ORBITER Programmer's Guide

Copyright (c) 2000-2009 Martin Schweiger 05 March 2009
Orbiter home: orbit.medphys.ucl.ac.uk/ or www.orbitersim.com

Contents

1 SPACECRAFT DESIGN 2
1.1 Module initialisation 2
1.2 Vessel initialisation 2
1.3 Reading and saving a vessel state 3
1.4 Defining class capabilities 4
1.5 Creating rocket engines 4
1.6 Air-breathing engines 10
1.7 Rendering re-entry flames 12
1.8 Adding particle streams for exhaust and reentry effects 13
1.9 Atmospheric flight model 15
1.10 Defining an animation sequence 17
1.11 Designing instrument panels 21
1.12 Designing virtual cockpits 29
2 PLANETS AND MOONS 35
2.1 Planet texture maps 35
2.2 Planet modules 37
2.3 Defining an atmosphere 40
3 REFERENCES 47

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 by writing a vessel configuration file without a custom module, 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 to individual vessels in global or static variables, because they can be overwritten by another vessel.

1.1 Module initialisation

When the user launches the simulation by picking a scenario from the Orbiter Launchpad dialog and pressing the "Launch Orbiter" button, Orbiter will load the vessel DLL module for each spacecraft type used in the simulation, and call its `InitModule` function. This function is called only once per Orbiter session, no matter how many spacecraft of that type appear in the simulation. It will not be called again if the user exits the simulation to the Launchpad, and reloads another simulation scenario. You can use it to initialise global (non-instance specific and non-session specific) parameters.

```
#define ORBITER_MODULE
#include "orbitersdk.h"

HINSTANCE g_hDLL;

DLLCLBK void InitModule (HINSTANCE hModule)
{
    g_hDLL = hModule;
    // perform global module initialisation here
}
```

In this example, we use the `InitModule` function to save the module instance handle passed to the function in global variable `g_hDLL`. This handle is useful later, e.g. when loading resources stored in the module file. Note the first line of the code example, which defines the `ORBITER_MODULE` flag. This flag should be included in all Orbiter DLL modules, to ensure proper execution of initialisation and cleanup functions.
At the end of a simulation run, Orbiter calls the `ExitModule` function for each DLL module.

```
DLLCLBK void ExitModule (HINSTANCE hModule)
{
    // perform module cleanup here
}
```

If you performed any dynamic memory allocation in `InitModule`, this is a good place to perform the corresponding cleanup operations which de-allocate that memory.

1.2 Vessel initialisation

To allow initialisation of individual spacecraft, Orbiter will call the ovcInit function each time a scenario is loaded, for each vessel of that type listed in the scenario file. Orbiter will also call ovcInit during the simulation if a new vessel of this type is created. The main purpose of ovcInit is to create an instance of a VESSEL-derived interface class. VESSEL is a class defined in the Orbiter API which is the primary means of communication between Orbiter and your own spacecraft class. In order to make use of the interface, you should derive your own vessel class derived from VESSEL. In ovcInit, you then create an instance of that class and return it back to Orbiter. Note that in the latest Orbiter version, the new VESSEL2 class has been introduced which inherits all the methods of VESSEL, and introduces a number of new

callback functions which replace the previous method of event notification. You should derive your vessel class from VESSEL2 to make use of this latest interface.

As an example, let's create a new class called `MyVessel`, and create an instance in `ovcInit`:

```
class MyVessel: public VESSEL2 {
public:
    MyVessel (OBJHANDLE hObj, int fmodel): VESSEL2 (hObj, fmodel) {}
    ~MyVessel () {}
    // add more vessel methods here
};

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

`ovcInit` passes two parameters to your module: a handle to the vessel for which you are about to create an interface, and a flag for the type of flight model requested by the user. Both parameters are passed on to the vessel constructor. The vessel handle is required to identify your vessel when requesting information from Orbiter. The flightmodel flag can be used to implement different behaviour in your module, for example to define an "easy" and a "complex" flight model, which can then be selected by the user. You don't need to store these parameters in your module, because you can retrieve them with the `GetHandle` and `GetFlightModel` methods of the `VESSEL` class.

To ensure proper cleanup at the end of a simulation session, you must implement the ovcExit function to delete your vessel:

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

Note that you need to cast the generic `VESSEL` pointer passed by Orbiter to your own vessel class to ensure that the correct destructors are called.

1.3 Reading and saving a vessel state

Next, you need to make sure that your vessel is able to read its initial state from a scenario file at the start of a simulation, and to save its state in a scenario at the end of the simulation. This is done by overloading the `clbkLoadStateEx` and `clbkSaveState` methods of the `VESSEL2` class. Note that you only need to overload these methods if your vessel requires nonstandard parameters to be stored in the scenario file. Standard parameters (such as position or velocity) are automatically read and written by the base class methods.

```
class MyVessel: public VESSEL2 {
public:
    MyVessel (OBJHANDLE hObj, int fmodel): VESSEL2 (hObj, fmodel) {}
    ~MyVessel () {}
    void clbkLoadStateEx (FILEHANDLE scn, void *status);
    void clbkSaveState (FILEHANDLE scn);
private:
    double myparam;
};

void MyVessel::clbkLoadStateEx (FILEHANDLE scn, void *status)
{
    char *line;

    while (oapiReadScenario_nextline (scn, line)) {
```

```
        if (!strnicmp (line, "MYPARAM", 7)) {
            sscanf (line+7, "%lf", &myparam);
        } else {
            ParseScenarioLineEx (line, status);
        }
    }
}

void MyVessel::clbkSaveState (FILEHANDLE scn)
{
    VESSEL2::clbkSaveState (scn);
    oapiWriteScenario_float (scn, "MYPARAM", myparam);
}
```

In the code fragment above, we use the overloaded clbkLoadStateEx function to read myparam from the scenario, were it is stored under the MYPARAM label. The function reads each line of the scenario file associated with our vessel, using the `oapiReadScenario_nextline` function. In the loop, we process the MYPARAM line, and pass everything else to Orbiter via `ParseScenarioLineEx` for default processing. Likewise, in clbkSaveState, the base class method `VESSEL2::clbkSaveState` is called to store all default parameters, before writing our private MYPARAM value. Of course, a real vessel implementation may need to store a large number of parameters in the scenario to make sure its status is completely defined when the scenario is loaded next time.

1.4 Defining class capabilities

One of the most important callback functions that should be overloaded is the clbkSetClassCaps method. It defines the general capabilities and properties of your spacecraft, e.g. its mass, size, visual representation, engine layout etc.

