry flame, leave the RGB channels of the texture pure white. Otherwise you may want to experiment with additional texture colours.

If you want to suppress rendering of reentry flames for your vessel altogether, use

```
...
SetReentryTexture (NULL);
...
```

1.4 Adding particle streams for exhaust and reentry effects

Orbiter supports *particle streams* for rendering contrails, exhaust gases, reentry plasma trails etc. Particle streams consist of a series of textured “billboard” objects which always face the camera. The streams can be customised with a set of parameters and allow the simulation of a variety of effects.

The PARTICLESTREAMSPEC structure

At creation, the particle stream can be customised by passing a PARTICLESTREAMSPEC structure to VESSEL::AddExhaustStream and VESSEL::AddReentryStream. The structure is defined as follows:

```
typedef struct {
       DWORD flags;
       double srcsize;
       double srcrate;
       double v0;
       double srcspread;
       double lifetime;
       double growthrate;
       double atmslowdown;
       enum LTYPE { EMISSIVE, DIFFUSE } ltype;
       enum LEVELMAP { LVL_FLAT, LVL_LIN, LVL_SQRT, LVL_PLIN, LVL_PSQRT }
           levelmap;
       double lmin, lmax;
       enum ATMSMAP { ATM_FLAT, ATM_PLIN } atmsmap;
       double amin, amax;
       SURFHANDLE tex;
} PARTICLESTREAMSPEC;
```

`srcrate`

The (average) rate at which particles are created by the emission source [Hz].

`v0`

The (average) emission velocity of particles by the emission source [m/s]

`ltype`

Defines the material lighting method when rendering the particles.

`EMISSIVE`:	Particles are rendered emissive (self-radiating). This is appropriate for streams representing ionized exhaust gases, or plasma streams during reentry.
`DIFFUSE`:	Particles are rendered diffuse (diffuse reflection of external light sources). This is appropriate for smoke and vapour trails.

`levelmap`

Defines the mapping between the level parameter *L* (e.g. thruster level) and the alpha value α (opacity) of the generated particle. The higher the alpha value, the more solid the stream will appear. This parameter is only used for exhaust streams. The following options are available:

`LVL_FLAT`:	constant mapping, i.e. alpha is independent of th reference level: $\alpha = \mathrm{lmin}$
`LVL_LIN`:	linear mapping: $\alpha = L$
`LVL_SQRT`:	square root mapping: $\alpha = \sqrt{L}$

`LVL_PLIN`: linear mapping in sub-range: $\alpha = \begin{cases} 0 & \text{if } L < \text{lmin} \\ \dfrac{L - \text{lmin}}{\text{lmax} - \text{lmin}} & \text{if } \text{lmin} \le L \le \text{lmax} \\ 1 & \text{if } L > \text{lmax} \end{cases}$

`LVL_PSQRT`: square root mapping in sub-range: $\alpha = \begin{cases} 0 & \text{if } L < \text{lmin} \\ \sqrt{\dfrac{L - \text{lmin}}{\text{lmax} - \text{lmin}}} & \text{if } \text{lmin} \le L \le \text{lmax} \\ 1 & \text{if } L > \text{lmax} \end{cases}$

`lmin, lmax`

Defines min and max levels for alpha mapping. Only used if `levelmap` is `CONST`, `PLIN` or `PSQRT` (see above). For `CONST`, only `lmin` is used. For `PLIN` and `PSQRT`, `lmin` < `lmax` is required. Note that `lmin` < 0 is valid – this will cause the stream to produce particles even when the reference level is 0. Likewise, `lmax` > 1 is valid – this will cause the alpha value of the particles to remain < 1 even at reference level 1.

`atmsmap`

Defines the mapping between atmospheric parameters and the alpha value α (opacity) of the generated particle. The following options are available:

`ATM_FLAT`: constant mapping, i.e. alpha is independent of atmospheric parameters: $\alpha = \text{amin}$

`ATM_PLIN`: linear mapping of ambient atmospheric parameter x:

$$\alpha = \begin{cases} 0 & \text{if } x < \text{amin} \\ \dfrac{x - \text{amin}}{\text{amax} - \text{amin}} & \text{if } \text{amin} \le x \le \text{amax} \\ 1 & \text{if } x > \text{amax} \end{cases}$$

`ATM_PLOG`: logarithmic mapping of ambient atmospheric parameter x:

$$\alpha = \begin{cases} 0 & \text{if } x < \text{amin} \\ \dfrac{\ln x/\text{amin}}{\ln \text{amax}/\text{amin}} & \text{if } \text{amin} \le x \le \text{amax} \\ 1 & \text{if } x > \text{amax} \end{cases}$$

For exhaust streams, atmospheric parameter x is the ambient atmospheric density, ρ. For reentry streams, x is defined as $x = \frac{1}{2}\rho v^3$ (v: airspeed) which is proportional to the friction power in turbulent airflow (omitting geometry-related parameters).

`amin, amax`

Defines min and max atmospheric parameter (ambient density or friction power) for alpha mapping. `amin` < `amax` is required. For `PLIN`, `amin` < 0 is admissible to enable particle generation at zero density. For `PLOG`, `amin` > 0 is required.

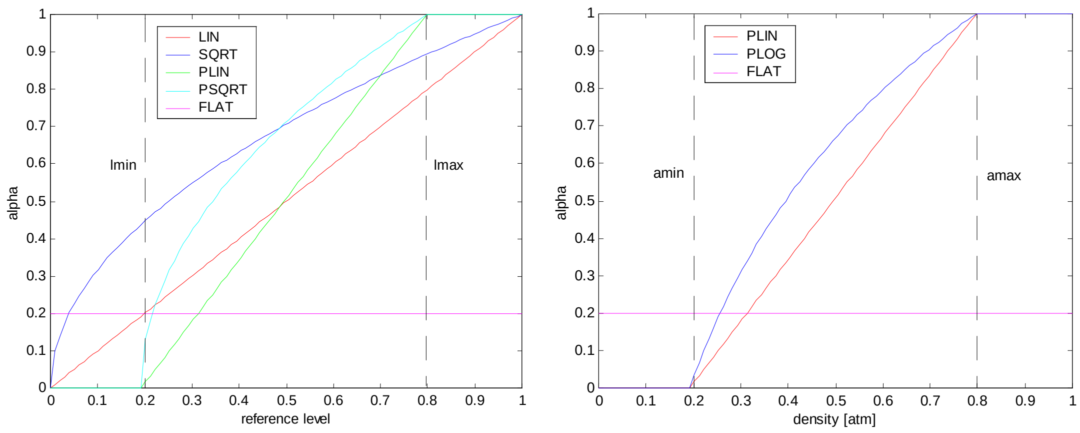


Figure 1: The particle alpha value as a function of reference level (left) and atmospheric parameter (right) for different 'levelmap' and 'atmsmap' modes.

