ORBITER
Programmers's Guide

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

02 December 2002

Contents

1 Spacecraft design

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

1.1 Vessel module callback functions

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

1.1.1 Vessel creation and destruction

ovcInit

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

```
#include "orbitersdk.h"

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

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

ovcExit

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

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

1.1.2 Reading and saving vessel states

ovcLoadStateEx

Whenever a simulation is started, Orbiter loads the current status of all vessels from a scenario file. The scenario contains all information required to completely define the status of a vessel at a given time (its position, velocity, thruster levels, fuel levels, etc.) Most modules will need to save and load specific parameters of their own, which are not recognised by Orbiter's generic scenario parser. For this purpose, Orbiter will call the ovcLoadStateEx callback function to allow the module to process its own scenario data.
If the module does not require any non-standard status parameters, ovcLoadStateEx need not be defined. Orbiter will then automatically parse its own generic data. For a list of generic vessel data in a scenario file, see section *Scenario files* in the Orbiter User Manual.

If the module does implement ovcLoadStateEx, it should define a loop which reads lines from the scenario by using the oapiReadScenario_nextline function. Any lines not recogised by the

module should be passed on to Orbiter by using the VESSEL::ParseScenarioLineEx function, to allow initialisation of generic data.

This is a typical implementation of ovcLoadStateEx:

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

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

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

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

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

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

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

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

1.2 Creating engines

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

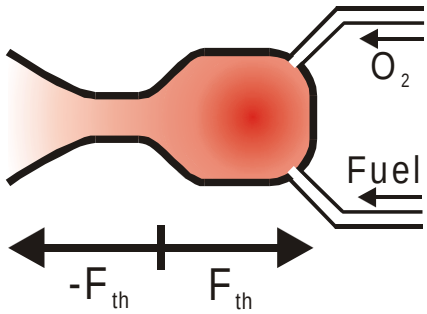
1.2.1 A bit of theory

Thrust force

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

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

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



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

How long will my fuel last?

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

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

Pressure-dependent thrust efficiency

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

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

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

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

and therefore

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

and likewise

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

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

Thrust level

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

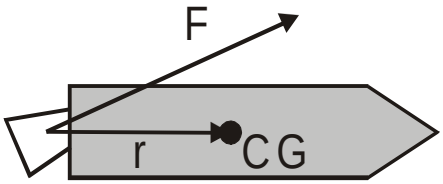
$$F(p) = F_{\max}(p) \cdot L$$

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

Thruster placement and thrust direction

The effect of a thruster depends on its placement on the vessel, and the direction in which the thrust force is generated. In the most general case, a thruster will produce both a linear acceleration (due to a force) and an angular acceleration (do to torque).

Torque is generated if the force vector does not pass through the vessel's centre of gravity (CG)



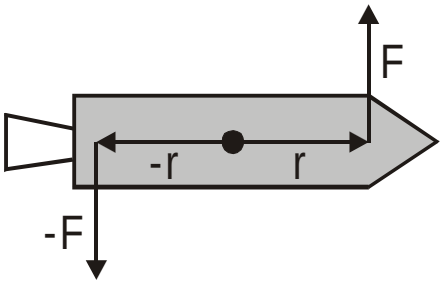
The torque is then given by the cross product

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

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

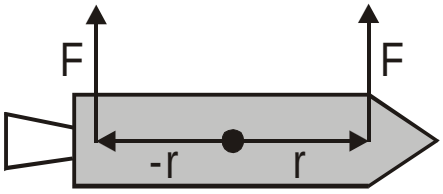
Attitude thrusters: Rotation

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



Attitude thrusters: Translation

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



Engine gimbal and thrust vectoring

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

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

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

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

1.2.2 Putting it all into the module

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

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

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

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

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

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

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

$$F' = F \cdot eff$$

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

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

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

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

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

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

Grouping thrusters
Although it is possible to address thrusters individually in your module, it is often easier to engage them in groups. Groups are also required to activate manual user thruster control via the keyboard or joystick, and the automatic navigation modes such as *killrot*.
Orbiter has a number of standard thruster groups, such as THGROUP_MAIN, THGROUP_RETRO, THGROUP_HOVER, and a full set of attitude thruster groups. For a full listing, see VESSEL::CreateThrusterGroup in the Reference Manual.
It is the responsibility of the vessel designer to make sure that thrusters are grouped in a sensible way. For example, whenever the user presses the "+" key on the numerical keypad, all thrusters in THGROUP_MAIN will fire. If the thrusters grouped in THGROUP_MAIN behave in an unexpected or non-intuitive way it will be confusing to the user. Furthermore, if attitude thrusters are not appropriately grouped, some or all of the navigation modes may fail.
To group thrusters, use the CreateThrusterGroup command:

```
void MyVessel::SetClassCaps (FILEHANDLE cfg)
```

```
{
    ...
    thg_main = CreateThrusterGroup (th_main, 2, THGROUP_MAIN);
    ...
}
```

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

1.2.3 Defining exhaust flames

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

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

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

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

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

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

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

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

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

1.3 Rendering re-entry flames

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

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

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

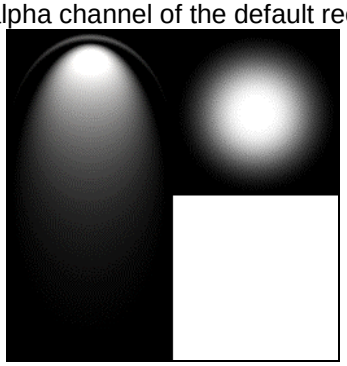
$$s = \min\left(1, \frac{P - P_0}{5P_0}\right)$$

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

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

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

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



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

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

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

1.4 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.

1.4.1 Semi-automatic animation

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

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

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

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

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

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

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

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

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

```
void MyVessel::DefineAnimations()
{
  static UINT groups[4] = {5,6,10,11}; // participating groups
  static ANIMCOMP nosewheel = {
    groups, 4,                  // group list and # groups
    0.3, 1.0,                   // limiting states
    0,-1.0,8.5,                 // rotation reference
    1.0,0.0,0.0,                // rotation axis
    (float)(-0.5*PI),           // rotation range
    0,                          // mesh no.
    0,                          // not used
    MESHGROUP_TRANSFORM::ROTATE // transform type
  };

  anim_gear = RegisterAnimSequence (0.0);
  AddAnimComp (anim_gear, &nosewheel);
}
```

The ANIMCOMP structure defines all the parameters of the animation. In this case:

groups — pointer to a list of indices (starting with 0) of mesh groups participating in the animation.

4	defines the number of groups to be part of the animation (must be ≥ 1).
0.3, 1.0	specifies that the animation should take place between states 0.3 and 1.0 of the animation sequence. This means that the wheel will remain fully retracted between states 0.0 and 0.3, and will gradually be deployed between states 0.3 and 1.0. This allows us to perform a different animation first, for example open the gear doors.
0,-1.0,8.5	defines a reference point (in the vessel's frame of reference) for the rotation.
1.0,0.0,0.0	defines the axis around which the rotation takes place.
(float)(-0.5*PI)	the angular range over which the rotation is performed (in radians)
0	the mesh index (starting with 0 for the vessel's first mesh)
0	the ngroup member of the MESHGROUP_TRANSFORM struct which is not used here.
MESHGROUP_TRANSFORM::ROTATE	the animation type (rotation)

RegisterAnimSequence() defines a new sequence. The sequence identifier is stored in anim_gear. Parameter "0.0" indicates that the wheel is defined in its retracted state in the mesh.
AddAnimComp() adds the nosewheel animation to the sequence.

Additional animations can be added to the same sequence by defining additional ANIMCOMP structures and adding them to anim_gear with AddAnimComp().

Additional sequences (for example to animate cargo doors) can be added by additional RegisterAnimSequence() calls.

Note that all ANIMCOMP structures must be defined static because Orbiter does not create a local copy. This means that the animations are global properties of the vessel class.

Performing the animation:
To animate the nose wheel now, we need to manipulate the animation sequence state by calling SetAnimState() 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;
    }
    SetAnimState (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, SetAnimState() is called to perform the animation.

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

1.4.2 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`.

1.5 Designing instrument panels

1.5.1 Defining a panel

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

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

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

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

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

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

```
      MAKEINTRESOURCE (IDB_PANEL1)));
    oapiSetPanelNeighbours (-1, -1, -1, 0);
    // register areas for panel 1 here
    break;
  case 2: // left side panel
    oapiRegisterPanelBackground (LoadBitmap (hDLL,
      MAKEINTRESOURCE (IDB_PANEL2)));
    oapiSetPanelNeighbours (-1, 0, -1, -1);
    // register areas for panel 2 here
    break;
  }
  return true;
}
```

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

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

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

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

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

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

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

1.5.2 Defining active panel areas

< To be completed >

1.5.3 The mouse event handler

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

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

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

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

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

The following mouse events are available:

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

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

1.5.4 The redraw event handler

< To be completed >