FlightGear READMEs

FlightGear Community

September 25, 2018

Contents

1 Introduction

This document is autogenerated from the various plain text README files found in the Docs/ directory of your FlightGear installation, presented in PDF format for ease of use. These are targetted at those delving into the internals of FG, or developing aircraft.

For help running or using FlightGear, please see Docs/getstart.pdf.

2 3DClouds

```
Configuring 3D Clouds
=====================

3D clouds can be created in two ways:
- By placing individual clouds using a command (e.g. from Nasal)
- Using the global weather function, which reads cloud definition from
  an XML file.

Placing Clouds Individually
===========================

Clouds are created using the "add-cloud" command, passing a property
node defining the location and characterstics of the cloud.
```

Location is defined by the following properties:

```
<layer>       - The cloud layer number to add the cloud to. (default 0)
<index>       - A unique identifier for the cloud in the layer. If a cloud
                already exists with this index, the new cloud will not be
                created, and 0 is returned.
<lon-deg>     - Longitude to place the cloud, in degrees (default 0)
<lat-deg>     - Latitude t place the cloud, in degrees (default 0)
<alt-ft>      - Altitude to place the cloud, relative to the layer (!) in ft
                (default 0)
<x-offset-m> - Offset in m from the lon-deg. +ve is south (default 0)
<y-offset-m> - Offset in m from the lat-deg. +ve is east (default 0)
```

The cloud itself is built up of a number of "sprites" - simple 2D textures
that are always rotated to be facing the viewer. These sprites are handled
by a OpenGL Shader - a small program that is run on your graphics card.

The cloud is defined by the following properties:

```
<min-cloud-width-m>   - minimum width of the cloud in meters (default 500)
<max-cloud-width-m>   - maximum width of the cloud (default min-cloud-width-m*1.5)
<min-cloud-height-m>  - minimum height of the cloud (default 400)
<max-cloud-height-m>  - maximum height of the cloud (default min-cloud-height-m*1.5)
<texture>             - texture file of sprites to use (default cl_cumulus.png)
<num-textures-x>      - number of cloud textures defined horizontally in the
                        texture file (default 4)
<num-textures-y>      - number of cloud textures defined vertically in the
                        texture file (default 4)
<height-map-texture>  - whether to choose the vertical texture index based on
                        sprite height within the clouds (default false)
<num-sprites>         - Number of sprite to generate for the cloud (default 20)
<min-sprite-width-m>  - minimum width of the sprites used to create the cloud
                        (default 200)
<max-sprite-width-m>  - maximum width of the sprites used to create the cloud
                        (default min-sprite-width-m*1.5)
<min-sprite-height-m> - minimum height of the spites used to create the cloud
                        (default 150)
<max-sprite-height-m> - maximum height of the sprites used to create the cloud
                        (default min-sprite-height-m*1.5)
<z-scale>             - vertical scaling factor to apply to to the sprite after
```

```
                        billboarding. A small value would create a sprite that
                        looks squashed when viewed from the side. (default 1.0)
<min-bottom-lighting-factor> - See Shading below (default 1.0)
<max-bottom-lighting-factor> - See Shading below (default min-...-factor + 0.1)
<min-middle-lighting-factor> - See Shading below (default 1.0)
<max-middle-lighting-factor> - See Shading below (default min-...-factor + 0.1)
<min-top-lighting-factor>    - See Shading below (default 1.0)
<max-top-lighting-factor>    - See Shading below (default min-...-factor + 0.1)
<min-shade-lighting-factor>  - See Shading below (default 0.5)
<max-shade-lighting-factor>  - See Shading below (default min-...-factor + 0.1)
```

```
Shading
-------
```

the [min|max]-...-lighting-factor properties allow you to define diffuse lighting
multipliers to the bottom, middle, top, sunny and shaded parts of the cloud. In
each case, individual clouds will have a random multiplier between the min and
max values used to allow for some variation between individual clouds.

The top, middle and bottom lighting factors are applied based on the pixels vertical
positon in the cloud. A linear interpolation is used either between top/middle (if
the pixel is above the middle of the cloud) or middle/bottom (if the pixel is below
the middle of the cloud).

The top factor is also applied to _all_ pixels on the sunny side of the cloud. The
shade factor is applied based on the pixel position away from the sun, linearly
interpolated from top to shade. E.g this is not a straight linear interpolation
from top to shade across the entire cloud.

The final lighting factor is determined by the minimum of the vertical factor and
the sunny/shade factor. Note that this is applied to the individual pixels, not
sprites.

```
Textures
--------
```

The texture to use for the sprites is defined in the <texture> tag.
To allow some variation, you can create a texture file containing multiple
sprites in a grid, and define the <num-textures-x/y> tags. The code
decides which texture to use for a given sprite : randomly in the x-direction

and based on the altitude of the sprite within the cloud in the y-direction
if <height-map-texture> is set. Therefore, you should put sprite textures
you want to use for the bottom of your cloud at the bottom of the texture
file, and those you want to use for the top of the cloud at the top of the
texture file.