1.5 Atmospheric flight model

1.5.1 Lift and drag theory

Drag is a force acting on the vessel in the direction of the freestream airflow. It is composed from several components:

1. The *skin friction drag* caused by the boundary layer surrounding the airfoil.
2. The *pressure drag* caused by separation of flow from the surface.
3. The *wave drag* at supersonic velocities.
4. *Induced drag*, caused by airflow around the wingtip (finite wing) from the lower to the upper surface.

The combination of components 1-3 is defined as *profile drag* or *parasite drag*.

Lift is an upward force (perpendicular to the airflow) caused by the shape of the airfoil and its orientation to the airflow.

Drag D and lift L of an airfoil are expressed by the drag and lift coefficients c_D and c_L, with

$$c_D = \frac{D}{q_\infty S}, \quad c_L = \frac{L}{q_\infty S}$$

where $q_\infty = \frac{1}{2}\rho_\infty V_\infty^2$ is the freestream dynamic pressure, and S is the wing area. Generally, c_D and c_L, will be functions of the angle of attack, the Mach number, and the Reynolds number. We now split c_D in the components of profile and induced drag. Induced drag is a result of lift and can be expressed as a function of c_L:

$$c_D = c_{D,e} + \frac{c_L^2}{\pi e A}$$

where e is a span efficiency factor, and A is the wing aspect ratio, defined as b^2/S with wing span b.

The profile component $c_{D,e}$ will change with angle of attack. We assume that $c_{D,e}$ can be expressed as the combination of a zero-lift component $c_{D,0}$ and a component depending on c_L:

$$c_{D,e} = c_{D,0} + r c_L^2$$

Here, r is a form constant which is usually determined empirically. We can now incorporate the lift-dependent term of $c_{D,e}$ into the factor e, to give

$$c_D = c_{D,0} + \frac{c_L^2}{\pi \varepsilon A}$$

where $\varepsilon = e/(r\pi e A + 1)$ is the *Oswald efficiency factor*.

When implementing an airfoil in Orbiter, the user must supply a function which calculates c_L and c_D for a given set of parameters (angle of attack, Mach number and Reynolds number). Orbiter provides a helper function (oapiGetInducedDrag) to calculate the induced drag component with the above formula.

1.5.2 Lift and drag in transonic and supersonic flight

(to be completed)

1.5.3 Angular moments and vessel stability

(to be completed)

1.5.4 Angular drag

Similar to (linear) drag which produces a force acting against a vessel's airspeed vector, a rotating vessel will experience angular drag which acts against the angular velocity, thus slowing the rotation. Orbiter uses the following formulae to calculate angular damping:

$$dM_x = -q' S_y c_{\alpha,x} \omega_x$$

$$dM_y = -q' S_y c_{\alpha,y} \omega_y$$

$$dM_z = -q' S_y c_{\alpha,z} \omega_z$$

where $q' = \frac{1}{2}\rho_\infty (V_\infty + V_0)^2$ is a modified dynamic pressure which ensures that angular drag also occurs at low airspeeds (Orbiter currently uses a fixed $V_0 = 30$ m/s). S_y is the vessel's cross section projected along the vertical (y) axis, used as a reference area. S_y is the y-component of the vector passed to VESSEL::SetCrossSections(). $c_{\alpha,x}$, $c_{\alpha,y}$ and $c_{\alpha,z}$ are the

drag coefficients for rotations around the x, y, and z vessel axes as defined by VESSEL::SetRotDrag(). ω_x, ω_y and ω_z are the angular velocities around the vessel axes, and dM_x, dM_y and dM_z are the changes in torque due to damping.
Angular drag is determined by the vessel shape. Developers can adjust the effect of angular damping in the atmosphere by adjusting the coefficients passed to VESSEL::SetRotDrag(). Higher coefficients make a vessel less responsive to control input, and reduce oscillations around equilibrium orientation.

1.5.5 API interface for airfoil definitions

To define the lift and drag characterisitics for a spacecraft in the DLL module, use the VESSEL::CreateAirfoil method. An airfoil is defined as a cross section through a wing. In Orbiter, we use the term airfoil for any components of the vessel which produce lift and/or drag forces. Multiple airfoils can be defined for a single vessel (for example for the left and right wing, the body, the horizontal and vertical stabilizers in the tail, etc.). It is usually best to keep the number of airfoils low to keep the flight model predictable and to improve simulation performance.

Orbiter distinguishes two different types of airfoil orientations: airfoils which create vertical lift (e.g. wings) and airfoils which create horizontal "lift", e.g. vertical stabilisers. Even vessels without any wings or other aerodynamic surfaces should define at least one horizontal and one vertical airfoil to define their atmospheric drag behaviour (even blunt objects such as reentry capsules which have no similarity to an aircraft produce drag *and* lift forces).

When calling the CreateAirfoil method, the user must provide

- basic airfoil parameters (orientation, wing area, chord length and wing aspect ratio).
- the force attack point (i.e. the point on the vessel on which the lift and drag forces for this airfoil act). This influences the angular momentae generated by the forces.
- a callback function which calculates the lift, drag and moment coefficients of the airfoil as a function of angle of attack α, Mach number M and Reynolds number Re.