```
void MyVessel::clbkSetClassCaps (FILEHANDLE cfg)
{
    SetEmptyMass (1000.0);
    SetSize (10.0);
    AddMesh (oapiLoadMeshGlobal ("MyVessel.msh"));
    // define vessel capabilities here
}
```

In the above example, we define a few essential parameters (empty mass and mean radius), and load a mesh to provide a visual representation for our new spacecraft class. In practical applications, many more parameters may have to be defined here. Note that the file handle passed to the function points to the configuration file (.cfg) of the vessel. This can be used to read parameters from the file, thereby allowing the user to overwrite parameters by editing the configuration file.

We now have a "skeleton implementation" for our new spacecraft class. To make it interesting, many more properties need to be defined, such as rocket engines (or air-breathing engines), aerodynamic properties, animations, etc. Some of these aspects are described in the rest of this chapter. For a complete (and sometimes quite complex) vessel implementations, see the sample projects in the Orbitersdk\samples subdirectory.

1.5 Creating rocket 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.

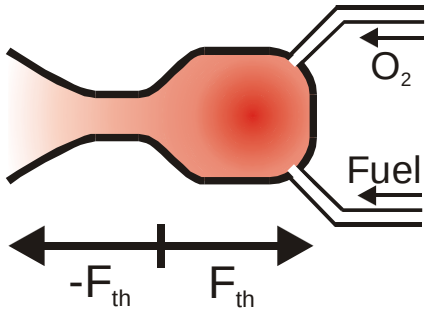
1.5.1 A bit of theory

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.



Sometimes the *thrust-specific fuel consumption* (TSFC) is quoted in the literature. This is the amount of propellant that needs to be burned per second to obtain 1N of thrust. Thus the TSFC is the inverse of the Isp and has units of [s m^{-1}], or more intuitively [kg s^{-1} N^{-1}].

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

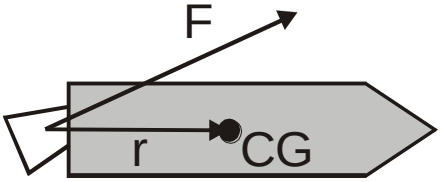
$$F(p) = F_{\max}(p) \cdot L$$

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)



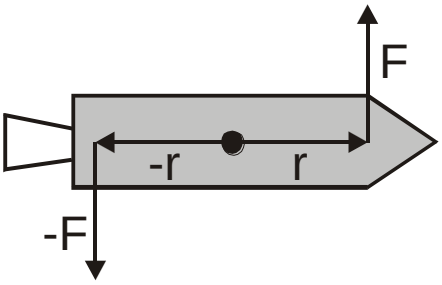
The torque is then given by the cross product

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

(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.

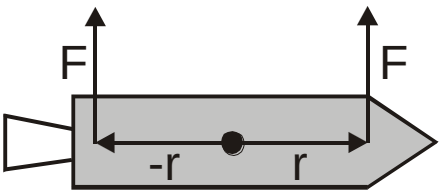
Attitude thrusters: Rotation

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 trapezoid rule.

1.5.2 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 clbkSetClassCaps callback function, for example like this (assuming that MyVessel is a class derived from VESSEL2):

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

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::clbkSetClassCaps (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::clbkSetClassCaps (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::clbkSetClassCaps (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 extrapolate 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::clbkSetClassCaps (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.

1.5.3 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 exhaust 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 exhaust 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.

1.6 Air-breathing engines

Orbiter is not limited to rocket engines. Other devices for generating thrust can be implemented as well, from turbojet engines to solar sails or some hypothetical future technology. Unlike conventional rocket engines, which are natively supported by the Orbiter core, custom designs require a bit more work from the developer. As an example, I will here discuss the (tentative) scramjet engine implementation used by the delta-glider.

A ramjet engine is a type of a jet engine which compresses the air for combustion not by any mechanical rotating machinery, but simply by "ramming" through the atmosphere, i.e. by using the aircraft's velocity in the airstream. This is an efficient way of generating thrust at supersonic speeds, but does not work at very low speed. (A scramjet is a variant where the air is not slowed down to subsonic speeds in the combustor and therefore avoids excessive heating at extreme velocities).

A typical ramjet engine is composed of 3 sections:

- the inlet diffuser where the air is isentropically decelerated, with pressure increasing from freestream pressure p_∞ to p_d, and temperature increasing from freestream temperature T_∞ to T_d.
- the combustion chamber, where the air-fuel mixture is burned at constant pressure $p_b = p_d$, and temperature increases from T_d to T_b.
- the exhaust nozzle, where the hot, high-pressure gas is expanded isentropically, with pressure decreasing from p_b to p_∞, and temperature decreasing from T_b to T_e.

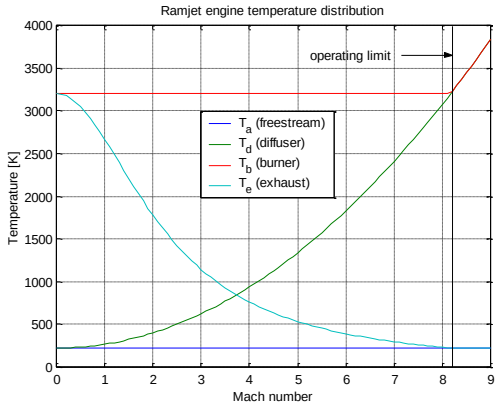
The temperatures and pressures in the three parts of the engine (diffuser, burner and exhaust) can be calculated in the following form:

$$T_d = T_\infty\left(1+\frac{\gamma-1}{2}M_\infty^2\right) \qquad p_d = p_\infty\left(\frac{T_d}{T_\infty}\right)^{\gamma/(\gamma-1)}$$

$$T_b = \max(T_{b0}, T_d) \qquad p_b = p_d$$

$$T_e = T_b\left(\frac{p_e}{p_b}\right)^{(\gamma-1)/\gamma} \qquad p_e = p_\infty$$

where M_∞ is the freestream Mach number, γ is the ratio of specific heats, and T_{b0} is the burner temperature limit, an engine design parameter defined by the heat resistance of the combustion chamber material. Note that if at high velocities $T_d > T_{b0}$, the engine will start to overheat purely from the isentropic compression in the diffuser, without any combustion taking place! The figure below shows an example for the temperature distribution in the engine compartments as a function of freestream Mach number. The example assumes a burner temperature limit of $T_{b0} = 3200$ K. In this case, the limiting velocity is $v = \text{Mach } 8.2$.



To calculate the thrust generated by a scramjet, we start from the fundamental thrust equation for jet propulsion,

$$F = (\dot{m}_a + \dot{m}_f)v_e - \dot{m}_a v_\infty + (p_e - p_\infty)A_e$$

where $\dot{m}_a$ and $\dot{m}_f$ are the air and fuel mass rates, respectively (using the common notation $\dot{x} = dx/dt$), v_e and v_∞ are the exhaust and freestream velocities, and A_e is the exhaust cross section.

Because of the assumption $p_e = p_\infty$ the last term vanishes. The *specific thrust* is then given by

$$\frac{F}{\dot{m}_a} = (1 + D)v_e - v_\infty$$

where $D = \dot{m}_f / \dot{m}_a$ is the *fuel-to-air ratio*.

The amount of fuel burned in the combustion chamber must be adjusted so that the burner temperature limit is not exceeded. This leads to the following expression for *D*:

$$D = \frac{T_b - T_d}{Q / c_p - T_b}$$

where Q is a fuel-specific heating value and c_p is the specific heat at constant pressure, given by

$$c_p = \frac{\gamma R}{\gamma - 1}$$

The mass flow of air collected by the engine is a function of air intake cross section A_i, freestream density ρ_∞ and freestream velocity v_∞:

$$\dot{m}_a = \rho_\infty v_\infty A_i$$

where v_∞ can be expressed in terms of the freestream Mach number:

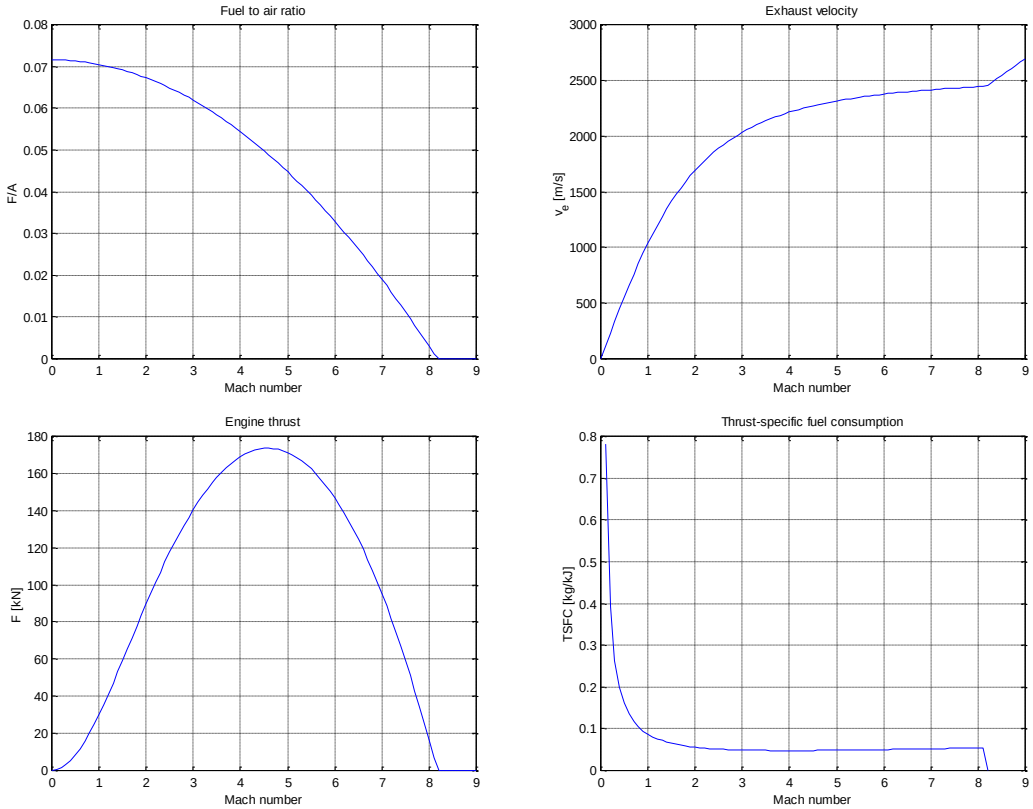
$$v_\infty = M_\infty \sqrt{\gamma R T_\infty}$$

From the above equations for D and $\dot{m}_a$ we can calculate the fuel rate $\dot{m}_f$ required to achieve combustion temperature T_b.

The final quantity required to calculate F is the exhaust velocity v_e. This can be obtained from the energy balance

$$c_p T_b = c_p T_e + v_e^2 / 2$$

We now have all the components to calculate the thrust F generated by the engine. The graphs below show various scramjet parameters for velocities in the range from Mach 0 to Mach 10 at an altitude of 10 km (assuming ρ_∞ = 0.43 kg/m^3 and T_∞ = 225 K). The DG engine design parameters in this example are $Q = 4.5 \cdot 10^7$ J/kg, A_i = 0.6 m^2, and T_{b0} = 3200 K.



1.7 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

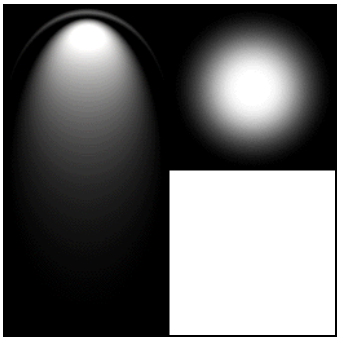
$$s = \min\left(1, \frac{P - P_0}{5P_0}\right)$$

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:

```
void MyVessel::clbkSetClassCaps (FILEHANDLE cfg)
{
  ...
  SURFHANDLE tex = oapiRegisterReentryTexture ("MyReentryFlame");
  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);
...
```

1.8 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: $\alpha = \begin{cases} 0 & \text{if } L < \mathrm{lmin} \\ \dfrac{L - \mathrm{lmin}}{\mathrm{lmax} - \mathrm{lmin}} & \text{if } \mathrm{lmin} \le L \le \mathrm{lmax} \\ 1 & \text{if } L > \mathrm{lmax} \end{cases}$

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

`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: $\alpha = \mathrm{amin}$

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

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

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

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

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.

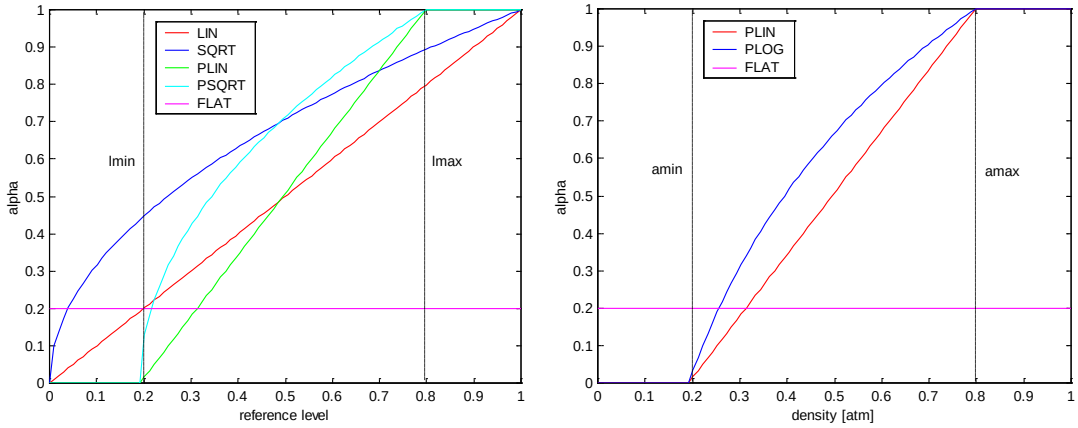


Figure 1: The particle alpha value as a function of reference level (left) and atmospheric parameter (right) for different 'levelmap' and 'atmsmap' modes.

1.9 Atmospheric flight model

1.9.1 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.

1.9.2 Lift and drag in transonic and supersonic flight

(to be completed)

1.9.3 Angular moments and vessel stability

(to be completed)

1.9.4 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.

1.9.5 API interface for airfoil definitions

To define the lift and drag characteristics 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 moments 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 (-π 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 (μ = 1.6894·10^{-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 not be exactly 0°).

1.10 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.

1.10.1 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.

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)
: VESSEL2 (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 (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 de-allocate 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.

1.10.2 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`.

1.11 Designing instrument panels

Instrument panels are a good way to give an individual feel to a spacecraft class and allow the user to monitor flight parameters and control specific aspects of the vessel via the mouse, without the need to remember a large number of keyboard commands.

There are two ways to define a cockpit interior: you can build one (or several) flat two-dimensional panels as bitmaps which are overlayed on top of the three-dimensional scenery of the simulation window (denoted as *panels* below), or you can construct a full three-dimensional mesh representation of the cockpit (denoted as *virtual cockpit*, or *VC* below). A vessel can implement both panels and virtual cockpits. The user can switch between them (and the generic cockpit view comprising two MFD displays and HUD) by pressing F8.

In this section we will discuss the steps required to define 2-D panels in the vessel module. Section 1.12 will deal with virtual cockpits.

1.11.1 Defining a panel

You will first need to create a bitmap which represents the 2-D instrument panel. You can use any paint tool capable of generating Windows BMP files. The panel can be saved in 8-bit or 24-bit mode, but 8-bit mode is strongly recommended to reduce the size of the resulting vessel module, and improve simulation performance.

Figure 2: The DG main panel bitmap.

Some thought should be given to the size of the panel bitmap. Remember that users will run Orbiter at different screen resolutions and window sizes. If the bitmap is made very large, a lot of panning will be required to bring different parts of the panel into view at low resolutions. If the bitmap is very small, it will cover only a small area of the screen at high resolutions. It is probably best to design panels for medium screen resolutions (between 1024x768 and 1280x960 pixels). Users with very low or very high screen resolutions will be able to adjust the panel size by using Orbiter's panel rescaling option.

You should also consider whether the panel is to cover the whole screen, or only part of it. The main panel should usually obstruct only part of the 3-D scenery, but side panels could take up the whole simulation window.

The main panel should typically also provide space for MFDs (multifunctional displays), which are the primary method to provide flight data to the pilot. Most common is a layout with two

MFDs, but fewer or more can be defined as well. The size of the MFD displays should be chosen so that they are easily readable over a 'typical' range of screen resolutions.

You can define more than one panel for a vessel. For example, you may define a main panel which is visible in the lower half of the screen when the pilot looks forward, an overhead panel, side panels, etc. The user can switch between the different panels with Ctrl+cursor keys. We will discuss later how to define the connectivity between panels. To start with, let's look at the definition of a single main panel.

Once you have created the panel BMP file, you should add it as a bitmap resource to your vessel module project. Now you are ready to write the code to support the panel. To do so, you need to overload the *clbkLoadPanel* method of the VESSEL2 class:

```
bool MyVessel::clbkLoadPanel (int id)
{
  ...
}
```

Here we assume that *MyVessel* is a class derived from VESSEL2 (see Section 1.2 on how to create vessel instances). *id* is a panel identifier which Orbiter will provide to let your function know which panel is required. If only a single main panel is defined, *id* will always be 0. If you define more than one panel, you should examine this parameter to decide which panel to load.

Orbiter will call your *clbkLoadPanel* method whenever it needs to load an instrument panel. This happens if

- the user switches to instrument panel view from another view mode by pressing F8.
- the user switches between panels with Ctrl+cursor keys.
- the user switches from an external view to a cockpit view.
- the user switches to a different spacecraft with F3.

In the body of *clbkLoadPanel*, we need to load the panel bitmap and pass it to Orbiter via the *oapiRegisterPanelBackground* function:

```
bool MyVessel::clbkLoadPanel (int id)
{
  HBITMAP hBmp = LoadBitmap (hDLL, MAKEINTRESOURCE(IDB_PANEL));
  oapiRegisterPanelBackground (hBmp);
  return true;
}
```

Here, *hDLL* is a module instance handle passed to the *InitModule* callback function of your module, and *IDB_PANEL* is assumed to be the numerical resource identifier of the panel bitmap. The return value of *clbkLoadPanel* should normally be *true*. *false* signifies an error, e.g. failure to load the panel bitmap.

oapiRegisterPanelBackground has an additional optional parameter which defines how the panel is connected to the edges of the simulation window, and how it can be scrolled across the screen with the cursor keys. A common choice for a main window is to connect it to the lower edge of the window, and allow it to be scrolled downward. This can be accomplished as follows:

```
  oapiRegisterPanelBackground (hBmp,
      PANEL_ATTACH_BOTTOM|PANEL_MOVEOUT_BOTTOM);
```

(This is in fact the default setting, so you only need to provide this parameter if you need to define a different behaviour.) For a full list of supported attachment and scroll parameters, see the *oapiRegisterPanelBackground* description in the Reference Manual.

oapiRegisterPanelBackground has a further optional parameter to define a transparent colour. Any part of the bitmap containing that colour will be transparent in the render window.

This allows to implement irregular panel shapes such as windows which provide a view of the 3-D scene though the panel.

The transparent colour is given in 0xRRGGBB format. Note that if Orbiter is run in 16-bit mode, not all colours can be represented. For that reason, it is recommended to use either black (0x000000) or white (0xFFFFFF) as the transparent colour which are always available in 16-bit mode, to avoid problems. In any case, you should always check that your panel appears correctly in both 16 and 32 bit modes before publishing your addon.

So the final version of our main panel loading call looks like this, where we allow the panel to be scrolled out at the bottom, and use white as the transparent colour:

```
oapiRegisterPanelBackground (hBmp,
    PANEL_ATTACH_BOTTOM|PANEL_MOVEOUT_BOTTOM, 0xFFFFFF);
```

At this point, you can try to compile your module and test the panel in Orbiter. You should be able to make the panel visible by pressing F8 when you are in the cockpit of an instance of your vessel class, and scroll it up and down with the cursor keys.

1.11.2 Defining active panel areas

Now we can start to do something interesting with our new panel. We need to *activate* areas of the panel. Active areas can do two things:

- They can be repainted from within the code, for example to dynamically update an instrument display, and/or
- they can register mouse button events to allow the user to interact with the panel.

A panel area is activated with the *oapiRegisterPanelArea* function. This must be called in your vessel's *clbkLoadPanel* method, after the panel has been loaded with *oapiRegisterPanel-Background*. Let's define an area that contains a button which the user can press:

```
oapiRegisterPanelArea (AID_BUTTON, _R(10,10,30,20),
    PANEL_REDRAW_MOUSE, PANEL_MOUSE_LBDOWN, PANEL_MAP_BACKGROUND);
```

The first parameter, *AID_BUTTON*, is a value that uniquely identifies the area across all panels. The next parameter defines a rectangular area in the panel given by the left, top, right and bottom edges (measured from the top left corner of the panel bitmap).
The next parameter, *PANEL_REDRAW_MOUSE*, specifies that the area must be redrawn whenever a mouse event occurs inside the area. Other areas may need to be redrawn at each frame, by explicitly requesting a redraw, or not at all.
PANEL_MOUSE_LBDOWN requests a notification whenever the user presses the left mouse button inside the area. You can also request mouse button releases, or continuous notifications as long as a button is pressed. A panel area defined with *PANEL_MOUSE_IGNORE* will never generate any mouse events.
PANEL_MAP_BACKGROUND requests the area background (i.e. the portion of the panel bitmap under the area) to be passed to the redraw function. Instead, you could request the current status of the area, or an un-initialised bitmap to be passed to the redraw function. See the documentation to *oapiRegisterPanelArea* in the Reference Manual for more details.

You can define more panel areas to turn your panel into a useful interface, but avoid overlapping areas.

Next, we need to implement the callback functions Orbiter will call to allow the module to respond to redraw and mouse events generated by the active areas.

1.11.3 The mouse event handler

To intercept mouse events generated by a panel you must overload the *clbkPanel-MouseEvent* method of the *VESSEL2* class:

```
bool MyVessel::clbkPanelMouseEvent (int id, int event, int mx, int my)
```

```
{
  ...
}
```

where *id* is the identifier of the panel area for which the event was generated (e.g. *AID_BUTTON* in our example), *event* specifies the mouse event type, and *mx,my* are the panel coordinates at which the event occurred.

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.

Inside *clbkPanelMouseEvent*, your code must check the area id and perform the appropriate actions:

```
bool MyVessel::clbkPanelMouseEvent (int id, int event, int mx, int my)
{
  switch (id) {
  case AID_BUTTON:
    DoProcessButtonPress (...);
    return true;
  case ...   // place response to other areas here
  }
  return false;
}
```

Here, *DoProcesButtonPress* is assumed to be a locally defined method which performs the required action.
The return value is currently only used for areas which use the *PANEL_REDRAW_MOUSE* flag. In this case, returning *true* will trigger a redraw event, while returning *false* will not. For efficiency, return *true* only if the area needs to be redrawn as a consequence of the mouse event.

The *mx* and *my* parameters define the area coordinates (0,0 is the top left corner of the area) at which the mouse event occurred. This is sometimes useful to fine-tune the response. For example, let's assume that the button defined in the example is actually a switch which can be flipped left or right. Then we could do this:

```
  ...
  case AID_BUTTON:
    if (mx < 10)
      DoProcessFlipLeft (...);
    else
      DoProcessFlipRight (...);
    return true;
  ...
```

1.11.4 The redraw event handler

To provide a visual cue of the button press, we may want to redraw the area (e.g. to simulate a control lamp lighting up). Other areas representing gauges and displays may need to be redrawn continuously without any mouse events. To respond to redraw requests, we need to overload the *clbkPanelRedrawEvent* method of the *VESSEL2* class:

```
bool MyVessel::clbkPanelRedrawEvent (int id, int event, SURFHANDLE surf)
{
  ...
}
```

As with the mouse event handler, your implementation of *clbkPanelRedrawEvent* should examine the area *id* (and the redraw *event*, if required), and redraw the corresponding area as required.

surf is a handle to the paint surface for the area in which all repainting takes place. The contents of the surface passed to the callback function depend on the parameters specified during the definition of the area:

PANEL_MAP_NONE	surf is undefined
PANEL_MAP_BACKGROUND	surf contains area background
PANEL_MAP_CURRENT	surf contains current area contents
PANEL_MAP_BGONREQUEST	surf is undefined, but area background can be obtained on request

PANEL_MAP_NONE is the most efficient option if the whole area needs to be redrawn at each redraw event. *PANEL_MAP_BACKGROUND* is least efficient, because it involves the most internal surface copy processes. If you need the background bitmap, but your area doesn't need to be redrawn for each redraw request generated (for example, if you have defined a gauge, to be redrawn at each simulation frame, but often the contents don't change between subsequent frames), it is more efficient to use the *PANEL_MAP_BGONREQUEST* flag, and obtaining the background bitmap explicitly with a call to *oapiBltPanelAreaBackground* whenever the area actually needs to be redrawn (see documentation to *oapiBltPanelAreaBackground* in the Reference Manual for more details).

Our redraw function might look like this:

```
bool MyVessel::clbkPanelRedrawEvent (int id, int event, SURFHANDLE surf)
{
  switch (id) {
  case AID_BUTTON:
    if (button_pressed)
      oapiBlt (surf, buttonSurf, 0, 0, 0, 0, 20, 10);
    else
      oapiBlt (surf, buttonSurf, 0, 0, 0, 10, 20, 10);
    return true;
  case ... // imprement redraw methods for other areas
  }
  return false;
}
```

Here, *buttonSurf* is assumed to be the surface handle to a bitmap which contains images of the button for both the pressed and the released state. (You can store this bitmap as a module resource and obtain a surface handle to it with the *oapiCreateSurface* method.) *oapiBlt* copies the relevant part of the bitmap into the area's surface (the *button_pressed* flag could for example have been set in the mouse event handler).

When more complex redrawing is required, you can obtain a device context handle to the surface with *oapiGetDC* and then use standard Windows GDI methods to paint in the surface. (see Windows API documentation). Don't forget to release the device context with *oapiReleaseDC* at the end.

The return value of *clbkPanelRedrawEvent* signals to Orbiter if the contents of the area have been redrawn. Return *true* only if you did modify the surface, *false* otherwise.

1.11.5 Defining panel MFDs

MFD (multifunctional displays) are probably the most important components of your panel. They are defined differently to other panel areas, because some of the redraw events are processed directly by Orbiter.

MFDs consist of a square display area (representing a colour CRT or LCD display) and rows of control buttons to the left and right. The number of buttons can be defined individually.

You reserve a panel area for an MFD with the *oapiRegisterMFD* method during setting up the panel in the overloaded *clbkLoadPanel* callback function:

```
bool MyVessel::clbkLoadPanel (int id)
{
  oapiRegisterPanelBackground (...);
  ...
  MFDSPEC mfds_left  = {{100, 10, 200, 110}, 6, 6, 10, 20};
  oapiRegisterMFD (MFD_LEFT,  mfds_left);
  ...
  return true;
}
```

The first parameter of *oapiRegisterMFD* identifies the MFD (left, right, or a user-defined MFD). The left and right MFDs can be controlled with keyboard commands, while user-defined MFDs can only be controlled with the mouse. Therefore you should always first define the left and right MFDs, and use user-defined ones only if more than two MFDs are to be defined in the panel.

The second parameter is a structure which defines the layout of the MFD. It contains:

- the rectangular area (left, top, right and bottom edge) of the panel area to contain the MFD display,
- the number of buttons along the left and right edges,
- the y-offset of the upper edge of the topmost button from the top edge of the display,
- the y-distance between the top edges of the buttons.

The button rows must be implemented as separate areas. Note that a single area is used for the left row of buttons, and another one for the right row. In addition, a bottom row of 3 buttons can be defined to perform MFD on/off, display of button commands, and display of mode contents:

```
bool MyVessel::clbkLoadPanel (int id)
{
  oapiRegisterPanelBackground (...);
  ...
  MFDSPEC mfds_left  = {{100, 10, 200, 110}, 6, 6, 10, 20};
  oapiRegisterMFD (MFD_LEFT,  mfds_left);
  oapiRegisterPanelArea (AID_LBUTTONS, _R(80,20,100,100), PANEL_REDRAW_USER,
      PANEL_MOUSE_LBDOWN|PANEL_MOUSE_LBPRESSED, PANEL_MAP_BACKGROUND);
  oapiRegisterPanelArea (AID_RBUTTONS,_R(200,20,220,100), PANEL_REDRAW_USER,
      PANEL_MOUSE_LBDOWN|PANEL_MOUSE_LBPRESSED, PANEL_MAP_BACKGROUND);
  oapiRegisterPanelArea (AID_BBUTTONS,_R(100,110,200,130),
      PANEL_REDRAW_NEVER, PANEL_MOUSE_LBDOWN);
  ...
  return true;
}
```

The button areas have been defined with the *PANEL_MOUSE_LBPRESSED* flag in addition to *PANEL_MOUSE_LBDOWN*, so that continued mouse presses can be recorded when required.

Mouse button events now need to be processed in the mouse event handler:

```
bool MyVessel::clbkPanelMouseEvent (int id, int event, int mx, int my)
{
  switch (id) {
  case AID_LBUTTONS:
  case AID_RBUTTONS:
    if (my%20 < 15) {
      int bt = my/20 + (id == AID_LBUTTONS ? 0 : 6);
      oapiProcessMFDButton (MFD_LEFT, bt, event);
      return true;
    }
    break;
  case ...
  }
  return false;
}
```

This code fragment processes all the buttons in the left and right button columns simultaneously. It first checks if the mouse event occurred over a button (*my%20 < 15*), assuming that each button is 15 pixels high, and buttons are spaced in 20 pixel intervals. It then checks if the event occurred in the left or right button column, and determines which of the buttons was pressed (*bt*). Finally, the *oapiProcessMFDButton* function is called with the appropriate parameters, to allow Orbiter to respond to the MFD request.

The bottom row of buttons is processed similarly:

```
  ...
  case AID_BBUTTONS:
    if (mx < 20)
      oapiToggleMFD_on (MFD_LEFT);
    else if (mx >= 30 && mx < 50)
      oapiSendMFDKey (MFD_LEFT, OAPI_KEY_F1);
    else if (mx > 60)
      oapiSendMFDKey (MFD_LEFT, OAPI_KEY_GRAVE);
    return true;
  ...
```

where *oapiToggleMFD_on* switches the MFD on/off, and the *oapiSendMFDKey* commands trigger the default actions of displaying the key commands and the MFD mode list.

Of course, the values of the various mouse x and y values in an actual implementation will depend on the geometry of the individual MFD layout. You could even define each single button as a separate area, but this will generally result in less efficient code.

Finally, the MFD buttons need to respond to redraw events, to reflect the change of button labels (for example, when the MFD mode changes). Note that the MFD display area itself is automatically updated by Orbiter and therefore doesn't need to implement a redraw response.