For example, the following Nasal snippet will create a cloud immediately above the
aircraft at an altitude of 1000 ft above /environment/clouds/layer[0]/elevation-ft :

```
var p = props.Node.new({ "layer" : 0,
                         "index": 1,
                         "lat-deg": getprop("/position/latitude-deg"),
                         "lon-deg": getprop("/position/longitude-deg"),
                         "alt-ft" : 1000 });
fgcommand("add-cloud", p);
```

Moving Individual Clouds
=========================

Clouds may be moved by using the "move-cloud" command. This takes the following
property arguments.

```
<layer>      - The cloud layer number containing the cloud to move. (default 0)
<index>      - The unique identifier of the cloud to move.
<lon-deg>    - Longitude to place the cloud, in degrees (default 0)
<lat-deg>    - Latitude t place the cloud, in degrees (default 0)
<alt-ft>     - Altitude to place the cloud, relative to the layer (!) in ft
               (default 0)
<x-offset-m> - Offset in m from the lon-deg. +ve is south (default 0)
<y-offset-m> - Offset in m from the lat-deg. +ve is east (default 0)
```

Deleting Individual Clouds
===========================

Clouds may be deleted by using the "del-cloud" command. This takes the following
property arguments.

```
<layer>      - The cloud layer number containing the cloud to delete. (default 0)
<index>      - The unique identifier of the cloud to delete.
```

```
Global 3D Clouds
================
```

The global weather system uses sets of clouds defined in
FG_ROOT/Environment/cloudlayers.xml

The file has 3 distinct sections: layers, cloud boxes and clouds,
described below.

Notes for those editing clouds:
- All distances are in m. Note that this is in contrast to cloud heights
  in METAR etc. which are in ft.
- The XML file is loaded into the properties system, so you can modify
  the settings in-sim, and see the results by re-generating the cloud
  layer. The simplest way to do this is to disable METAR, and control
  the cloud layers using the Clouds dialog, and in particular the coverage.
- Texture files are in .png format, and have a transparent background.
  To make the textures easier to edit, create a black layer behind them,
  so there is some contrast between the background and the white cloud.
  Having a grid based on the texture dimensions also helps, so you don't
  bleed over the edges, which causes ugly sharp horizontal and vertical
  lines.

```
Clouds
======
```

The cloud definitions are as described above for placing individual
clouds, but no position information is used (this is defined in the
cloud box and layers below).

```
Cloud Boxes
===========
```

The <boxes> section contains definitions of groups of cloads,for example
an entire towering CB mass.

The <boxes> section contains a number of named types, which are referenced
by the <layers> section, described below. Therefore, the names used are
completely user-defined.

Each of the named section consists of one or more <box> section,
defining a particular cloud type

Each <box> section contains the following tags:

```
<type>   - The cloud to use, defined above
<count>  - The number of clouds to generate (+/- 50%)
<width>  - The x and y within which these clouds should be generated
<height> - The height within which the clouds should be generated
<hdist>  - The horizontal distribution of the clouds within the area.
           Equates to a sum of random distributions. Defaults to 1.
           1 = even distribution, 2 = distributed towards the center.
           3 = very strongly distributed towards the center.
<vdist>  - The vertical distribution of the clouds. As for hdist.
```

If the /sim/rendering/clouds3d-density is less than 1.0 (100%), then a
proportional number of clouds will be displayed.

The following example shows a stratus cloud group, which consists of 5
st-large clouds and 5 st-small clouds, distributed in a box 2000mx2000m,
and 100m high, evenly distributed.

```
<st>
  <box>
    <type>st-large</type>
    <count>5</count>
    <width>2000</width>
    <height>100</height>
  </box>
  <box>
    <type>st-small</type>
    <count>5</count>
    <width>2000</width>
    <height>100</height>
  </box>
</st>
```

Layers

```
======
```

The <layers> section contains definitions for a specific layer type.
The layer type is derived from the METAR/Weather settings by FG itself.

Each layer type is a named XML tag, i.e.: ns, sc, st, ac, cb, cu.
If a layer type is not defined, then a 2D layer is used instead.

The layer type contains one or more <cloud> definitions. This
defines a type of cloud box, and a weighting for that type (<count>).

For example, the following XML fragment will produce 3 "cb" cloud boxes
for every 1 "cu":

```
<cloud>
    <name>cb</name>
    <count>3</count>
</cloud>
<cloud>
    <name>cu</name>
    <count>1</count>
</cloud>
```

Clouds are randomly distributed across the sky in the x/y plane, but the
height of them is set by the weather conditions, with a random height range
applied, defined by <grid-z-rand>

3   Add-ons

-*- coding: utf-8; fill-column: 72; -*-

```
Add-ons in FlightGear
=====================
```

This document explains how add-ons work in FlightGear. The add-on
feature was first added in FlightGear 2017.3. This document describes an
evolution of the framework that appeared in FlightGear 2017.4.