The coefficients decide how much lift and drag is generated by the airfoil. The lift and drag forces (L and D) are obtained from the moments (c_L and c_D) by

$$L(\alpha, M, \mathrm{Re}) = c_L(\alpha, M, \mathrm{Re}) q_\infty S$$
$$D(\alpha, M, \mathrm{Re}) = c_D(\alpha, M, \mathrm{Re}) q_\infty S$$

with freestream dynamic pressure $q_\infty = 1/2\, \rho v^2$, and reference area S. The function which calculates c_L and c_D must be able to handle arbitrary angles of attack ($-\pi$ to π) and very high Mach numbers which may occur during LEO insertion and atmospheric entry (orbital velocity for a low Earth orbit is equivalent to $M > 20$!)

The Reynolds number is a parameter dependent on atmospheric viscosity μ:

$$\mathrm{Re} = \frac{\rho v c}{\mu}$$

with freestream airspeed v and density ρ. In the current Orbiter version, μ is assumed constant ($\mu = 1.6894 \cdot 10^{-5}$ kg m^{-1} s^{-1}). In future versions, μ will depend on the atmospheric composition and temperature.

The direction of the lift force vector is defined in Orbiter as

$$\hat{L}_\alpha = (0, -v_z, v_y) / \sqrt{v_y^2 + v_z^2}$$
$$\hat{L}_\beta = (-v_z, 0, v_x) / \sqrt{v_x^2 + v_z^2}$$

for vertical and horizontal lift components, respectively, where (v_x, v_y, v_z) is the freestream airflow vector in vessel coordinates. This means that $\hat{L}_\alpha$ is rotated 90° counter-clockwise against the projection of the airflow vector into the yz-plane, and $\hat{L}_\beta$ is rotated 90° counter-clockwise against the projection of the airflow vector into the xz-plane. Since α and β are defined as

$$\alpha = \arctan v_y / - v_z$$

$$\beta = \arctan v_x / - v_z$$

we find the following relations between α or β and the direction of lift:

α	**lift direction**
0°	up (+y)
90°	forward (+z)
180°	down (-y)
270°	backward (-z)

β	**lift direction**
0°	right (+x)
90°	forward (+z)
180°	left (-x)
270°	backward (-z)

This convention must be taken into account when defining the lift coefficient profile. For example, the c_L profile for a vertical stabiliser with symmetric airfoil should be positive for 0° ≤ β ≤ 90° and 180° ≤ β ≤ 270°, and negative for 90° ≤ β ≤ 180° and 270° ≤ β ≤ 360°. The lift profile in this case may therefore resemble $\sin 2\beta$. For asymmetric airfoils the lift profile will look more complicated (for example, the zero-lift angle will usually usually not be exactly 0°).

1.6 Defining an animation sequence

Animation sequences can be used to simulate movable parts of a vessel. Examples are the deployment of landing gear, cargo door operation, or animation of airfoils.
Animations are implemented in *vessel modules*, using the VESSEL interface class.
Orbiter allows 3 types of animation: rotation, translation and scaling. More complex can be built from these basic operations.

1.6.1 Semi-automatic animation

Mesh requirements:
Animations are performed by transforming mesh groups. Therefore, all parts of the mesh participating in an animation must be defined in separate groups. Multiple groups can participate in a single transformation.

Module prerequisites:
If it doesn't exist already, create a C++ project for the vessel module.
Derive a class from VESSEL, e.g.

```
class MyVessel: public VESSEL {
  // ...
};
```

Implement the `ovcInit` and `ovcExit` callback functions to create and destroy an instance of `MyVessel`, e.g.

```
DLLCLBK VESSEL *ovcInit (OBJHANDLE hvessel, int flightmodel)
{
        return new MyVessel (hvessel, flightmodel);
}

DLLCLBK void ovcExit (VESSEL *vessel)
{
        delete (MyVessel*)vessel;
}
```

Defining an animation sequence:
Create a member function for MyVessel to define animation sequences, and call it from the constructor, e.g.

```
MyVessel::MyVessel (OBJHANDLE hObj, int fmodel)
: VESSEL(hObj, fmodel)
{
  DefineAnimations();
}
```

In the body of DefineAnimations(), you now have to specify how the animation should be performed. Here is an example for a nose wheel animation:

```
void MyVessel::DefineAnimations()
{
  static UINT groups[4] = {5,6,10,11}; // participating groups

  static MGROUP_ROTATE nosewheel (
    0,                          // mesh index
    groups, 4,                  // group list and # groups
    _V(0,-1.0,8.5),             // rotation reference point
    _V(1,0,0),                  // rotation axis
    (float)(0.5*PI)             // angular rotation range
  );

  anim_gear = CreateAnimation (0.0);
  AddAnimationComponent (anim_gear, 0, 1, &nosewheel);
}
```

You first need to determine which mesh groups take part in the animation. In this case, the nose wheel consists of the four groups 5, 6, 10 and 11, and these are listed in the “groups” array.

Next, you must define the parameters of the rotation. This is done by creating a MGROUP_ROTATE instance. Besides the mesh index and group indices, this also requires the rotation reference point (i.e. the point around which the mesh groups are rotated), the axis of rotation, and the rotation range.

A new animation is created by calling CreateAnimation. The parameter passed to CreateAnimation defines the animation state in which the mesh groups are stored in the mesh. The return value identifies the animation.

Finally, the rotation of the nose wheel is added to the animation by calling AddAnimationComponent. The parameter are the animation identifier, the cutoff states of the component, and the transformation. The cutoff states define over which part of the animation the component transformation is applied. In this example, the cutoff states are 0 and 1, that is, the rotation of the nose wheel occurs over the full duration of the animation.

