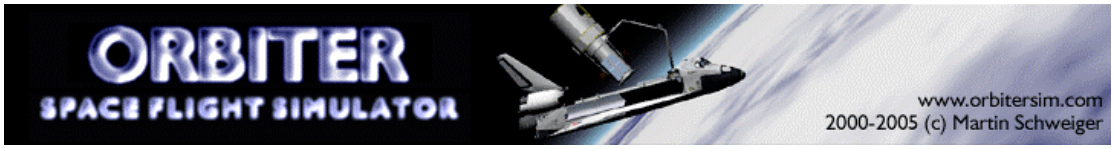
ORBITER
Developer's Guide: 3D Models

 16 January 2005
Orbiter home: www.medphys.ucl.ac.uk/~martins/orbit/orbit.html or www.orbitersim.com



Contents

1 Introduction

This document contains guidelines on developing new 3D models for spacecraft visualisation in ORBITER. It is aimed at addon developers, or anyone who wants to extend ORBITER's functionality.

2 How to create a new spacecraft class

To add a new spacecraft class to ORBITER, the following steps must be performed:

- Define the *physical parameters* of the new spacecraft class in a configuration file in the `Config` subdirectory.
- Create a *surface mesh* which defines the ship's visual appearance, in the `Meshes` subdirectory.
- Optionally, add any *textures* used by the vessel, to the `Textures` subdirectory.
- Add a *scenario* which includes one or more spacecraft of the new class in the `Scenarios` subdirectory.

The above steps allow you to create a basic ship with generic parameters. To fully customize your new spacecraft one more step is required:

- Add a DLL module for the new vessel class which customises its behaviour (moving parts, custom cockpit panels, custom flight model, etc.) in the `Modules` subdirectory.

Refer to the OrbiterAPI document for information on writing vessel modules for Orbiter. A number of sample modules with source code is contained in the Orbitersdk\samples folder.

Note that every spacecraft class *must* have a configuration file, even if all its parameters are defined in a DLL module. (In this case, the only entry in the configuration file may be the module name.)

3 The vessel class configuration file

Configuration files are ASCII text files which can be edited with any text editor capable of writing plain text files (e.g `notepad`). All vessel configuration files are by default located in ORBITER's `Config` subdirectory (unless the *ConfigDir* entry in `Orbiter.cfg` points to a different directory).

Below is a description of the default vessel configuration options recognised by Orbiter. Note that not all options need to be present in a configuration file. In particular vessels defined via costomised modules may specify various parameters directly in the module. Furthermore, vessel modules may read additional custom parameters not listed here from the configuration file.

Item	Type	Description
BaseClass	S	Optional; parent class. Missing entries are taken from this class. Allows the construction of class hierarchies. (Make sure not to introduce circular dependencies!)
Module	S	Optional; name of plugin module for vessel customisation. The module must be located in the *Modules* folder.
MeshName	S	Name of the mesh used for visualisation
EnableFocus	B	*true* if vessel can receive input focus (default: true)
EnableXPDR	B	*true* if vessel carries a transponder (default: false)
XPDR	I	transponder channel (in units of 0.05 kHz from 108.0 kHz). Only used if EnableXPDR=true. This default channel may be overridden by a vessel's scenario script.
Mass	F	Vessel mass (empty) [kg]