```
bool MyVessel::clbkPanelRedrawEvent (int id, int event, SURFHANDLE surf)
{
  switch (id) {
  case AID_LBUTTONS:
  case AID_RBUTTONS:
    side = (id == AID_LBUTTONS ? 0:1);
    hDC = oapiGetDC (surf);
    for (int bt = 0; bt < 6; bt++) {
      if (label = oapiMFDButtonLabel (MFD_LEFT, bt+side*6))
        TextOut (hDC, 5, 2+20*bt, label, strlen(label));
      else break;
    }
    oapiReleaseDC (surf, hDC);
    return true;
  case ...
```

```
  }
  return false;
}
```

This uses the *oapiMFDButtonLabel* function to retrieve the label text for each of the buttons (button labels consist of 1 to 3 characters). The redraw function can be customised to reflect the style in which the button labels are displayed (for example by changing the font size or colour).

Note that the bottom row of buttons does not necessarily need to implement a redraw method, because their labels never change.

1.11.6 Multiple panels

To implement multiple panels for a vessel, the *clbkLoadPanel* method must load different panels depending on the provided panel *id*, and each of the panels 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 the overloaded *clbkLoadPanel* could look like this:

```
bool MyVessel::clbkLoadPanel (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 its own background bitmap via the *oapiRegisterPanelBackground* function.

In a vessel that 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 *clbkLoadPanel*. For example, if panel 0 calls *oapiSetPanelNeighbours (2,-1,1,-1)*, then panels 1 and 2 must be handled by *clbkLoadPanel*.

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 1 as its top neighbour, then panel 1 should define panel 0 as its bot-

tom neighbour. If only a single panel (panel 0) is supported, calling *oapiSetPanelNeighbours* is not necessary.

1.12 Designing virtual cockpits

The concepts used for defining virtual 3-D cockpits are similar to those of 2-D panels. They too are defined via a load function, mouse and redraw event handlers. In fact, some of the redraw methods defined for panel areas may be reused to update parts of the virtual cockpit textures, so it is useful to familiarise yourself with 2-D panel implementations before progressing to virtual cockpits.

1.12.1 Defining a virtual cockpit

A virtual cockpit requires a 3-D mesh representation. In principle it is possible to add the cockpit directly to the mesh used to represent the vessel in external views, and flag this mesh to be visible both in external and cockpit views (via the *SetMeshVisibility* method), but in general it is more efficient to design a separate mesh for the cockpit which is visible only in virtual cockpit view mode (using *SetMeshVisibility* with the *MESHVIS_VC* flag). Make sure that the cockpit mesh is consistent with the external mesh. A good way to achieve this is by building the VC together with the external mesh in your 3D design program, but exporting the cockpit and the external parts to separate mesh files.

Figure 3: A view of the DG virtual cockpit.

To make the VC mode available in your mesh class, you must overload the *clbkLoadVC* method of the *VESSEL2* class:

```
bool MyVessel::clbkLoadVC (int id)
{
  ...
}
```

This will allow the user to switch to VC mode with the F8 key. The *id* parameter is currently always 0. Eventually it will allow to select different cockpit positions.

In the body of *clbkLoadVC*, you can define camera parameters:

```
bool MyVessel::clbkLoadVC (int id)
{
  SetCameraOffset (_V(0,1.5,6.0));
  SetCameraDefaultDirection (_V(0,0,1));
  SetCameraRotationRange (RAD*120, RAD*120, RAD*70, RAD*70);
  SetCameraShiftRange (_V(0,0,0.1), _V(-0.2,0,0), _V(0.2,0,0));
  ...
}
```

SetCameraOffset defines the camera (or pilot eye) position in the vessel coordinate frame. *SetCameraDefaultDirection* defines the default direction the pilot is looking toward, *SetCameraRotationRange* defines how far he can turn his head left right, up and down, and *SetCameraShiftRange* allows to simulate the pilot 'leaning' forward, left or right, for example to get a better view out of a window.

Note that you only need to define these camera parameters here if they change between different cockpit view modes. If all camera modes use the same parameters, they can be defined globally in the overloaded *clbkSetClassCaps* method.

Once you have implemented the *clbkLoadVC* method thus far and defined the VC mesh, you should be able to compile the module and test the virtual cockpit mode in Orbiter. Try rotating the view with Alt+cursor keys, and 'leaning' with Ctrl+Alt+cursor keys. When you are satisfied with the camera parameters, you can proceed to activate VC areas.

1.12.2 Defining active VC areas

As with 2-D panels, virtual cockpit areas must be activated to allow dynamic updates or to respond to user input. This is how an active area is defined in *clbkLoadVC*:

```
bool MyVessel::clbkLoadVC (int id)
{
  ...
  SURFHANDLE tex = oapiGetTextureHandle (vcmesh, 10);
  oapiVCRegisterArea (AID_BUTTON, _R(0,0,20,10), PANEL_REDRAW_ALWAYS,
      PANEL_MOUSE_LBDOWN, PANEL_MAP_BGONREQUEST, tex);
  oapiVCSetAreaClickmode_Spherical (AID_BUTTON, _V(5,3.3,7.1), 2.5);
  ...
}
```

As with 2-D panel area definitions, the first parameter of *oapiVCRegisterArea* defines a unique identifier for the area (*AID_BUTTON* in this case). The next parameter defines a rectangular area (in pixel units) in a texture that is updated dynamically in a redraw event. If the area doesn't need to update any textures (e.g. because it only responds to mouse events, or because it provides visual feedback by modifying the mesh geometry), this parameter is ignored and can be set to _R(0,0,0,0).

The third parameter defines the events which trigger a redraw notification for the area. In this case we have set it to PANEL_REDRAW_ALWAYS, i.e. we request a redraw notification at each simulation frame (typical for gauges whose displays change constantly). Note that unlike 2-D panels, the term 'redraw event' stands for any change in the visual representation of the area. This may consist of repainting a dynamic texture, but it could also mean a mesh group animation or direct editing of mesh vertices or texture coordinates.

The fourth parameter defines the mouse events which trigger a notification for the area. They are used in the same way as 2-D panel areas, but an additional function call is required to define the mouse-sensitive area (see below).

The fifth parameter defines the initial contents of the drawing bitmap passed to the redraw notification. It is used in the same way as for 2-D panels. However, if the redraw event does not update a dynamic texture, this *must* be set to PANEL_MAP_NONE.

The last parameter is a handle to the dynamic texture passed to redraw notifications. In this example, we have obtained the texture handle from mesh group 10 of the VC mesh via a call to *oapiGetTextureHandle*. Note that textures obtained for dynamic repainting must be labelled as dynamic in the mesh file (see next section). If the area does not need to redraw a texture during a redraw event, this parameter can be set to *NULL*. In that case, there is a shorter version of *oapiVCRegisterArea* for convenience which omits the second, fifth and sixth parameters.

Unlike 2-D panels, the mouse-sensitive region of a VC area must be defined with a separate function call. In virtual cockpits, the sensitive region is a 3-D volume. Orbiter draws a virtual ray from the camera position through the screen point at which a mouse event occurred, and checks whether the ray intersects a mouse-sensitive volume. If so, the corresponding mouse event is generated.

You can define either a spherical or a quadrilateral mouse-sensitive region. A spherical region is defined via the *oapiVCSetAreaClickmode_Spherical* method, where you specifiy the centre of the spherical region in the vessel frame of reference, and its radius. This will trigger a mouse event whenever the user clicks inside the projection of the sphere onto the simulation window.

Quadrilateral (e.g. rectangular) regions are defined via the oapVCSetAreaClickmode_Quadrilateral method, where you specify the four corners of the mouse-sensitive region in space. Again, this will trigger mouse events whenever the user clicks inside the projection of the sensitive area on the simulation window.

Spherical regions are slightly more efficient for Orbiter to test, but quadrilateral regions return information about the relative position at which the mouse click occurred, so they are somewhat more versatile.

1.12.3 Defining dynamic textures

One way to provide information to the pilot in VC mode is by repainting the bitmaps used to texture VC mesh groups. For example, you can implement gauges and data displays in this way. You may even be able to re-use a panel area redraw method used for 2-D panels to update a VC texture, minimising the additional coding effort.

Important: Orbiter can only draw into uncompressed textures. For this reason, textures which support dynamic repainting must be marked in the mesh file with a 'D' (*dynamic*), e.g.

```
...
Textures 2
tex1.dds
tex2_dyn.dds D
```

Dynamic textures are less efficient than static ones, so you should try to keep them to a minimum. Collect all parts that require dynamic updates in one or few (small) texture files, and keep them apart from the static parts.

1.12.4 The mouse event handler

Whenever a mouse event occurs inside the mouse-sensitive volume of an active area, a notification is passed to your module. To respond to such events, you must overload the *clbkVCMouseEvent* method of the VESSEL2 class.

```
bool MyVessel::clbkVCMouseEvent (int id, int event, VECTOR3 &p)
{
  ...
}
```

where *id* is the area identifier, and *event* is the mouse event that triggered the notification (The VC notification uses the same event types as 2-D panels).

Parameter *p* returns some information about the mouse position at the event. The information returned depends on the area type for which the event was generated. For spherical regions, p.x contains the distance of the mouse position from the centre of the area, while p.y and p.z are not used. For quadrilateral regions, p.x and p.y contain the relative mouse x and y positions within the region, where the top left corner of the region has coordinates (0,0), and the bottom right corner has coordinates (1,1). This allows to define differentiated responses depending on where in the region the event occurred, similar to the procedure in 2-D panel regions.