Now let’s consider a slightly more complicated example, where the animation consists of two components: (a) opening the wheel well cover, and (b) deploying the gear.

```
void MyVessel::DefineAnimations()
{
  static UINT cover_groups[2] = {0,1};
  static MGROUP_ROTATE cover (0, cover_groups, 2,
    _V(-0.5,-1.5,7), _V(0,0,1), (float)(0.45*PI));

  static UINT wheel_groups[4] = {5,6,10,11};
  static MGROUP_ROTATE nosewheel (0, wheel_groups, 4,
    _V(0,-1.0,8.5), _V(1,0,0), (float)(0.5*PI));

  anim_gear = CreateAnimation (0.0);
  AddAnimationComponent (anim_gear, 0, 0.5, &cover);
  AddAnimationComponent (anim_gear, 0.4, 1, &nosewheel);
}
```

The rotations for the gear well cover and the landing gear are defined by two separate MGROUP_ROTATE variables. After creating the animation, both rotations are added as components. The cover is opened during the first part of the animation (between states 0 and 0.5) while the gear is deployed in the final part (between states 0.4 and 1). Note that there is a

small overlap (between 0.4 and 0.5), which means that the gear begins to rotate before the cover is fully opened.
When the animation is played backward to retract the gear, the components are rotated in the inverse order: the gear is retracted first, then the cover is closed.

Animations can be arranged in a hierarchical order, so that a parent animation can transform mesh groups which are themselves animations. Consider for example the wheel on a landing gear which is spinning while the gear is being retracted. In this case, the gear animation is defined as a rotation around the gear hinge point, while the wheel animation is a rotation around the wheel axis. The wheel animation must be defined as a child of the gear animation, because the wheel is rotated together with the gear.

```
void MyVessel::DefineAnimations()
{
  ANIMATIONCOMPONENT_HANDLE parent;

  static UINT gear_groups[2] = {5,6};
  static MGROUP_ROTATE gear (0, gear_groups, 2,
    _V(0,-1.0,8.5), _V(1,0,0), (float)(0.45*PI));

  static UINT wheel_groups[2] = {10,11};
  wheel = new MGROUP_ROTATE nosewheel (0, wheel_groups, 2,
    _V(0,-1.0,6.5), _V(1,0,0), (float)(2*PI));

  anim_gear = CreateAnimation (0.0);
  parent = AddAnimationComponent (anim_gear, 0, 1, &gear);

  anim_wheel = CreateAnimation (0.0);
  AddAnimationComponent (anim_wheel, 0, 1, wheel, parent);
}
```

The gear and wheel rotations are defined by the MGROUP_ROTATE variables “gear” and “wheel”. Note that in this case “wheel” is not defined static, since reference point and axis will be modified by the parent. Therefore, “wheel” must be defined as a data member of the MyVessel class. Since “wheel” is allocated dynamically, don’t forget to deallocate it with

```
MyVessel::~MyVessel()
{
  ...
  delete wheel;
  ...
}
```

The return value of the AddAnimationComponent() call for the gear animation is a handle which identifies the transformation. We use this value for the optional parent parameter when defining the animation component for the wheel animation. This makes the wheel animation a child of the gear animation.

A complex example for hierarchical animations can be found in the RMS arm animation of Space Shuttle Atlantis in Orbitersdk\samples\Atlantis\Atlantis.cpp.

Apart from rotations, mesh groups can also be transformed by translating and scaling. The corresponding MGROUP_TRANSFORM derivates are MGROUP_TRANSLATE and MGROUP_SCALE:

```
  MGROUP_TRANSLATE t1 (0, groups, 2, _V(0,10,5));
  MGROUP_SCALE t2 (0, groups, 2, _V(5,0,2), _V(2,2,2));
```

In both cases, the first three parameters are the same as for MGROUP_ROTATE (mesh, index, group list and number of groups). For MGROUP_TRANSLATE, the last parameter defines the translation vector. For MGROUP_SCALE, the last two parameters define the scale origin, and the scale factors in the three axes.

Performing the animation:
To animate the nose wheel now, we need to manipulate the animation sequence state by calling SetAnimation() with a value between 0 (fully retracted) and 1 (fully deployed). This is typically done in the Timestep() member function, e.g.

```
void MyVessel::Timestep (double simt)
{
  if (gear_status == CLOSING || gear_status == OPENING) {
    double da = oapiGetSimStep() * gear_speed;
    if (gear_status == CLOSING) {
      if (gear_proc > 0.0)
        gear_proc = max (0.0, gear_proc-da);
      else
        gear_status = CLOSED;
    } else  { // door opening
      if (gear_proc < 1.0)
        gear_proc = min (1.0, gear_proc+da);
      else
        gear_status = OPEN;
    }
    SetAnimation (anim_gear, gear_proc);
  }
}
```

Here, gear_status is a flag defining the current operation mode (CLOSING, OPENING, CLOSED, OPEN). This will typically be set by user interaction, e.g. by pressing a keyboard button. If the animation is in progress (OPENING or CLOSING), we determine the rotation step (da) as a function of the current frame interval (oapiGetTimeStep()). The value of gear_speed defines how fast the gear is deployed.
Next, we update the deployment state (gear_proc), and check whether the sequence is complete (≤0 if closing, or ≥1 if opening). Finally, SetAnimation() is called to perform the animation.

The DeltaGlider sample module (Orbitersdk\samples\DeltaGlider) contains a complete example for an animation implementation.

1.6.2 Manual animation

As an alternative to the (semi-)automatic animation concept described in the previous section, Orbiter also allows manual animation. This can be more versatile, but requires more effort from the module developer, because the complete animation sequence must be implemented explicitly.
A manual animation sequence is created by the functions `VESSEL::RegisterAnimation()` and `VESSEL::UnregisterAnimation()`. A call to `RegisterAnimation` causes Orbiter to call the module's `ovcAnimate` callback function at each frame, provided the vessel's visual exists. `UnregisterAnimation` cancels the request.
Note that `RegisterAnimation/UnregisterAnimation` pairs can be nested. Each call to `RegisterAnimation` increments a reference counter, each call to `UnregisterAnimation` decrements the counter. Orbiter will call `ovcAnimate` as long as the counter is > 0.
It is up to the module to implement its animations in the body of `ovcAnimate`. Typically this will involve calls to `MeshgroupTransform()`, to rotate, translate or scale mesh groups as a function of the last simulation time step. Note that `ovcAnimate` is called only once per frame, even if more than one `RegisterAnimation` request has been logged. The module must therefore decide which animations need to be processed in `ovcAnimate`.
`UnregisterAnimation` should never be called from inside `ovcAnimate`, since `ovcAnimate` is only called if the visual exists. This could cause the unregister request to be lost. It is better to test for animation termination in `ovcTimestep`.