Size	F	(Mean) vessel radius [m]
MaxMainThrust	F	Main thruster rating [N]
MaxRetroTrust	F	Retro thruster rating [N]
MaxHoverThrust	F	Hover thruster rating [N]
MaxAttitudeThrust	F	Thrust rating for reaction contol engines [N]
TouchdownPoints	V V V	3 surface contact points in local vessel coordinates. For aircraft-like configurations these are: nose wheel, left main wheel, right main wheel. (the order is important to define the "up" direction). Other spacecraft types may interpret the points differently.
CameraOffset	V	Camera position inside the vessel for cockpit view
CW	F F F F	Airflow resistance coefficients: forward, backward, transversal, vertical. Only used by legacy flight model (if no airfoils are defined in the module).
WingAspect	F	The wing aspect ratio (wingspan^2 / wing area). Used for atmospheric drag calculation in the legacy flight model.
WingEffectiveness	F	A wing form factor: ~3.1 for elliptic wings, ~2.8 for tapered wings, ~2.5 for rectangular wings. Only used by legacy flight model.
CrossSections	V	Cross sections in axis directions (z=longitudinal) [m^2]
RotResistance	V	Resistance against rotation around axes in atmosphere, where angular deceleration due to atmospheric friction is $a^{(\omega)}_{x,y,z} = -v^{(\omega)}_{x,y,z}\ \rho\ r_{x,y,z}$ with angular velocity $v^{(\omega)}$ and atmospheric density ρ.
Inertia	V	Principal moments of inertia, mass-normalised (see below) [m^2]
PropellantResource*i*	F [F]	Specs for propellant resource *i* ($i \geq 1$). First value: max. fuel capacity [kg]. Second value: fuel efficiency factor (>0, default: 1)
MaxFuel	F	Max. fuel mass [kg]. Obsolete; only used if no propellant resources are defined
Isp	F	Default value for fuel-specific impulse [m/s]: Amount of thrust [N] obtained by burning 1kg of fuel per second. Vessel modules can override this value for individual engines.
MEngineRef*i*	V	Reference position for main thruster *i* (*i*=1…)
REngineRef*i*	V	Reference position for retro thruster *i* (*i*=1…)
HEngineRef*i*	V	Reference position for hover thruster *i* (*i*=1…)
AttRef*dij*	V	Reference position for attitude thruster (for rotation around axis *d* (*d*=X,Y,Z), rotation direction *i* (*i*=1,2) and thruster index *j* (*j*=1,2)) for a total of 12 attitude thrusters
LongAttRef*ij*	V	Reference position for attitude thrusters (for linear forward/backward translation), direction *i* (*i*=1,2) and thruster index *j* (*j*=1,2)) for a total of 4 attitude thrusters
DockRef	V	Docking reference point for first docking port (obsolete)
DockDir	V	Docking approach direction for first docking port (obsolete)
DockRot	V	Longitudinal alignment direction (normal to DockDir) for first docking port (obsolete)
<*Docklist*>	List	List of positions and approach directions for docking ports (see below).
<*Attachment list*>	List	List of positions and approach directions for attachment points (see below).

(S=String, B=Bool, F=Float, V=Vector)

Notes:

- A vessel class can be derived from a different vessel class, by defining the BaseClass entry. All properties not defined in the new class configuration file are taken from the base class.

- The mesh name should not contain the file extension (.msh) and should not contain a directory path.

- The MaxFuel entry has been replaced by PropellantResource, which allows the definition of multiple propellant resources (fuel tanks).

- The DockRef, DockDir, DockRot entries have been replaced with the more versatile Docklist (see below), which allows the configuration of multiple docking ports and IDS frequencies.

```
BEGIN_DOCKLIST
   <Dock-spec 0>
   <Dock-spec 1>
   . . .
   <Dock-spec n-1>
END_DOCKLIST
```

where *<Dock-spec i>*:

<x_i> <y_i> <z_i> <dx_i> <dy_i> <dz_i> <rx_i> <ry_i> <rz_i> [<ids-channel>]

<x_i> <y_i> <z_i> is the reference position of the docking port in the vessel's local coordinates. *<dx_i> <dy_i> <dz_i>* is the direction in which a ship approaches the docking port in the station's local reference frame.

<rx_i> <ry_i> <rz_i> is a reference direction perpendicular to the approach direction used for aligning an approaching ship's rotation along its longitudinal axis.

<ids-channel> is an optional parameter which allows to define the channel for an IDS (Instrument Docking System) transmitter for the dock. The value is an integer from which the frequency is calculated by $f = f_{min}$ + *<ids-channel>* * 0.05 kHz, where f_{min} = 108.0 kHz. The IDS setting can be overridden by individual vessels via the IDS option in the scenario file. Defining the IDS in the config file is usually only useful for objects with a single instance, for example space stations.