Contents
----

1. Terminology

2. The addon-metadata.xml file

3. Add-ons and the Property Tree

    a) Add-on metadata
    b) Subtree reserved for add-on developers

4. Resources under the add-on directory

5. Persistent storage location for add-ons

6. Add-on-specific menus and dialogs

    a) Add-on-specific menus
    b) Add-on-specific dialogs

7. How to run code after an add-on is loaded

8. Overview of the C++ API

9. Nasal API

Introduction
----

fgfs can be passed the --addon=<path> option, where <path> indicates an
add-on directory. Such a directory, when used as the argument of
--addon, receives special treatment :

  1) The add-on directory is added to the list of aircraft paths.

  2) The add-on directory must contain a PropertyList file called
     addon-metadata.xml that gives the name of the add-on, its
     identifier (id), its version and possibly a few other things (see

details below).

3) The add-on directory may contain a PropertyList file called
   addon-config.xml, in which case it will be loaded into the Property
   Tree at FlightGear startup, as if it were passed to the --config
   fgfs option.

4) The add-on directory must contain a Nasal file called
   addon-main.nas. This file will be loaded at startup too, and its
   main() function run in the namespace __addon[ADDON_ID]__, where
   ADDON_ID is the add-on identifier specified in the
   addon-metadata.xml file. The main() function is passed one
   argument: the addons.Addon object (a Nasal ghost, see below)
   corresponding to the add-on being loaded. This operation is done by
   $FG_ROOT/Nasal/addons.nas at the time of this writing.

Also, the Property Tree is populated (under /addons) with information
about registered add-ons. More details will be given below.

The --addon option can be specified zero or more times; each of the
operations indicated above is carried out for every specified add-on in
the order given by the --addon options used: that's what we call add-on
registration order, or add-on load order. In other words, add-ons are
registered and loaded in the order specified by the --addon options
used.

1. Terminology
   ~~~~~~~~~~~~

add-on base path

  Path to a directory containing all of the add-on files. This is the
  path passed to the --addon fgfs option, when one wants to load the
  add-on in question.

add-on identifier (id)

  A string such as org.flightgear.addons.ATCChatter or
  user.joe.MyGreatAddon, used to uniquely identify an add-on. The add-on

identifier is declared in <path>/addon-metadata.xml, where <path> is
the add-on base path.

add-on registration

  When a --addon option is processed, FlightGear ensures that the add-on
  identifier found in the corresponding addon-metadata.xml file isn't
  already used by an add-on from a previous --addon option on the same
  command line, and stores the add-on metadata inside dedicated C++
  objects. This process is called add-on registration.

add-on loading

  The following sequence of actions:

    a) loading an add-on's addon-main.nas file in the namespace
       __addon[ADDON_ID]__
    b) calling its main() function

  is performed later (see $FG_ROOT/Nasal/addons.nas) and called add-on
  loading.

2. The addon-metadata.xml file
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Every add-on must have in its base directory a file called
'addon-metadata.xml'. This section explains how to write this file.

Sample addon-metadata.xml file
==============================

Here is an example of an addon-metadata.xml file, for a hypothetical
add-on called âFlying Turtleâ distributed by Joe User:

  <?xml version="1.0" encoding="UTF-8"?>

  <PropertyList>
    <meta>
      <file-type type="string">FlightGear add-on metadata</file-type>

```
      <format-version type="int">1</format-version>
</meta>

<addon>
  <identifier type="string">user.joe.FlyingTurtle</identifier>
  <name type="string">Flying Turtle</name>
  <version type="string">1.0.0rc2</version>

  <authors>
    <author>
      <name type="string">Joe User</name>
      <email type="string">optional_address@example.com</email>
      <url type="string">http://joe.example.com/foobar/</url>
    </author>

    <author>
      <name type="string">Jane Maintainer</name>
      <email type="string">jane@example.com</email>
      <url type="string">https://jane.example.com/</url>
    </author>
  </authors>

  <maintainers>
    <maintainer>
      <name type="string">Jane Maintainer</name>
      <email type="string">jane@example.com</email>
      <url type="string">https://jane.example.com/</url>
    </maintainer>
  </maintainers>

  <short-description type="string">
    Allow flying with new foobar powers.
  </short-description>

  <long-description type="string">
    This add-on enables something really great involving turtles...
  </long-description>

  <license>
    <designation type="string">
```

```
        GNU GPL version 2 or later
      </designation>

      <file type="string">
        COPYING
      </file>

      <url type="string">
        https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html
      </url>
    </license>

    <min-FG-version type="string">2017.4.0</min-FG-version>
    <max-FG-version type="string">none</max-FG-version>

    <urls>
      <home-page type="string">
        https://example.com/quux
      </home-page>

      <download type="string">
        https://example.com/quux/download
      </download>

      <support type="string">
        https://example.com/quux/support
      </support>