1.7 Designing instrument panels

1.7.1 Defining a panel

In order to implement instrument panel support for your vessel you must implement the ovcLoadPanel callback function:

```
DLLCLBK bool ovcLoadPanel (VESSEL *vessel, int id)
{
  ...
}
```

where `vessel` is a pointer to the VESSEL interface instance for which the panel is to be generated, and `id` is a panel identifier. Orbiter will call this function whenever it needs to load a new panel, for example because the user switched to a different panel, selected a different vessel, or activated panel mode with F8.

If the vessel only supports a single panel, `id` will always be 0. If multiple panels are supported, your callback function must test the value of `id` to determine which panel to load. To implement multiple panels, each of the panel must define its connectivity to neighbouring panels via the `oapiSetPanelNeighbours` function.

Example: If your vessel supports a main panel, an overhead and a left side panel, the structure of ovcLoadPanel would look like this:

```
DLLCLBK bool ovcLoadPanel (VESSEL *vessel, int id)
{
  switch (id) {
  case 0: // main panel
    oapiRegisterPanelBackground (LoadBitmap (hDLL,
       MAKEINTRESOURCE (IDB_PANEL0)));
    oapiSetPanelNeighbours (2, -1, 1, -1);
    // register areas for panel 0 here
    break;
  case 1: // overhead panel
    oapiRegisterPanelBackground (LoadBitmap (hDLL,
      MAKEINTRESOURCE (IDB_PANEL1)));
    oapiSetPanelNeighbours (-1, -1, -1, 0);
    // register areas for panel 1 here
    break;
  case 2: // left side panel
    oapiRegisterPanelBackground (LoadBitmap (hDLL,
      MAKEINTRESOURCE (IDB_PANEL2)));
    oapiSetPanelNeighbours (-1, 0, -1, -1);
    // register areas for panel 2 here
    break;
  }
  return true;
}
```

Each panel must register a background bitmap via the `oapiRegisterPanelBackground` function. The bitmap must be passed in standard Windows HBITMAP format. The easiest way to include a panel bitmap in your vessel DLL is to include it as a bitmap resource so that it can be loaded with the Windows `LoadBitmap` command. The hDLL parameter is the Windows module instance handle. You can obtain it from the DllMain callback function, for example

```
HINSTANCE hDLL; // global: module handle

BOOL WINAPI DllMain (HINSTANCE hModule, DWORD ul_reason_for_call,
  LPVOID lpReserved)
{
   hDLL = hModule;
}
```

If the vessel defines multiple panels, the user can switch between them by using Ctrl-Arrow keys. Orbiter must know the relative location of bitmaps to each other, so that the correct panel can be loaded. This connectivity is provided by the oapiSetPanelNeighbours function. This function tells Orbiter which panels are to the left, right, top and bottom of the current panel. A value of –1 indicates that no panel is located at that side.

Important: All the panel id's defined during `oapiSetPanelNeighbours` must be supported by `ovcLoadPanel`. For example, if panel 0 calls `oapiSetPanelNeighbours (2,-1,1,-1)`, then panels 1 and 2 must be handled by `ovcLoadPanel`.

All panels must call the `oapiSetPanelNeighbours` function, otherwise there is no way for the user to switch back to a different panel. Panel connectivities should usually be reciprocal, i.e. if panel 0 defines panel 1as its top neighbour, then panel 1 should define panel 0 as its bottom neighbour. If only a single panel (panel 0) is supported, calling `oapiSetPanelNeighbours` is not necessary.

`ovcLoadPanel` should return true if the panel was loaded successfully. It should return false if the panel initialisation failed for any reason.

1.7.2 Defining active panel areas

< To be completed >

1.7.3 The mouse event handler

To intercept mouse events generated by a panel you must implement the `ovcPanelMouseEvent` callback function:

```
DLLCLBK bool ovcPanelMouseEvent (VESSEL *vessel, int id, int event, int mx,
int my)
{
  ...
}
```

where `vessel` is a pointer to the VESSEL interface instance for which the mouse event was generated, `id` is the identifier of the panel area for which the event was generated (as specified in oapiRegisterPanelArea), `event` specifies the mouse event type, and `mx`,`my` are the panel coordinates at which the event occured.

To make a panel area generate mouse events, the required events must be defined during the registration of the area. For example, to create an instrument which generates mouse events whenever the left mouse button is pressed, oapiRegisterPanelArea must be defined with the `PANEL_MOUSE_LBDOWN` flag. Mouse bitflags can be combined. If you want to generate an event whenever the left mouse button is pressed or released, use the `PANEL_MOUSE_LBDOWN | PANEL_MOUSE_LBUP` flags.
A panel area defined with `PANEL_MOUSE_IGNORE` will never generate any mouse events.

Important: A button-up event is always generated for the instrument which produced the preceding button-down event, even if the mouse has been dragged out of the panel area in the mean time.

The following mouse events are available:

`PANEL_MOUSE_LBDOWN`	Left mouse button pressed down.
`PANEL_MOUSE_RBDOWN`	Right mouse button pressed down.
`PANEL_MOUSE_LBUP`	Left mouse button released.
`PANEL_MOUSE_RBUP`	Right mouse button released.
`PANEL_MOUSE_LBPRESSED`	Left mouse button down
`PANEL_MOUSE_RBPRESSED`	Right mouse button down.

The `PANEL_MOUSE_LBPRESSED` and `PANEL_MOUSE_RBPRESSED` events are sent continuously while the buttons are held down. This allows the implementation of mouse-dragging event, for example to move sliders with the mouse.