Inside *clbkVCMouseEvent*, your code must check the area id and perform the appropriate actions:

```
bool MyVessel::clbkVCMouseEvent (int id, int event, VECTOR3 &p)
{
  switch (id) {
  case AID_BUTTON:
    DoProcessButtonPress (...);
    return true;
  case ...   // place response to other areas here
  }
  return false;
}
```

1.12.5 The redraw event handler

Any active areas which specified a redraw flag other than *PANEL_REDRAW_NEVER* during initialisation, will trigger redraw notifications for the appropriate events. Your code needs to overload the *clbkVCRedrawEvent* method of the VESSEL2 class to respond to those events.

```
bool MyVessel::clbkVCRedrawEvent (int id, int event, SURFHANDLE surf)
{
  ...
}
```

where *id* is the area identifier, *event* is the redraw event that triggered the notification, and *surf* is a handle to the dynamic texture to be redrawn. *surf* may be *NULL* if you didn't specify a texture during the area initialisation.

Inside *clbkVCRedrawEvent*, check the area id and perform the appropriate redraw action for that area. Typically, this will be one of the following:

- Repainting the dynamic texture passed to the notification handler. This is done in the same way as repainting 2-D panel areas. In fact, you may even be able to re-use the same code. Repainting textures is a good way to update displays and instrument gauges.
- Animating a mesh group. This can be used to simulate flipping a switch or pushing a lever. See Section 1.10 for details on animations.
- Editing a mesh. You can use the *oapiMeshGroup* function to access the vertices of a mesh group, and edit the vertex positions or texture coordinates. Editing texture coordinates may be a good alternative to redrawing a texture if the texture is to switch between discrete pre-defined states.

clbkVCRedrawEvent should return *true* only if you have modified the dynamic texture. If the texture was not modified, or is undefined, the function should return *false*.

1.12.6 Defining MFDs in the virtual cockpit

To define a multifunctional display inside a virtual cockpit, you need to perform the following steps:

Create a new group in the mesh consisting of a flat square area (defined by 4 vertices and 2 triangles). This is going to be the MFD display. The texture coordinates of the vertices should be: top left corner: (0,0), top right corner: (1,0), bottom left corner: (0,1) and bottom right corner: (1,1). Set 'TEXTURE 0' and 'FLAG 3' for this group. This will exclude the group from normal rendering (Orbiter uses a special render pass for MFDs). You can select a material of your choice. A material with specular reflection will produce a 'glass surface' effect.

In *clbkLoadVC*, define the MFD display with *oapiVCRegisterMFD*:

```
bool MyVessel::clbkLoadVC (int id)
{
  ...
  static VCMFDSPEC mfds_left  = {1, 100};
  oapiVCRegisterMFD (MFD_LEFT, &mfds_left);
  ...
}
```

VCMFDSPEC is a structure which contains the mesh index and group index of the MFD display group defined in the previous step. *oapiVCRegisterMFD* registers this group as an MFD display (in this case, the left MFD).

Next, you need to define the MFD control buttons. How you implement them is mostly up to you. Typically, you define each button as a rectangle and collect all rectangles into a single mesh group. Reserve space on a dynamic texture for drawing the button labels, and set the texture coordinates for the button rectangles accordingly.
Then you define an active area for each button to receive mouse events (but no redraw events). You also define a dummy area for redraw events. Pass the dynamic texture handle reserved for that purpose to the redraw area. This could look as follows:

```
bool MyVessel::clbkLoadVC (int id)
{
  ...
  oapiVCRegisterArea (AID_LBUTTONS, _R(0,0,20,100),PANEL_REDRAW_USER,
      PANEL_MOUSE_IGNORE, PANEL_MAP_BACKGROUND, tex);
  oapiVCRegisterArea (AID_RBUTTONS,_R(20,0,40,100), PANEL_REDRAW_USER,
      PANEL_MOUSE_IGNORE, PANEL_MAP_BACKGROUND, tex);
  for (i = 0; i < 6; i++) {
    oapiVCRegisterArea (AID_LBUTTON1+i, PANEL_REDRAW_NEVER,
        PANEL_MOUSE_LBDOWN|PANEL_MOUSE_LBPRESSED);
    oapiVCSetAreaClickmode_Spherical (AID_LBUTTON1+i,
        _V(0.2,0.1-i*0.02,2.0), 0.01);
    oapiVCRegisterArea (AID_RBUTTON1+i, PANEL_REDRAW_NEVER,
        PANEL_MOUSE_LBDOWN|PANEL_MOUSE_LBPRESSED);
    oapiVCSetAreaClickmode_Spherical (AID_RBUTTON1+i,
        _V(0.4,0.1-i*0.02,2.0), 0.01);
  }
  ...
}
```

You should also define mouse-active areas for the three bottom MFD buttons.

In the mouse event handler, trap any mouse clicks on the MFD buttons and pass them to the *oapiProcessMFDButton* function:

```
bool MyVessel::clbkVCMouseEvent (int id, int event)
{
  if (id >= AID_LBUTTON1 && id < AID_LBUTTON1+12) {
    oapiProcessMFDButton (MFD_LEFT, id-AID_LBUTTON1, event);
    return true;
  }
  ...
  return false;
}
```

In the redraw event handler, trap MFD button redraw requests and redraw the buttons as required:

```
bool MyVessel::clbkVCRedrawEvent (int id, int event, SURFHANDLE surf)
{
  switch (id) {
  case AID_LBUTTONS:
    RedrawMFDButtons (surf, MFD_LEFT, 0);
    return true;
  case AID_RBUTTONS:
    RedrawMFDButtons (surf, MFD_RIGHT, 0);
    return true;
  case ...
}
```

where *RedrawMFDButtons* is assumed to be a locally defined function performing the redraw action. You may be able to re-use the same method used for drawing the MFD buttons in the 2-D panel (see Section 1.11.5).

Finally, trigger a redraw event in the body of the MFD mode change callback notification.

```
void MyVessel::MFDMode (int mfd, int mode)
{
  switch (mfd) {
  case MFD_LEFT:
    oapiTriggerVCRedrawArea (0, AID_LBUTTONS);
    oapiTriggerVCRedrawArea (0, AID_RBUTTONS);
    break;
  case ...
}
```

1.12.7 Defining the HUD in the virtual cockpit

< 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
- 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 4 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 4: 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").

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 Planet modules

Planet modules can be used to control the motion of a planet (or any other celestial body, such as a moon, the sun, or an asteroid) within the solar system. This allows to implement sophisticated analytic ephemerides solutions which take into account perturbations from other celestial objects.

Planets which are not controlled via a DLL module are updated directly by Orbiter. Depending on the settings in the definition file, Orbiter either uses an unperturbed 2-body approximation, resulting in a conic section trajectory (e.g. an ellipse), or uses a dynamic update procedure based on the gravitational forces acting on the planet. Both methods have limitations: the 2-body approach ignores perturbations and is only valid if no massive bodies other than the orbit reference object are nearby. The dynamic update accumulates numerical errors over time, causing the orbits slowly to diverge from the correct trajectories.

By using a planet module, analytic perturbation solutions can be used which avoid the shortcomings of the methods described above. Perturbation solutions typically describe the perturbed orbit of a planet by expressing the state vectors as a trigonometric series. These series are valid over a limited period of time, after which they start to diverge. Examples of perturbation solutions used in Orbiter are the VSOP87 solution for the 8 major planets and the sun, or the ELP2000 solution for the moon.

Planet modules can also define an atmosphere model for the celestial body. Atmosphere models return atmospheric data (temperature, density and pressure) at a specified altitude (and other optional parameters, such as geographic position and time). Atmospheric models can be implemented either directly in the planet module, or in a separate plugin module. Putting the atmosphere model into a separate plugin makes it easier to swap models later.

The following sections give a brief introduction into the design of planet modules. A general knowledge of writing orbiter plugins is assumed.

2.2.1 First steps

Create a new DLL project for your planet module, e.g. in Orbitersdk\samples\MyPlanet. Set up all the usual include and library paths for Orbiter plugins. Add orbiter.lib and orbitersdk.lib as additional dependencies.

2.2.2 The CELBODY2 interface class

The communication between the Orbiter core and the planet module is performed via callback functions defined in the CELBODY and CELBODY2 classes. (CELBODY2 is derived from CELBODY and contains all the properties of the base class, plus a significantly extended atmospheric parameter interface.) The CELBODY interface is retained for backward compatibility, but all new planet modules should refer to the CELBODY2 interface.

We now need to the class interface for the new planet module by deriving a custom class from CELBODY2. Create a new header file in your project, e.g. MyPlanet.h, and add the following:

```
#include "OrbiterAPI.h"
#include "CelbodyAPI.h"

class DLLEXPORT MyPlanet: public CELBODY2 {
public:
    MyPlanet (OBJHANDLE hObj);
    void clbkInit (FILEHANDLE cfg);
    int clbkEphemeris (double mjd, int req, double *ret);
    int clbkFastEphemeris (double simt, int req, double *ret);
};
```

OrbiterAPI.h contains the general API interface, and CelbodyAPI.h contains the planet module-specific interface, in particular the CELBODY, CELBODY2 and ATMOSPHERE classes.

The clbkEphemeris and clbkFastEphemeris methods are callback functions which Orbiter will call whenever the planet positions and velocities ("ephemerides") need to be updated. They will be described in more detail below. The clbkInit method is called by Orbiter after the planet module has been loaded. It receives a file handle for the planet's configuration file. This allows the module to read configuration parameters from the file.
The CELBODY2 interface contains a few more methods related to defining an atmospheric model. These will be discussed below. Check the API Reference manual for a complete list of class methods.