      <code-repository type="string">
        https://example.com/quux/code-repository
      </code-repository>
    </urls>

    <tags>
      <tag type="string">first tag</tag>
      <tag type="string">second tag</tag>
      <tag type="string">etc.</tag>
    </tags>
  </addon>
</PropertyList>
```

General rules
=============

We use the terms âfieldâ or ânodeâ interchangeably here to refer to
nodes of the addon-metadata.xml PropertyList file (technically, a field
always has a value, possibly empty, therefore fields are all leaf
nodes).

Leading and trailing whitespace in each field of addon-metadata.xml is
removed. All other whitespace is a priori preserved (this could depend
on the particular field, though).

Most fields are optional. In most cases, omitting a field is the same as
leaving it empty. But don't write empty tag fields, it is really too
ugly. ;-)

Name and id
===========

Nodes: /addon/name and /addon/identifier

The add-on name is the pretty form. It should not be overly long, but
otherwise isn't constrained. On the other hand, the add-on identifier
(id), which serves to uniquely identify an add-on:
  - must contain only ASCII letters (A-Z, a-z) and dots ('.');
  - must be in reverse DNS style (even if the domain doesn't exist),
    e.g., org.flightgear.addons.ATCChatter for an add-on distributed in
    FGAddon, or user.joe.FlyingTurtle for Joe User's âFlying Turtleâ
    add-on. Of course, if Joe User owns a domain name and uses it to
    distribute his add-on, he should put it here.

Authors and maintainers
=======================

Nodes: /addon/authors and /addon/maintainers

Authors are people who contributed significantly to the add-on.
Maintainers are people currently in charge of maintaining it.

It is possible to declare any number of authors and any number of maintainers---the example above shows only one maintainer for shortness, but this is not a restriction.

For each author and maintainer, you can give a name, an email address and a URL. The name must be non-empty, but the email address and URL need not be specified or may be left empty, which is equivalent. Obviously, if no email address nor URL is given for any maintainer, it is highly desirable that /addon/urls/support contains a usable URL for contacting the add-on maintainers.

The data in children nodes of /addon/maintainers may refer either to real persons or to more abstract entities such as mailing-lists. In case of a real person, the corresponding URL, if specified, is expected to be the person's home page. On the other hand, if a declared âmaintainerâ is a mailing-list, a good use for the 'url' field is to indicate the address of a web page from which people can subscribe to the mailing-list.

Short and long descriptions
===========================

Nodes: /addon/short-description and /addon/long-description

The short description should fit on one line (try not to exceed, say, 78 characters), and in general consist of only one sentence.

The long description is essentially free-form, but only break lines when you do want a line break at this point. In other words, don't wrap lines manually in the XML file: this will be automatically done by the software displaying the add-on description, according to the particular line width it uses (which can depend on the user's screen or configuration, etc.). A single \n inside a paragraph (see footnote [1]) means a hard line break. Two \n in a row (i.e., a blank line) should be used to separate paragraphs. Example:

This is a paragraph.
This is the second line of the same paragraph. It can be very, very, very long and co

This is a different paragraph. Again, don't break lines (i.e., don't press Enter) unl

Licensing terms
===============

Nodes: /addon/license/designation
       /addon/license/file
       /addon/license/url

The /add-on/license/designation node should describe the add-on
licensing terms in a short but accurate way, if possible. If this is not
practically doable, use the value âCustomâ. If the add-on is distributed
under several licenses, use the value âMultipleâ. In all cases, make
sure the licensing terms are clearly specified in other files of the
add-on (typically, at least README.txt or COPYING). Values for
/add-on/license/designation could be âGNU GPL version 2 or laterâ, âCC0
1.0 Universalâ, â3-clause BSDâ, etc.

In most cases, the add-on should contain a file containing the full
license text. Use the /add-on/license/file node to point to this file:
it should contain a file path that is relative to the add-on base
directory. This path must use slash separators ('/'), even if you use
Windows.

The /add-on/license/url node should contain a single URL if there is an
official, stable URL for the license under which the add-on is
distributed. The term âofficialâ here is to be interpreted in the
context of the particular license. For instance, for a GNU license
(GPL2, LGPL2.1, etc.), the URL domain must be gnu.org; for a CC license
(CC0 1.0 Universal, CC-BY-SA 4.0...), it must be creativecommons.org,
etc.

Minimum and maximum FlightGear versions
=======================================

Nodes: /addon/min-FG-version and /addon/max-FG-version

These two nodes are optional and may be omitted unless the add-on is
known not to work with particular FlightGear versions.
/addon/min-FG-version defaults to 2017.4.0 and /addon/max-FG-version to
the special value 'none' (only allowed for /addon/max-FG-version). Apart

from this special case, every non-empty value present in one of these
two fields must be a proper FlightGear version number usable with
simgear::strutils::compare_versions(), for instance '2017.4.1'.

Add-on version
==============

Node: /addon/version

The /addon/version node gives the version of the add-on and must obey a
strict syntax[2], which is a subset of what is described in PEP 440:

  https://www.python.org/dev/peps/pep-0440/

Valid examples are, in increasing sort order:

  1.2.5.dev1   # first development release of 1.2.5
  1.2.5.dev4   # fourth development release of 1.2.5
  1.2.5
  1.2.9
  1.2.10a1.dev2   # second dev release of the first alpha release of 1.2.10
  1.2.10a1   # first alpha release of 1.2.10
  1.2.10b5   # fifth beta release of 1.2.10
  1.2.10rc12   # twelfth release candidate for 1.2.10
  1.2.10
  1.3.0
  2017.4.12a2
  2017.4.12b1
  2017.4.12rc1
  2017.4.12

.devN suffixes can of course be used on beta and release candidates too,
just as with the 1.2.10a1.dev2 example given above for an alpha release.
Note that a development release always sorts before the corresponding
non-development release (e.g., 2017.2.1b5.dev4 comes before 2017.2.1b5).

Other fields
============

The other nodes of 'addon-metadata.xml' should be self-explanatory. :-)