- The attachment list is similar to the docklist: it allows to specify points at which vessels can be connected to each other. Unlike docking ports, attachment points define parent-child hierarchies, and each attachment point is either a parent or a child port. For more details see the *Vessel attachment management* section in the API Reference Manual.

```
BEGIN_ATTACHMENT
   <Attach-spec 0>
   <Attach-spec 1>
   . . .
   <Attach-spec n-1>
END_ATTACHMENT
```

where *<Attach-spec i>*:

<type> <x_i> <y_i> <z_i> <dx_i> <dy_i> <dz_i> <rx_i> <ry_i> <rz_i> <id>

<type> is a single character: 'P' – "attach to a parent", or 'C' – "attach to a child".
The next 9 entries define the attachment position and direction in the same way as docking ports.
<id> is a string of up to 8 characters used for defining compatibility between attachment points.

- **Inertia tensor J:** Relates angular momentum and angular velocity: $\mathbf{L} = J \cdot \boldsymbol{\omega}$

$$J = \frac{1}{M} \int_{Vol} m(r) \begin{pmatrix} y(r)^2 + z(r)^2 & x(r)y(r) & x(r)z(r) \\ y(r)x(r) & x(r)^2 + z(r)^2 & y(r)z(r) \\ z(r)x(r) & z(r)y(r) & x(r)^2 + y(r)^2 \end{pmatrix} dr$$

where M is the total vessel mass, and the integration is over the vessel volume. Note that this definition normalises by M, so the unit of J is [m^2]. The principal moments of inertia (PMI) J_x, J_y, J_z required by the configuration file are the diagonal elements of J in a reference frame in which J is diagonal:

$$\hat{J} = \begin{pmatrix} J_x & 0 & 0 \\ 0 & J_y & 0 \\ 0 & 0 & J_z \end{pmatrix}$$

The SDK contains a simple tool to calculate the inertia tensor for a given mesh: `Orbitersdk\utils\shipedit.exe`. The tool requires "well behaved" meshes (composed of closed compact surfaces) and assumes a homogeneous density distribution inside the mesh. The latter is not very realistic, so the results must be interpreted carefully. They should still serve as a good starting point for experimentation.

3.1 Configuration files for individual vessels

A vessel only requires an individual definition file if it is not an instance of a vessel class. In this case the format for the vessel's .cfg file is identical to the vessel class .cfg files described above.

4 The mesh file

ORBITER uses a proprietary mesh file format. Mesh files are ASCII text files. (A binary format may be introduced in the future). Mesh files are located in the `Meshes` subdirectory unless the MeshDir entry in Orbiter.cfg points to a different directory.

ORBITER meshes are defined in a left-handed coordinate system. Vessel meshes should be oriented such that the *vessel's nose* (or more precisely, its *main thrust direction*) points in the positive z-direction, the positive x-axis points *right*, and the positive y-axis points *up*.

The units for vertex coordinates are *meters* [m].

Mesh file format:

`MESHX1`	header
`GROUPS` *<n>*	*<n>:* number of groups
<group 1>	group spec 1
<group 2>	group spec 2
…	
<group n>	group spec *n*
`MATERIALS` *<m>*	*<m>:* number of materials
<mtrl-name 1>	material name 1
<mtrl-name 2>	material name 2
…	
<mtrl-name m>	material name *m*
<material 1>	material spec 1
<material 2>	material spec 2
…	
<material m>	material spec *m*
`TEXTURES` *<t>*	*<t>:* number of textures

<tex-name 1>	texture name 1
<tex-name 2>	texture name 2
…	
<tex-name t>	texture name *t*

Group specs:

[MATERIAL *<i>*]	material index; optional
[TEXTURE *<j>*]	texture index; optional
[TEXWRAP *<wrap>*]	texture wrap mode: *<wrap>* = U or V or UV; optional
[NONORMAL]	"no normals" flag; see below; optional
[FLAG *<f>*]	multi-purpose bit-flags; see below; optional
GEOM *<nv> <nt>*	*<nv>:* vertex count, *<nt>:* triangle count
<vtx 0>	vertex spec 0
<vtx 1>	vertex spec 1
…	
<vtx nv-1>	vertex spec *nv*-1
<tri 0>	triangle spec 0
<tri 1>	triangle spec 1
…	
<tri nt-1>	triangle spec *nt*-1