To implement the methods in our MyPlanet class, create a source file in your project, e.g. MyPlanet.cpp. Add the following lines:

```
#define ORBITER_MODULE
#include "MyPlanet.h"

MyPlanet::MyPlanet (OBJHANDLE hObj): CELBODY2 (hObj)
{
  // add constructor code here
}

void MyPlanet::clbkInit (FILEHANDLE cfg)
{
  // read parameters from config file (e.g. tolerance limits, etc)
  // perform any required initialisation (e.g. read perturbation terms from data files)
}

bool MyPlanet::bEphemeris() const
{
  return true;
  // class supports ephemeris calculation
}

int clbkEphemeris (double mjd, int req, double *ret)
{
  // return planet position and velocity for Modified Julian date mjd in ret
}

int clbkFastEphemeris (double simt, int req, double *ret)
```

```
{
   // return interpolated planet position and velocity for simulation time simt in ret
}
```

The first line defining ORBITER_MODULE is required to ensure that all initialisation functions are properly called by Orbiter.

clbkEphemeris and clbkFastEphemeris are the functions which will contain the actual ephemeris calculations for the planet at the requested time. clbkEphemeris is only called by Orbiter if the planet's state at an arbitrary time is required (for example by an instrument calculating the position at some future time). When Orbiter updates the planet's position for the next simulation time frame, the clbkFastEphemeris function will be called instead. This means that clbkFastEphemeris will be called at each frame, each time advancing the time by a small amount. This can be used for a more efficient calculation. Instead of performing a full series evaluation, which can be lengthy, you may implement an interpolation scheme which performs the full calculation only occasionally, and interpolates between these samples to return the state at an intermediate time.

For both functions, the requested type of data is specified as a group of EPHEM_xxx bitflags in the *req* parameter. This can be any combination of position and velocity data for the celesital body itself and/or the barycentre of the system defined by the body and all its children (moons). The functions should calculate all required data, either in cartesian or polar coordinates, and fill the *ret* array with the results. *ret* contains 12 entries, used as follows:

ret[0-2]: true position
ret[3-5]: true velocity
ret[6-8]: barycentric position
ret[9-11]: barycentric velocity

Only the fields requested by *req* need to be filled. In cartesian coordinates, the position fields must contain the *x*, *y* and *z* coordinates in [m], and the velocity fields must contain the velocities d*x*/d*t*, d*y*/d*t*, d*z*/d*t* in [m/s]. In spherical polar coordinates, the position fields must contain longitude φ [rad], latitude θ [rad] and radial distance *r* [AU], and the velocity fields must contain the polar velocities dφ/d*t* [rad/s], dθ/d*t* [rad/s] and d*r*/d*t* [AU/s].
The functions should indicate the fields actually calculated via the return value. This is in particular important if not all requests could be satisified (e.g. position and velocity was requested, but only position could be calculated). The return value is interpreted as a bitflag that can contain the same EPHEM_xxx flags as the *req* parameter. If all requests could be satisfied, it should be identical to *req*. In addition, the return value should contain additional flags indicating the properties of the returned data, including EPHEM_POLAR if the data are returned as spherical polar coordinates, or EPHEM_TRUEISBARY if the true and barycentric coordinates are identical (i.e. the celestial body does not have child bodies).

2.2.3 The API interface

Next, we need to define the API interface that will allow Orbiter to load an instance of the celestial body interface. This is done by implementing the InitInstance and ExitInstance functions in MyPlanet.cpp:

```
DLLCKBK CELBODY *InitInstance (OBJHANDLE hBody)
{
  // instance initialisation
  return new MyPlanet;
}

DLLCLBK void ExitInstance (CELBODY *body)
{
  // instance cleanup
  delete (MyPlanet*)body;
}
```

InitInstance and ExitInstance are called by Orbiter each time an instance of the planet is loaded or discarded. There are also functions InitModule and ExitModule, which are called only once per simulation run, and can be used to initialise and clean up global resources:

```
DLLCLBK void InitModule (HINSTANCE hModule)
{
  // module initialisation
}

DLLCLBK void ExitModule (HINSTANCE hModule)
{
  // module cleanup
}
```

Because usually only a single instance of a specific planet object is created during a simulation, the difference between InitInstance and InitModule is not as significant here as it is for vessel modules. The InitModule and ExitModule methods can be omitted if the module doesn't need any global parameter initialisation.

2.3 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), although local temperature and pressure variations can be implemented by customised atmosphere models.

2.3.1 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.

2.3.2 A more sophisticated atmosphere

Where the simple model described above is not adequate, a more detailed atmospheric model can be implemented in a plugin module. This section assumes that a module for the celestial body has already been created, as outlined in Section 2.2.

The atmosphere model interface is described by the ATMOSPHERE class defined in CelbodyAPI.h. To create a custom atmosphere model, create a new header file in your planet project, e.g. MyAtmosphere.h. The atmosphere class interface should look like

```
#include "OrbiterAPI.h"
#include "CelbodyAPI.h"

class DLLEXPORT MyAtmosphere: public ATMOSPHERE {
public:
  MyAtmosphere (CELBODY2 *body);
  const char *clbkName () const;
  bool clbkConstants (ATMCONST *atmc) const;
  bool clbkParams (const PRM_IN *prm_in, PRM_OUT *prm_out);
};
```

The constructor takes the CELBODY2 class instance of the associated celestial body as a parameter.

The clbkName callback function should return a short name identifying the model.

The clbkConstants callback function should return in atmc some basic atmosphere parameters, such as the mean density and pressure at ground level, gas constant and ratio of specific heats, as well as some rendering parameters.

Note that some of the parameters returned by clbkConstants may be overwritten by the settings defined in the celestial body's configuration file. Configuration file entries take precedence over clbkConstants.

The clbkParams callback function should return atmospheric temperature, density and pressure at the location specified by the data in the prm_in parameter. Simple models may depend on altitude only, but more sophisticated models can make use of the additional parameters such as position (longitude and latitude), solar flux, geomagnetic index, and date.

Create a source file, e.g. MyAtmosphere.cpp, to implement the actual model. A very simplistic implementation may look like this:

```
#include "MyAtmosphere.h"

static double T0       = 288.0;  // ground level temperature [K]
static double p0       = 101325; // ground level pressure [Pa]
static double rho0     = 1.2250; // ground level density [kg/m^3]
static double R        = 286.91; // gas constant
static double gamma    = 1.4;    // ratio of specific heats
static double altlimit = 200e3;  // cutoff altitude
static double C        = rho0/p0;

MyAtmosphere::MyAtmosphere (CELBODY2 *body): ATMOSPHERE (body)
{
}

const char *MyAtmosphere::clbkName () const
{
  static char *name = "Simple";
  return name;
}

bool MyAtmosphere::clbkConstants (ATMCONST *atmc) const
{
  atmc->p0       = p0;
  atmc->rho0     = rho0;
  atmc->R        = R;
  atmc->gamma    = gamma;
  atmc->altlimit = altlimit;
  return true;
```

```
}

bool MyAtmosphere::clbkParams (const PRM_IN *prm_in, PRM_OUT *prm_out)
{
  double z = (prm_in->flag & PRM_ALT ? prm_in->alt : 0.0);
  if (z < 200e3) {
    double scale = exp (-C*z);
    prm_out->T = T0;
    prm_out->rho = rho0 * scale;
    prm_out->p   = p0   * scale;
    return true;
  } else {
    prm_out->T   = 0;
    prm_out->rho = 0;
    prm_out->p   = 0;
    return false;
  }
}
```

The above example serves only as an illustration. The actual atmosphere models provided with the Orbiter distribution are more complex. For some background on the supported Earth atmosphere models, see the technical note in Doc\Technotes\earth_atm.pdf.

Now we need to link the atmosphere model into the celestial body interface. This can be done with the SetAtmosphere function of the CELBODY2 class. Add the following statement to the clbkInit method of your MyPlanet definition:

```
#include "MyAtmosphere.h"

void MyPlanet::clbkInit (FILEHANDLE cfg)
{
  SetAtmosphere (new MyAtmosphere (this));
}
```

The atmosphere instance will be destroyed automatically when the planet instance is deleted.

2.3.3 External atmosphere modules

Instead of implementing the atmosphere model inside the planet module, it can can also be implemented in a separate plugin module. This makes it easier to exchange the atmosphere model for a different one later on, without having to have access to the rest of the planet module code.

To implement the atmosphere as a separate module, create a new DLL project for it. Add the MyAtmosphere.h and MyAtmosphere.cpp files created in the previous section to the project. Since the atmosphere is now defined in its own module, add the line

```
#define ORBITER_MODULE
```

on top of MyAtmosphere.cpp

In addition, you need to define an API interface to the module code. It should look like this:

```
DLLCLBK void InitModule (HINSTANCE hModule)
{
  // module initialisation
}

DLLCLBK void ExitModule (HINSTANCE hModule)
{
  // module cleanup
}

DLLCLBK ATMOSPHERE *CreateAtmosphere (CELBODY2 *cbody)
```

```
{
  return new MyAtmosphere (cbody);
}

DLLCLBK void DeleteAtmosphere (ATMOSPHERE *atm)
{
  delete (MyAtmosphere*)atm;
}
```

By convention, external planetary atmosphere modules should be placed in the Modules\Celbody\<Name>\Atmosphere folder, where <Name> is the celestial body's name. So in our case, Modules\Celbody\MyPlanet\Atmosphere\MyAtmosphere.dll.