```
3. Add-ons and the Property Tree
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
a) Add-on metadata
   ^^^^^^^^^^^^^^^^
```

The most important metadata for each registered add-on is made
accessible in the Property Tree under /addons/by-id/ADDON_ID and the
property /addons/by-id/ADDON_ID/loaded can be checked or listened to, in
order to determine when a particular add-on is loaded. There is also a
Nasal interface to access add-on metadata in a convenient way (see
below).

More precisely, when an add-on is registered, its name, id, base path,
version (converted to a string), loaded status (boolean) and load
sequence number (int) become available in the Property Tree as
/addons/by-id/ADDON_ID/{name,id,path,version,loaded,load-seq-num}. The
loaded status is initially false, and set to true when the add-on
loading phase is complete.

There are also /addons/addon[i]/path nodes where i is 0 for the first
registered add-on, 1 for the second one, etc.

```
b) Subtree reserved for add-on developers
   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

For each add-on, the subtree of the global Property Tree starting at
/addons/by-id/ADDON_ID/addon-devel is reserved for the specific needs of
the add-on, where ADDON_ID stands for the add-on identifier. For
instance, developers of the add-on whose identifier is
user.joe.FlyingTurtle can store whatever they want under
/addons/by-id/user.joe.FlyingTurtle/addon-devel with the assurance that
doing this won't clash with properties used by the add-on framework.

Example:

  /addons/by-id/user.joe.FlyingTurtle/addon-devel/some/property and

```
/addons/by-id/user.joe.FlyingTurtle/addon-devel/other/property
```

could be two properties used for the specific needs of the
add-on whose identifier is user.joe.FlyingTurtle.

Add-on developers should *not* use other places in the /addons subtree
of the Property Tree for their custom properties.

4. Resources under the add-on directory
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Many functions in FlightGear use files that are located using the
SimGear ResourceManager class. This class allows one to point to files
by relative path in aircraft source files and other places. The resource
manager queries a set of providers, some of which look inside aircraft
paths (starting with the current aircraft), others inside scenery paths,
others under $FG_ROOT, etc. The first file that matches the specified
resource path is used.

One of these providers only handles resource paths with a very specific
syntax, which is:

```
[addon=ADDON_ID]path/relative/to/the/addon/directory
```

  (for instance, [addon=user.joe.FlyingTurtle]images/eject-button.png)

When you use such a syntax in a place that is expected to contain a
resource path, it will only find the specified file under the directory
of the add-on whose identifier is ADDON_ID. This allows one to
specifically target a particular file inside a particular add-on,
instead of crossing fingers and hoping that the specified resource won't
be found by coincidence in another place such as an aircraft directory,
a scenery directory or inside $FG_ROOT (such mistakes can easily happen
when unrelated places use files with rather generic names, such as
button.png, system.xml, etc.).

The [addon=ADDON_ID]relative/path syntax is useful where resources are
specified inside non-Nasal files (e.g., in property-rule configuration
files, which use the XML format). For the particular case of Nasal code,

there is a better way that is explained below (see âNasal APIâ): the
resourcePath() method of addons.Addon objects returns a string like
"[addon=ADDON_ID]relative/path" when you pass it the string
"relative/path". This is a good thing to use, because then your Nasal
code doesn't need to know about the particular syntax for
add-on-specific resources and, more interestingly, doesn't have to
hardcode the add-on identifier every time you need to access a resource
inside the add-on directory.