1.7.4 The redraw event handler

< To be completed >

2 Planets and moons

Orbiter allows to create new planets or planetary systems in a few simple steps. To create a new planet, you need to do the following:

- find or create a surface texture map
- optionally, find or create texture maps for a cloud layer, for a land/sea mask, and for night lights
- convert the texture map(s) into ORBITER's .tex format by invoking pltex
- optionally, create surface areas with very high resolution textures by running pltex –9 and using TileManager.
- create a configuration file (.cfg) in the Config subfolder, containing physical and orbital planet parameters.
- Add an entry for the planet in the configuration file of the planetary system (e.g. Sol.cfg).
- Optionally, create a DLL plugin module to allow detailed control of planet movement and atmosphere definition.

2.1 Planet texture maps

2.1.1 Texture format

Each planet has an associated surface texture file <*pname*>.tex, where <*pname*> is the planet's name. Optionally, additional texture files <*pname*>_cloud.tex (for defining a cloud layer), <*pname*>_lmask.tex (for defining a land area mask) and <*pname*>_lights.tex (for defining surface night lights) may be present.
Each texture file contains a series of texture maps, stored as DirectDraw surfaces (dds) in DXT1 compression format.
ORBITER uses a variable resolution approach for both meshes and texture maps to render planetary surfaces. The rendering resolution level is determined by the apparent radius of the planet. At low resolutions ORBITER uses a single spherical mesh covered by a single texture. At higher resolutions the spherical surface is constructed from a series of sphere patches, each containing its own texture patch. This method allows efficient rendering by removing hidden patches before invoking the rendering pipeline.
ORBITER currently supports 9 resolution levels for planetary surfaces, as listed in Table 1. At the highest resolution the sphere is constructed from 364 patches with an effective texture resolution of 16384x8192. Figure 2 shows a detail of the Martian surface rendered at different resolution levels.

Level	Resolution*	Mesh patches	Triangles (total)**	Texture memory***	
				with DXT1	w/o DXT1
1	64 x 64	1	144	2K	16K
2	128 x 128	1	256	10K	80K
3	256 x 256	1	576	42K	336K
4	512 x 256	2	1024	106K	848K
5	1024 x 512	8	2592	362K	2.9M
6	2048 x 1024	24	4672	1.1M	9.0M
7	4096 x 2048	100	25440	4.3M	34.6M
8	8192 x 4096	364	116448	16.0M	127.8M
9	16384 x 8192	1456	276640	63.9M	511.2M

Table 1: Supported resolution levels for planetary surfaces.

*Resolution: Effective texture map resolution at the equator.
**Triangles: This is the total number of triangles for all patches. In practice fewer triangles will be rendered because hidden patches are removed before entering the rendering pipeline.

***Texture memory: Video/AGP memory required to hold texture maps up to this resolution level for a single planet. With DXT1: video hardware supports DXT1 texture compression. W/o DXT1: video hardware doesn't support DXT1 texture compression.

High resolution levels require significant amounts of video/AGP memory and should only be used on systems with adequate 3D graphics subsystems. On older graphics cards which do not natively support DXT1 texture compression ORBITER needs to convert textures into RGBA format which increases memory requirements 8-fold. Conversion to RGBA will also dramatically increase the loading time when starting ORBITER.

Important: Do not try to use resolution level ≥ 8 if your video card does not support DXT1 texture compression or has less than 32MB of texture memory!

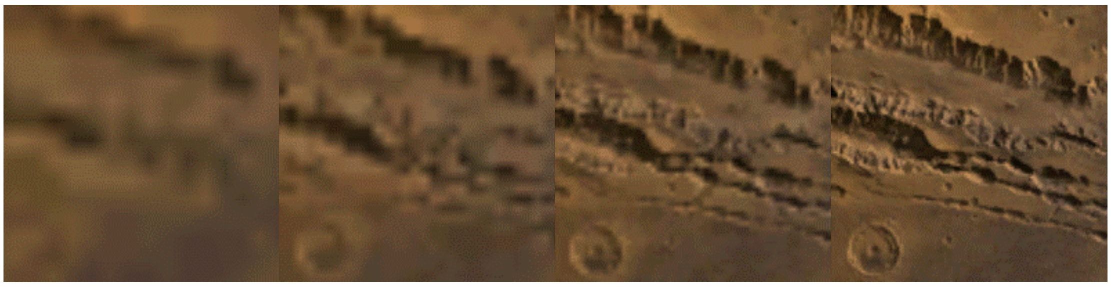


Figure 2: Mars texture detail at resolution levels 5, 6, 7 and 8 (from left).

2.1.2 Where ORBITER looks for textures

ORBITER first searches for the texture file in the location specified by the `HightexDir` entry in the *Orbiter.cfg* file. If the texture file is not found or if `HightexDir` is not defined then ORBITER searches in the directory specified by the `TextureDir` entry. This allows switching between high and low resolution texture maps conveniently by inserting or removing the `HightexDir` entry.
If no texture file is found then the planet is rendered without a surface texture.
Each planet's configuration file <*pname*>.cfg contains an entry `MaxPatchResolution` which defines the maximum texture resolution level to use with this planet (valid range 1 to 8). If the texture file contains higher resolution levels than defined by `MaxPatchResolution` then the additional resolutions are skipped. This allows reducing texture memory requirements without modifying the texture file. If the texture file contains fewer resolution levels than defined by `MaxPatchResolution` then the maximum resolution is reduced accordingly.

2.1.3 Using pltex to generate custom planet textures

If you prefer, you can use your own planet maps instead of those provided by ORBITER. The ORBITER download page contains a planet texture conversion tool (*pltex*) which allows to convert planet maps from BMP bitmap format to ORBITER's texture format. It resamples the map to the requested resolutions, splits it into surface patches and converts them to DXT1 compressed texture format.
The source map should contain the complete surface in spherical projection, where the left edge corresponds to longitude 180° W, the right edge to longitude 180° E, the bottom edge to latitude 90° S, and the top edge to latitude 90° N. The width/height ratio of the bitmap should be close to 2/1.
Pltex requires the source map in 24bit or 8bit Windows BMP format. If your map is in any other format (e.g. JPEG or GIF) you need to convert it into BMP (using your favourite graphics conversion tool) before invoking pltex.