Vertex specs:

<x> <y> <z> [<nx> <ny> <nz> [<tu> <tv>]]	
	<x> <y> <z>: vertex position
	<nx> <ny> <nz>: vertex normal (optional)
	<tu> <tv>: texture coordinates (optional)

Missing normals are automatically calculated as the mean of the normals of adjacent faces. Texture coordinates are only required if the group uses a texture.

Triangle specs:

<i> <j> <k>	vertex indices (zero-based). Left-hand face is rendered.

Material specs:

MATERIAL *<mtrl-name>*	material header
<dr> <dg> <db> <da>	Diffuse colour (RGBA)
<ar> <ag> <ab> <aa>	Ambient colour (RGBA)
<sr> <sg> <sb> <sa> <pow>	Specular colour (RGBA) and specular power (float)
<er> <eg> <eb> <ea>	Emissive colour (RGBA)

4.1 Mesh groups

Meshes are divided into groups. Each group can define its own material and texture specification. For example, if you want different parts of the object to have different material properties, you need to split the mesh into groups accordingly.

Each group contains

- An optional material index. Indices ≥1 select a material of the mesh's material list. Index 0 means "default material" (which is white, diffuse and opaque). If the group doesn't specify a material index it inherits the previous group's material. The first group in the mesh *must* specify a material index, otherwise the result is undefined.
- An optional texture index. Indices ≥1 select a texture from the mesh's texture list. Index 0 means "no texture". If the group doesn't specify a texture index it inherits the previous group's texture. The first group in the mesh *must* specify a texture index, otherwise the result is undefined.
- An optional TEXWRAP flag. This defines how textures wrap around the object. "U" causes textures to wrap in the u-coordinate direction in texel space, "V" wraps in v-coordinate direction, and "UV" wraps in both directions. Default is no wrapping.

- An optional NONORMAL flag. This indicates that vertex definitions in this group don't contain normal definitions, and the first two numbers after the vertex coordinate (x,y,z) triplet is interpreted as texture coordinate (u,v) pair.
- An optional FLAG entry. This allows to specify a user-defined 32-bit flag (in hex format) whose interpretation is context-dependent. Below is a list of flags currently recognised by Orbiter:

Mesh type	Flag	Interpretation
Vessel	0x00000001	Do not use this group to render ground shadows
Vessel	0x00000002	Do not render this group
Vessel	0x00000004	Do not apply lighting when rendering this group

- A GEOM specification, defining the number of vertices and triangles in the group.
- A vertex list (see below)
- A triangle list (see below)

Vertex lists

Each group contains a vertex list, defining the positions, and optionally normal directions and texture coordinates of the vertices in the group.
Each line in the list defines a vertex, and contains up to 8 floating point numbers (separated by spaces)

- The first 3 numbers contain the cartesian vertex coordinates (x,y,z) in the object local coordinate space. Units are meters [m]
- The next 3 numbers (if present) contain the vertex normal direction (nx,ny,nz) (unless the group has set the NONORMAL flag). The normal direction is the direction perpendicular to the mesh surface at the vertex position. Orbiter needs this to generate correct lighting effects. If no normals are specified (or if the NONORMAL flag is set) Orbiter guesses the normal direction as the average of the normals of the surrounding triangles. This works well for smooth surfaces, but should be avoided for surfaces which contain sharp edges. Normal directions should be normalised, i.e. $\text{sqrt}(nx^2+ny^2+nz^2) = 1$.
- The next 2 numbers (if present) contain the vertex texture coordinates (u,v). Texture coordinates are only required if the group uses a texture (i.e. has texture index ≥1). Texture coordinates define how a rectangular 2D texture is mapped onto the object surface. Texture coordinate (0,0) refers to the lower left corner of the texture, (1,1) refers to the upper right corner. Coordinates > 1 are allowed and cause textures to repeat periodically.