5. Persistent storage location for add-ons
   ~~~~~~

If an add-on needs to store data that persists across FlightGear
sessions, it can use a specific directory tree whose path is obtained
with addon.storagePath, where 'addon' is an addons.Addon instance. This
corresponds to $FG_HOME/Export/Addons/ADDON_ID, however it is simpler
and better to use addon.storagePath instead of hardcoding and manually
assembling this path in each add-on. Since the directory is likely not
to exist until the add-on creates it, the recommended usage pattern is
the following:

  1) Create the add-on-specific storage directory if it doesn't already
     exist, and optionally get its path at the same time:

        storageDir = addon.createStorageDir();

     Typically, you'll run this in the add-on main() function (at least,
     early enough) if your add-on uses the storage directory. Note that
     there is no need to check yourself whether the directory already
     exists: addon.createStorageDir() does that for you.

  2) At any time, you can get a path to the add-on-specific storage
     directory with:

        storageDir = addon.storagePath

     Accessing addon.storagePath doesn't check for the existence nor the
     type of $FG_HOME/Export/Addons/ADDON_ID, thus it is a bit faster
     than addon.createStorageDir(). Use addon.storagePath whenever you

21

know that the directory has already been created.

The features described in this section were added in FlightGear 2018.2.

6. Add-on-specific menus and dialogs
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

a) Add-on-specific menus
   ~~~~~~~~~~~~~~~~~~~~~~~

Add-ons can easily provide their own menus. If an add-on is loaded that
has a file named 'addon-menubar-items.xml' in its base directory, the
menus described in this file are added to the FlightGear menu bar. The
file should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<PropertyList>
  <meta>
    <file-type type="string">FlightGear add-on menu bar items</file-type>
    <format-version type="int">1</format-version>
  </meta>

  <menubar-items>
    <menu>
      ...
    </menu>

    ...

    <menu>
      ...
    </menu>
  </menubar-items>
</PropertyList>
```

In this file, each <menu> element must be a valid menu description for
the FlightGear menu system (the FlightGear standard menubar in
$FG_ROOT/gui/menubar.xml provides good examples). Here is an example

that adds one menu with an entry for running some Nasal code and another
entry for opening a custom dialog (see below for add-on-specific dialogs):

```
<menu>
  <label>My Menu</label>
  <enabled type="bool">true</enabled>

  <item>
    <label>Run Foobar Nasal</label>
    <binding>
      <command>nasal</command>
      <script>foobar.doBaz();</script>
    </binding>
  </item>

  <item>
    <label>My Foobar Dialog</label>
    <binding>
      <command>dialog-show</command>
      <dialog-name>my-foobar-dialog</dialog-name>
    </binding>
  </item>
</menu>
```

This feature was added in FlightGear 2018.2.

  For older versions, one can add menus via addon-config.xml, but it's a
  bit hackish because of the menu index problem.

b) Add-on-specific dialogs
  ^^^^^^^^^^^^^^^^^^^^^^^^^^

As is the case for aircraft, add-ons can provide their own dialogs by
shipping the corresponding XML files in the subfolder gui/dialogs of the
add-on base directory. In other words, with a file like

  <addon-base-path>/gui/dialogs/my-foobar-dialog.xml

starting with:

```
<?xml version="1.0" encoding="UTF-8"?>
<PropertyList>
  <name>my-foobar-dialog</name>

...
```

the following <item> element inside 'addon-menubar-items.xml' (see
above) describes a valid menu entry for showing the custom dialog.

```
    <item>
      <label>My Foobar Dialog</label>
      <binding>
        <command>dialog-show</command>
        <dialog-name>my-foobar-dialog</dialog-name>
      </binding>
    </item>
```

See $FG_ROOT/gui/dialogs to get inspiration from FlightGear's standard
dialogs.

This feature was added in FlightGear 2018.2.

7. How to run code after an add-on is loaded
   ~~~~~~

You may want to set up Nasal code to be run after an add-on is loaded;
here is how to do that:

```
  var addonId = "user.joe.FlyingTurtle";
  var loadedFlagNode = props.globals.getNode("/addons")
                              .getChild("by-id", 0, 1)
                              .getChild(addonId, 0, 1)
                              .getChild("loaded", 0, 1);

  if (loadedFlagNode.getBoolValue()) {
    logprint(5, addonId ~ " is already loaded");
  } else {
    # Define a function to be called after the add-on is loaded
```

```
    var id = setlistener(
      loadedFlagNode,
      func(changedNode, listenedNode) {
        if (listenedNode.getBoolValue()) {
          removelistener(id);
          logprint(5, addonId ~ " is loaded");
        };
      },
      0, 0);
  }
```