Synopsis:

```
pltex [–i <mapname>] [-l <minres> -h <maxres>] [-9]
```

<mapname>: source texture file name
<minres>: minimum resolution level (1..8)
<maxres>: maximum resolution level (*<minres>*..8)

- If command line options are omitted then pltex requests values interactively.
- If a higher maximum resolution is requested than can be obtained from the source map, pltex adjusts the maximum resolution accordingly. See Table 1 for map resolutions at the various resolution levels.
- The only justification for <*minres*> $\neq$ 1 is if you want to compose certain resolution levels from a different source map, e.g. generate Earth resolution levels 1 to 7 from a map that includes clouds, and level 8 from a map without clouds. In that case pltex must be run twice, and the output texture files concatenated.
- The option to use alpha (transparency) maps is intended for semi-transparent cloud maps.
- You can use pltex to generate a set of level 9 texture patches by specifying the –9 command line option. In that case, both <*minres*> and <*maxres*> must be set to 9. Note that level 9 textures are treated differently to levels 1-8. Level 9 is not automatically assembled into the <planet>.tex file. Instead, after generating the individual patches (1456 in total!) with pltex, you need to run the TileManager application bundled with the orbiter base package to add patches into the <*planet*>_tile.tex file containing high-resolution patches. See the TileManager help file for details.

Pltex will generate a texture file <*mapname*>.tex. If necessary, rename to <*pname*>.tex where <*pname*> is the planet's name, and copy to the `TextureDir` directory (usually "Textures") or `HightexDir` directory (usually "Textures2") (see section **Error! Reference source not found.**).

Note:
Generating high-resolution texture maps (level 8 and higher) may take a long time and requires a large amount of system memory.

2.2 Defining an atmosphere

Planetary atmospheres have a significant influence on the flight behaviour of spacecraft. The primary atmospheric parameters are temperature, pressure and density as a function of altitude.

Defining a simple atmospheric model is possible by setting a few parameters in the planet's configuration file. More sophisticated models must be coded in the planet's DLL module.

Orbiter currently does not model local atmospheric perturbations (climatic/weather effects).

2.2.1 A simple atmosphere

To define a simple exponentially decaying atmosphere, define the following items in the planet's configuration (.cfg) file:

`AtmPressure0`:	The static atmospheric pressure [Pa] at altitude zero, p_0.
`AtmDensity0`:	The atmospheric density [kg/m^3] at altitude zero, ρ_0.
`AtmAltLimit`:	The altitude above which atmospheric effects can be neglected.

where altitude zero is defined as distance `Size` (as defined in the configuration file) from the planet's centre.

The pressure and density at any altitude h is then calculated by Orbiter as

$$p = \begin{cases} p_0 e^{-Ch} & \text{if } h < \text{AtmAltLimit} \\ 0 & \text{otherwise} \end{cases}, \quad \rho = \begin{cases} \rho_0 e^{-Ch} & \text{if } h < \text{AtmAltLimit} \\ 0 & \text{otherwise} \end{cases}$$

where $C = \frac{\rho_0}{p_0} g_0$, and g_0 is the gravitational acceleration at altitude zero.

This model assumes constant temperature.

2.2.2 A more sophisticated atmosphere

Where the simple model described above is not adequate, the planet's DLL module can be used to define the Planet_AtmPrm function, which returns atmospheric parameters as a function of altitude. An example is the implementation of Earth's *standard atmosphere*[1] in Vsop87.dll. This model is commonly used in aviation applications and defines a temperature distribution as shown in Figure 3 (up to altitude 105 km), consisting of sections of constant temperature and sections where temperature varies linearly with altitude.

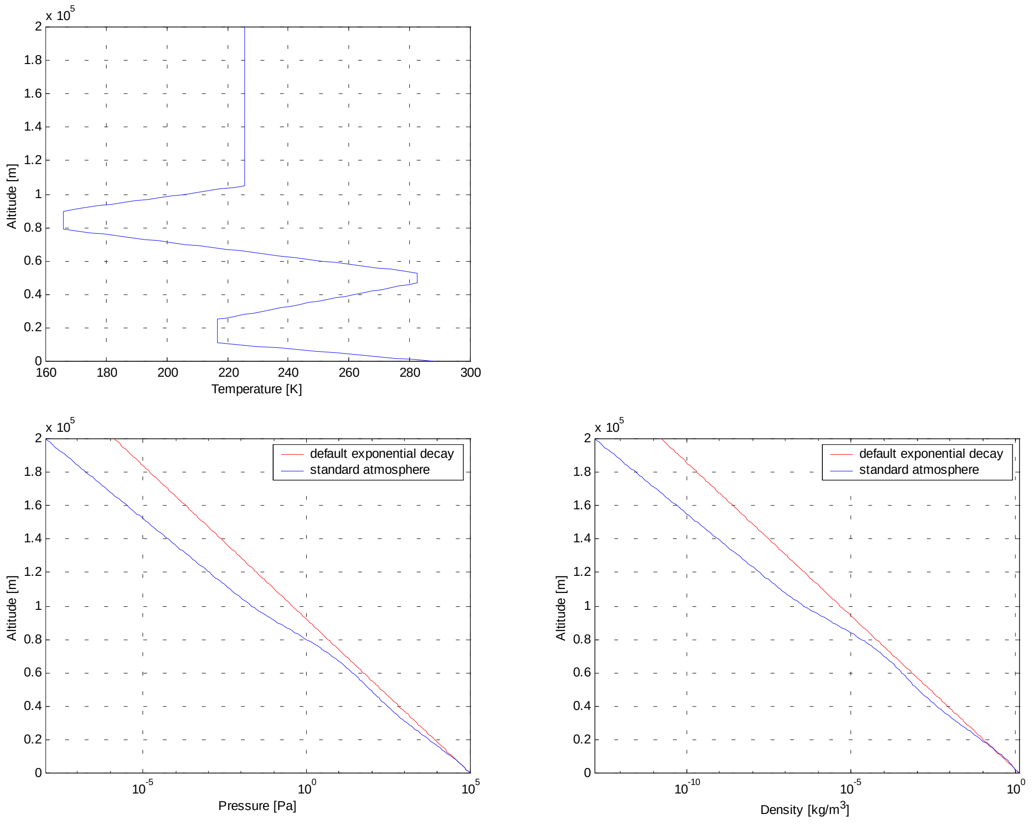


Figure 3: Temperature distribution in Earth's standard model (top left), and comparison between simple exponentially decaying model and standard model for pressure (bottom left) and density (bottom right). Note that the standard model is only defined up to altitude 105 km. Above this, Orbiter assumes constant temperature.