Notes:

- Vertices located at sharp edges or corners require multiple entries in the vertex list, because they have multiple normal directions (in other words, the surfaces are *non-differentiable* at edges). In that case you should always define the normals in the mesh file, and not leave it to Orbiter to generate them for you. Otherwise the edges will appear unrealistically smooth.
- Likewise, vertices with multiple vertex coordinates (e.g. at the edge between two texture maps) need multiple entries in the vertex list.

Triangle lists

The group's triangle list follows immediately below the vertex list. It defines the triangles which compose the group's mesh surface.

- Each line in the list defines a triangle and consists of 3 integer numbers (i,j,k). Each of the numbers specifies a vertex from the group's vertex list (starting from 0)
- Only the "clockwise" (CW) side of each triangle is rendered: the side which, if you look at it, has the vertices arranged in a clockwise order. The opposite "counterclockwise" (CCW) side is invisible.
- If you need to render both sides of a triangle (e.g. for a thin plate) you need to define two triangles.
- If you want to flip the rendered side of a triangle (e.g. to correct for "inside out" artefacts) you need to rearrange the triangle indices in the following way:
  (i,j,k) -> (i,k,j)

4.2 Material list

Materials allow to specify the homogeneous lighting properties of a mesh group. The material lists consists of

- A header line, `MATERIALS <m>`, defining the number <m> of materials.
- A list of material *names*.
- A list of material *properties*.

Each material property specification consists of 4 RGBA quadruplets, where R, G and B define the red, green and blue components, and A is the opacity. RGB values should be between 0 and 1, but can be > 1 for special effects. A *must* be between 0 (fully transparent) and 1 (fully opaque).

- The first line specifies the *diffuse material colour*. This is the colour that is diffusely (in all directions) reflected from an illuminated surface.
- The second line specifies the *ambient material colour*. This is the colour of an unlit surface.
- The third line specifies the specular colour. This is the colour of light reflected by a polished surface into a narrow beam. The *power* entry specifies the width of the cone into which specular light is reflected. Higher values mean a narrower cone, i.e. sharper reflections. Typical values are around 10. If omitted, the default value for power is 0.
- The fourth line specifies the *emissive colour*. This is the colour of light emitted by a glowing surface.

4.3 Texture list

The texture list contains the names of texture files used by the various mesh groups. Texture names should contain file extensions ".dds" but no directory paths. Textures must be located in Orbiter's `Textures` subdirectory.

Notes:

- Textures must be in DDS format ("Direct Draw Surface"). A DirectX SDK tool, `dxtex`, which is included in the Orbiter SDK package, allows to convert BMP bitmaps into DDS.
- You should store the textures either in DXT1 compressed format (opaque textures or textures with binary transparency), or in DXT5 compressed format (for textures with continuous transparency).
- For maximum compatibility, avoid textures larger than 256x256 pixels, because of limitations of some older graphics cards.
- If a texture is to be dynamically updated during the simulation (e.g. instrument panels in virtual cockpits), the texture name should be followed by the flag 'D'. Orbiter will decompress these textures to allow more efficient dynamic updates.

4.4 Performance optimisation

To achieve the best results with your new mesh, consider the following points:

- Texture groups which use the same texture should be stored in sequence in the mesh. Unnecessary switching between textures can degrade performance if textures must be swapped in and out of video memory.
- Within a sequence of groups using the same textures, groups which use the same material should be stored in sequence. Again, this avoids the need of switching render parameters.
- Avoid large numbers of very small groups. If small groups use the same parameters (material, texture, etc.) they should be merged into a single group.
- Groups which use transparent materials or textures should be sorted to the end of the mesh. If transparent groups overlap, the innermost ones should be listed before the outer ones.
  In order to render transparency correctly, DirectX requires the scene seen through the transparent object to be fully built before the transparent object itself is rendered. Any objects rendered after the transparent object will be masked by it.
- Objects with transparency and specular reflection are more expensive to render than opaque and diffusive objects, so use these features sparingly.