8. Overview of the C++ API
   ~~~~~~~~~~~~~~~~~~~~~~~~~

The add-on C++ infrastructure mainly relies on the following classes:
AddonManager, Addon and AddonVersion. AddonManager is used to register
add-ons, which later leads to their loading. AddonManager relies on an
std::map<std::string, AddonRef>, where keys are add-on identifiers and
AddonRef is SGSharedPtr<Addon> at the time of this writing (changing it
to another kind of smart pointer should be a mere one-line change). This
map holds the metadata of each registered add-on. Accessor methods are
available for:

  - retrieving the lists of registered and loaded add-ons;

  - checking if a particular add-on has already been registered or
    loaded;

  - for each add-on, obtaining an Addon instance which can be queried
    for its identifier, its name, identifier, version, base path, the
    minimum and maximum FlightGear versions it requires, its base node
    in the Property Tree, its order in the load sequence...

The AddonVersion class handles everything about add-on version numbers:
  - initialization from the individual components or from a string;
  - conversion to a string and output to an std::ostream;
  - access to every component;
  - comparisons using the standard operators: ==, !=, <, <=, >, >=.

Registering an add-on using AddonManager::registerAddon() ensures
uniqueness of the add-on identifier and makes its name, identifier, base
path, version (converted to a string), loaded status (boolean) and load
sequence number (int) available in the Property Tree as
/addons/by-id/ADDON_ID/{name,id,path,version,loaded,load-seq-num}.

Note: if C++ code needs to use the add-on base path, better use
      AddonManager::addonBasePath() or Addon::getBasePath(), whose
      return values can't be tampered with by Nasal code.

AddonManager::registerAddon() fails with a specific exception if the
running FlightGear instance doesn't match the min-FG-version and
max-FG-version requirements declared in the addon-metadata.xml file, as
well as in the obvious other cases (required files such as
addon-metadata.xml not found, invalid syntax in such files, etc.). The
code in options.cxx (fgOptAddon()) catches such exceptions and displays
the appropriate error message with SG_LOG() and
fatalMessageBoxThenExit().

9. Nasal API
   ~~~~~~~~~

The Nasal add-on API all lives in the 'addons' namespace. It gives Nasal
code easy access to add-on metadata, for instance like this:

```
  var myAddon = addons.getAddon("user.joe.FlyingTurtle");
  print(myAddon.id);
  print(myAddon.name);
  print(myAddon.version.str());

  foreach (var author; myAddon.authors) {
    print(author.name, " ", author.email, " ", author.url);
  }

  foreach (var maintainer; myAddon.maintainers) {
    print(maintainer.name, " ", maintainer.email, " ", maintainer.url);
  }

  print(myAddon.shortDescription);
```

```
  print(myAddon.longDescription);
  print(myAddon.licenseDesignation);
  print(myAddon.licenseFile);
  print(myAddon.licenseUrl);
  print(myAddon.basePath);
  print(myAddon.minFGVersionRequired);
  print(myAddon.maxFGVersionRequired);
  print(myAddon.homePage);
  print(myAddon.downloadUrl);
  print(myAddon.supportUrl);
  print(myAddon.codeRepositoryUrl);

  foreach (var tag; myAddon.tags) {
    print(tag);
  }

  print(myAddon.loadSequenceNumber);
  # myAddon.node is a props.Node object for /addons/by-id/ADDON_ID
  print(myAddon.node.getPath());
```

Among other things, the Nasal add-on API allows one to get the version
of any registered add-on as a ghost and reliably compare it to another
instance of addons.AddonVersion:

```
  var myAddon = addons.getAddon("user.joe.FlyingTurtle");
  var firstVersionOK = addons.AddonVersion.new("2.12.5rc1");
  # Or alternatively:
  #   var firstVersionOK = addons.AddonVersion.new(2, 12, 5, "rc1");

  if (myAddon.version.lowerThan(firstVersionOK)) {
    ...
```

Here follows the complete Nasal add-on API, at the time of this writing.
All strings are encoded in UTF-8.

Queries to the AddonManager:

```
  addons.isAddonRegistered(string addonId) -> bool (1 or 0)
  addons.registeredAddons()                -> vector<addons.Addon>
                                              (in registration/load order)
```

```
   addons.isAddonLoaded(string addonId)      -> bool (1 or 0)
   addons.loadedAddons()                     -> vector<addons.Addon>
                                                (in lexicographic order)
   addons.getAddon(string addonId)           -> addons.Addon instance (ghost)
```