The pressure and density in the sections of constant temperature is calculated as

$$p(h) = p_1 e^{-[g_0/(RT)](h-h_1)}, \quad \rho(h) = \rho_1 e^{-[g_0/(RT)](h-h_1)}$$

where h_1 and p_1, ρ_1 are the base altitude and corresponding atmospheric parameters for the section.

The pressure and density in the sections of linearly varying temperature is calculated as

$$p(h) = p_1 \left(\frac{T(h)}{T_1} \right)^{-g_0/(aR)}, \quad \rho(h) = \rho_1 \left(\frac{T(h)}{T_1} \right)^{-[(g_0/(aR))+1]}$$

where a is the temperature gradient [K/m], and R is the specific gas constant (286.91 JK^{-1}kg^{-1} for air).

Note that the gravitational acceleration g cannot be assumed constant over the altitude range required by Orbiter. To take this into account, altitude h in the above equations must be interpreted as a *geopotential* altitude. Conversion between geometric altitude h_G and geopotential altitude h is given by

$$h = \frac{r}{r + h_G} h_G$$

where r is the planet's mean radius. The graphs in Figure 3 show h_G.

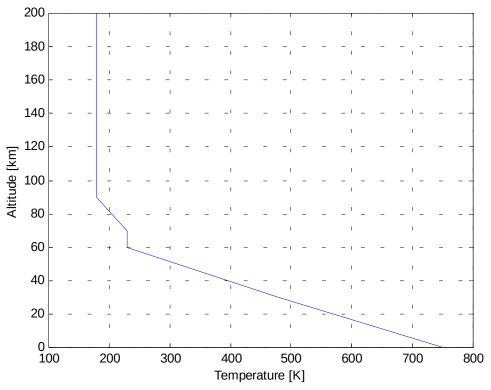
Note that even if the atmosphere is defined via a module function, the AtmAltLimit item in the configuration file is still required to determine the cutoff altitude. The AtmPressure0 and AtmDensity0 values are ignored.

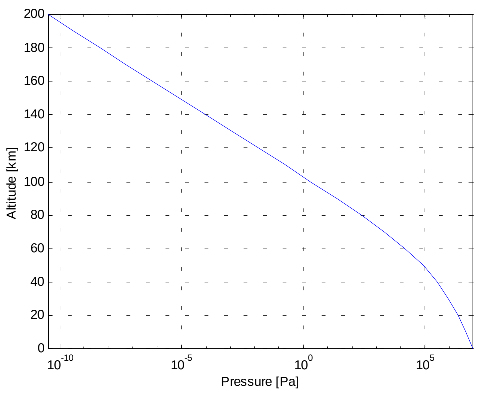
Currently only Earth and Mars feature customised atmosphere models. Other planets will follow later.

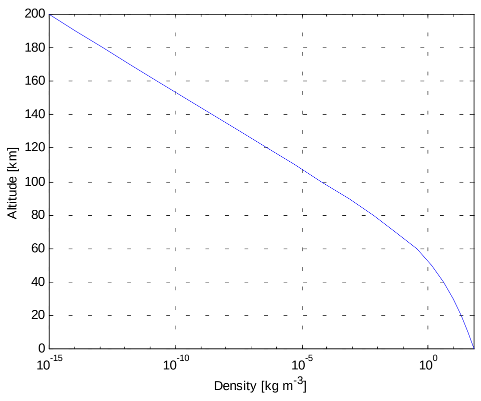
2.2.3 Venus atmosphere

We use the following atmospheric parameter profiles for Venus:

Altitude [km]	0	30	60	70	90	200
Temperature [K]	750	480	230	230	180	180
Pressure [Pa]	9.2M	897k	14.2k	1.85k	18.5	$3.4 \cdot 10^{-11}$
Density [kg m^{-3}]	65	9.9	0.33	0.043	$5.4 \cdot 10^{-4}$	$1.0 \cdot 10^{-15}$







Atmospheric parameters:

Surface pressure:	p_0 = 9.2 MPa
Surface density:	ρ_0 = 65 kg m^{-3}
Ratio of specific heats:	γ = 1.2857
Specific gas constant:	R = 188.92 J K^{-1} kg^{-1}

Orbiter defines the upper atmosphere altitude limit as 200 km. The cloud layer is set at an altitude of 60 km.

2.2.4 The speed of sound

Orbiter uses the equation for an ideal gas to compute the speed of sound as a function of absolute temperature:

$$a = \sqrt{\gamma R T}$$

where γ is the ratio of specific heat at constant pressure c_p, and specific heat at constant temperature, c_v, for the gas, $\gamma = c_p / c_v$ For air at normal conditions, γ = 1.4. This value is used by Orbiter as a default. It can be overridden by setting the `AtmGamma` parameter in the planet's configuration file.

R is the specific gas constant. By default, Orbiter uses the value for air, 286.91 J K^{-1} kg^{-1}. This can be overridden by setting the `AtmGasConstant` parameter in the planet's configuration file.

Mach number: The Mach number is an essential parameter in aerodynamics. It expresses a velocity v in units of the current speed of sound:

$$M = v/a$$

3 References

1. J. D. Anderson, Jr. "Introduction to Flight", 4th edition, McGraw-Hill, 2000.