We now need to modify the MyPlanet code to allow it to load its atmosphere interface from an external module. Replace the SetAtmosphere statement in the clbkInit function with

```
void MyPlanet::clbkInit (FILEHANDLE cfg)
{
  LoadAtmosphereModule ("MyAtmosphere");
}
```

However, this causes the atmospheric module name to be hardcoded in the planet module. A more flexible method is to specify the atmospheric module in the celestial body's configuration file, using the MODULE_ATM entry. Our MyPlanet.cfg file might look like this:

```
NAME = MyPlanet
MODULE = MyPlanet
MODULE_ATM = MyAtmosphere
```

If the MODULE_ATM entry is defined in the configuration file, then the default CELBODY2::clbkInit implementation will load the atmosphere module directly, so we only need to make sure to call the base class method:

```
void MyPlanet::clbkInit (FILEHANDLE cfg)
{
  CELBODY2::clbkInit (cfg);
}
```

Calling the CELBODY2::clbkInit method also enables another interesting feature: Before reading the MODULE_ATM entry in the planet configuration file, Orbiter scans the Config\<Name>\Atmosphere.cfg file for an entry "Model" and uses that, if present. This file is written by the *Atmosphere configuration* tool in the Extra tab of the Orbiter launchpad, which provides a convenient method for users to change atmosphere models. This mechanism allows to add new atmosphere modules without the need to change any configuration files. As long as the atmosphere DLL modules are placed in the correct location (Modules\Celbody\<Name>\Atmosphere), they will be scanned automatically by the atmosphere selector tool.

2.3.4 Adding and replacing atmosphere models

Most of the celestial body modules in the default Orbiter distribution have built-in support for external atmosphere modules, and some of them (Earth, Mars and Venus) come with one or several atmosphere modules. To add additional choices for atmosphere models for a body, create one as outlined above, and simply drop the DLL library into the Modules\Celbody\<Name>\Atmosphere folder. If that folder doesn't exist yet, you have to create it. The user can then select the new model from the Extra tab in the Orbiter Launchpad (Celestial body configuration | Atmosphere configuration).

For best support of the atmosphere model selection tool, your atmosphere module should contain two additional API functions:

```
DLLCLBK char *ModelName ()
```

```
{
  static char *name = "MyAtmosphere";
  return name;
}

DLLCLBK char *ModelDesc ()
{
  static char *desc = "My custom atmosphere model";
  return desc;
}
```

The string returned by the ModelName function represents the model in the dialog's selection list box. The string returned by ModelDesc should contain a short description (max 256 characters displayed in the dialog box when the model is selected.

If you don't want to design your own custom atmosphere model, you can quickly add atmospheres to planets by replicating existing ones. Simply copy an atmosphere module from the Modules\Celbody\<Name>\Atmosphere folder of one planet to that of another one. It then becomes available in the list of atmospheres for that planet. Note that the module only provides the physical atmospheric parameters. You will still have to edit the definition file to provide visual effects.

Of course, replicating an atmosphere should be regarded as a quick and dirty trick for experimentation. Atmospheres are always tailor-made for specific bodies, and don't realistically fit anywhere else.

2.3.5 Earth default atmosphere models

The Orbiter distribution contains three Earth atmosphere models that can be selected by the user from the Extra tab in the Launchpad dialog. See *Doc\Technotes\earth_atm.pdf* for further details on the different models.

Jacchia71-Gill Atmosphere Model. This is an implementation of the Jacchia-71 (J71) model[1], using a polynomial series approximation by Gill[2]. It uses a static US Standard Atmosphere model below 90km, and a diffusion-equilibrium solution between 90 and 2500km altitude. The only model parameter is the exospheric temperature.

NRLMSISE-00 Atmosphere Model. This model is based on the MSISE90 model, with the addition of further corrections based on observation data. MSISE90 provides the neutral temperature and density from ground level to thermospheric altitudes. Unlike the Jacchia models, the low-altitude data are not static, but vary with location.

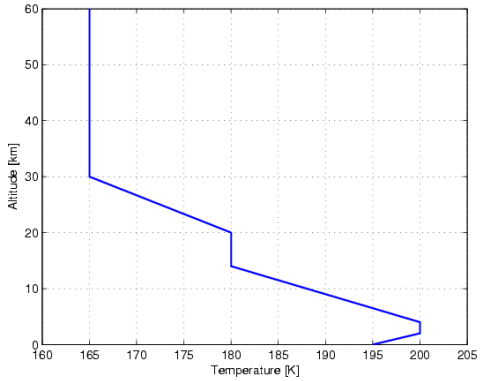
Orbiter 2006 Legacy model. This is the model that was used in the Orbiter 2006 Edition. It is based on a static standard model[3] below 105km, and assumes constant temperature and exponentially decaying density and pressure between 105 and 200km. This model underestimates density and pressure above ~120km, which reduces the orbit decay of object in low Earth orbit.

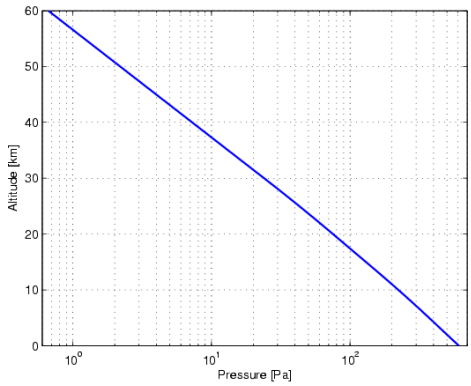
In addition, the atmosphere can be disabled for testing/debugging purposes.

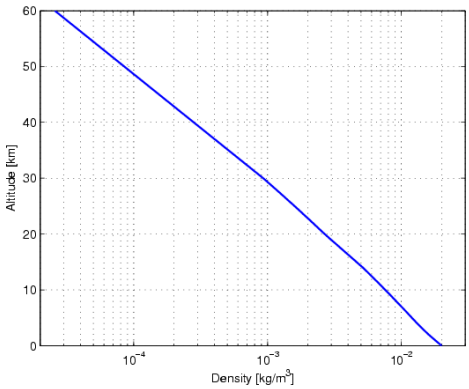
2.3.6 Mars atmosphere

Orbiter uses the following atmospheric parameter profiles for Mars:

Altitude [km]	0	2	4	14	20	30
Temperature [K]	195	200	200	180	180	165
Pressure [Pa]	610.0	499.5	410.1	145.1	75.2	23.9
Density [kg m^{-3}]	0.02	0.0160	0.0131	0.0052	$2.7 \cdot 10^{-3}$	$9.3 \cdot 10^{-4}$







Atmospheric parameters:

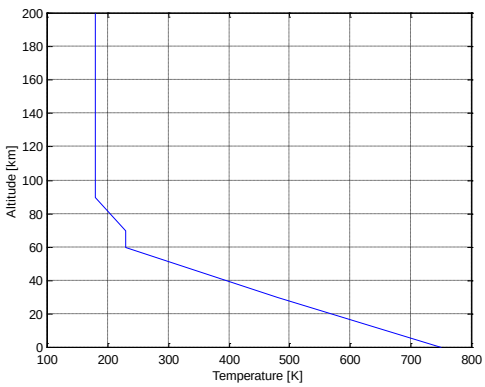
Surface pressure:	p_0 = 610.0 Pa
Surface density:	ρ_0 = 0.020 kg m^{-3}
Ratio of specific heats:	γ = 1.2941
Specific gas constant:	R = 188.92 J K^{-1} kg^{-1}

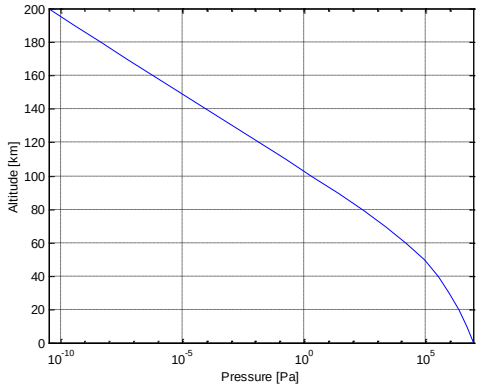
Orbiter defines the upper atmosphere altitude limit as 100 km.

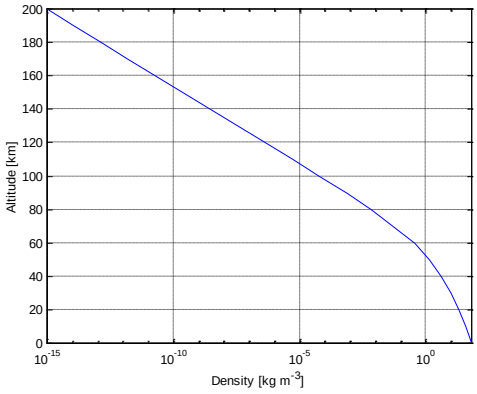
2.3.7 Venus atmosphere

Orbiter uses 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.

2.3.8 The speed of sound

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

$$a = \sqrt{\gamma RT}$$

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] L. G. Jacchia, "Revised static models of the thermosphere and exosphere with empirical temperature profiles", SAO Special Report 332, 1971.

[2] E. Gill, "Smooth bi-polynomial interpolation of Jacchia 1971 atmospheric densities for efficient satellite drag computation", DLR-GSOC, IB 96-1, 1996.

[3] J. D. Anderson, Jr. “Introduction to Flight”, 4th edition, McGraw-Hill, 2000.