- And most importantly, *keep the vertex count low!* (See section 6.1)

4.5 Mesh converters

If you want to convert an existing model into an Orbiter mesh, check the Orbiter web forum for mesh converters created by other users. There is currently a converter which converts from Truespace asc format, which many 3D editors can export. If you have written your own mesh editor or converter, publish it!

5 Creating a scenario

To actually fly your new creation in Orbiter, you need to create a scenario which contains one (or more) ships of the new class. The easiest way to do this is by editing one of the existing scenario files in the Scenarios subdirectories. For example, to try a new vessel class, you could:

- Open the "Habana spaceport.scn" file in notepad
- Replace the line
  `GL-01:DeltaGlider`
  with
  `GL-01:<new class>`
  where <new class> is the name of your new vessel class.
- Save the scenario under a new name.

When you launch this modified scenario, you will find yourself in the cockpit of your new ship, parked on a launchpad of Habana International Spaceport.

6 Publishing addons

Now that you have created and tested your new ship, you want to share it with the rest of the Orbiter community. Here is how to do it:

6.1 Function over style

It may be tempting to create an extremely detailed 3D model with tens of thousands of mesh vertices, and megabytes of textures – but don't. Remember that many people may have lower-powered computers than you, and that the sexiest spacecraft is worthless if it degrades framerate to un-playability. Also, your model may compete with tens or hundreds of other objects for processor cycles.
Therefore, aim for *low polygon count*. Creating good-looking objects with low-resolution meshes is the high art of 3D modelling for real-time applications. If you are importing an existing model into Orbiter, see whether your 3D editor has an option to reduce the polygon count (sometimes called *optimisation* or *decimation*) before converting to the Orbiter mesh format. As a rough guideline, I would suggest to keep the vertex count below 10 000 vertices for each spacecraft.

Likewise, try to limit the textures used by your object. Graphics cards have limited texture memory, and your ship will share this space with lots of other objects. Therefore I suggest

- a small number of texture maps, at dimension 256x256 or smaller.
- The use of generic texture maps (e.g. textures for solar panels etc.) which can be re-used by other models, helps reduce texture memory requirements. Have a look at the Orbiter Textures directory, to see whether an existing standard Orbiter texture is usable for your model.
- Remember that some older graphics cards do not support textures larger than 256x256. Avoid anything larger if you want to ensure compatibility.

These limits are mere suggestions for published addons to ensure reasonable performance on low end computers – you are free to ignore them. You should however mention the complexity of your model (i.e. the vertex count) and possible incompatibilities in the readme file accompanying the addon, to help users decide if they can use your addon.

Of course there are no limitations for models created for your private use.

6.2 Creating an addon package

Create a zip file containing all the components of your new model (readme file, configuration file, mesh, textures, scenario). The zip file should contain directory information of the files, so that everything ends up in the right place when the user unpacks the archive in the Orbiter home directory.
Do NOT include modified versions of standard Orbiter files (like planet configuration files or solar system configuration files) which may overwrite existing files. If your package needs to modify standard files this should be described in the readme file.
Test that the package unzips and runs ok before publishing it (ideally in a fresh Orbiter installation)

Make sure your package contains a readme file with at least the following information:

- Your name and email
- The Orbiter version for which the package was written
- A description of the package (what kind of ship, etc.)
- A list of files in the package
- Installation instructions, including required changes to standard Orbiter configuration files.
- An (approximate) vertex count so that users can estimate the performance impact

6.3 Putting it on the web

Put your new package on a web site (with a short description and an optional screen shot) and tell others about it! If you haven't got web space, submit it to one of the Orbiter repositories set up by Orbiter users. The "Related sites" page on the Orbiter home site has a list of Orbiter addon repositories. One of the most popular Orbiter archives can be found at AVSIM (www.avsim.com). For widest publicity, you should put a note with link in the Orbiter addon forum, or email the Orbiter mailing list. Finally, you may want to consider submitting a screenshot of your addon to be published on the official Orbiter web site. Check the Orbiter gallery page for a link to submission guidelines.