Read-only data members (attributes) of addons.Addon objects:

```
   id                  the add-on identifier, in reverse DNS style (string)
   name                the add-on âpretty nameâ (string)
   version             the add-on version (instance of addons.AddonVersion,
                       ghost)
   authors             the add-on authors (vector of addons.Author ghosts)
   maintainers         the add-on maintainers (vector of addons.Maintainer
                       ghosts)
   shortDescription    the add-on short description (string)
   longDescription     the add-on long description (string)
   licenseDesignation  licensing terms: "GNU GPL version 2 or later",
                       "CC0 1.0 Universal", etc. (string)
   licenseFile         relative, slash-separated path to a file under
                       the add-on base directory containing the license
                       text (string)
   licenseUrl          stable, official URL for the add-on license text
                       (string)
   basePath            path to the add-on base directory (string)
   storagePath         path to the add-on storage directory (string)
                       This is $FG_HOME/Export/Addons/ADDON_ID.
                       [added in FlightGear 2018.2]
   minFGVersionRequired minimum required FG version for the add-on (string)
   maxFGVersionRequired max. required FG version... or "none" (string)
   homePage            add-on home page (string)
   downloadUrl         add-on download URL (string)
   supportUrl          add-on support URL (string)
   codeRepositoryUrl   URL pointing to the development repository of
                       the add-on (Git, Subversion, etc.; string)
   tags                vector containing the add-on tags used to help
                       users find add-ons (vector of strings)
   node                base node for the add-on in the Property Tree:
                       /addons/by-id/ADDON_ID (props.Node object)
   loadSequenceNumber  0 for the first registered add-on, 1 for the
                       second one, etc. (integer)
```

```
Member functions (methods) of addons.Addon objects:

  createStorageDir() -> string
                        Create the add-on storage directory if it
                        doesn't already exist (that is,
                        $FG_HOME/Export/Addons/ADDON_ID). Return its
                        path as a string.
                        [added in FlightGear 2018.2]
  resourcePath(string relPath) -> string
                        Return a resource path suitable for use with the
                        simgear::ResourceManager. 'relPath' must be
                        relative to the add-on base directory, and
                        mustn't start with a '/'. You can use this
                        method for instance to specify an image file for
                        display in a Canvas widget.

                        In you want a full path to the resource file
                        (e.g., for troubleshooting), call resolvepath()
                        with the return value of addons.Addon.resourcePath().

Read-only data members (attributes) of addons.AddonVersion objects:

  majorNumber           non-negative integer
  minorNumber           non-negative integer
  patchLevel            non-negative integer
  suffix                string such as "", "a1", "b2.dev45", "rc12"...

Member functions (methods) of addons.AddonVersion objects:

  new(string version)                          | construct from string

  new(int major, int minor=0, int patchLevel=0, | construct
      string suffix="")                        | from components

  str()                                        | string representation

  equal(addons.AddonVersion other)             |
  nonEqual(addons.AddonVersion other)          | compare to another
  lowerThan(addons.AddonVersion other)         | addons.AddonVersion
```

```
    lowerThanOrEqual(addons.AddonVersion other)    | instance
    greaterThan(addons.AddonVersion other)         |
    greaterThanOrEqual(addons.AddonVersion other) |
```

Read-only data members (attributes) of addons.Author objects:

```
  name                    author name (non-empty string)
  email                   email address of the author (string)
  url                     home page of the author (string)
```

Read-only data members (attributes) of addons.Maintainer objects:

```
  name                    maintainer name (non-empty string)
  email                   email address of the maintainer (string)
  url                     home page of the maintainer, if a person; if the
                          maintainer is a mailing-list, the URL can point
                          to a web page from which people can subscribe to
                          that mailing-list (string)
```

Footnotes
----

[1] \n represents end-of-line in string literals of languages such as C,
    C++, Python and many others. We use this convention here to
    represent the end-of-line character sequence in the XML data.

[2] MAJOR.MINOR.PATCHLEVEL[{a|b|rc}N1][.devN2] where MAJOR, MINOR and
    PATCHLEVEL are non-negative integers, and N1 and N2 are positive
    integers.

4   Airspeed-indicator

The airspeed indicator can be initialized in an instrumentation.xml file.
If not specified, the generic indicator will be loaded
from the Aircraft/Generic/generic-instrumentation.xml.

The normal setup is :

```
  <airspeed-indicator>
```

```
    <name>airspeed-indicator</name>
    <number>0</number>
    <total-pressure>/systems/pitot/total-pressure-inhg</total-pressure>
    <static-pressure>/systems/static/pressure-inhg</static-pressure>
    <has-overspeed-indicator>1</has-overspeed-indicator>
  </airspeed-indicator>
```

Of course the total and static pressure may be sourced from any other
pitot and static system when defined:

```
  <airspeed-indicator>
    <name>airspeed-indicator</name>
    <number>1</number>
    <total-pressure>/systems/pitot[1]/total-pressure-inhg</total-pressure>
    <static-pressure>/systems/static[1]/pressure-inhg</static-pressure>
    <has-overspeed-indicator>0</has-overspeed-indicator>
  </airspeed-indicator>
```

Note that the Aircraft/Generic/generic-systems.xml only initiates one
pitot and one static system, see also README.systems

```
<total-pressure> is optional --- defaults to "/systems/pitot/total-pressure-inhg"
```
For supersonic aircraft with an airspeed indicator NOT compensating for
a shockwave in front of the pitot tube (most probably the case), use:
```
<total-pressure>/systems/pitot/measured-total-pressure-inhg</total-pressure>
```

```
<static-pressure> is optional --- defaults to "/systems/static/pressure-inhg"
<has-overspeed-indicator> is optional --- defaults to 0 / off
```

The <has-overspeed-indicator> provides a property for "barber-pole" animation,
 and is set to 0 / false by default ,

If enabled , these properties should be added in the aircraft -set file,
with that aircraft's correct figures.

```
    <airspeed-indicator>
        <ias-limit>248.0</ias-limit>
        <mach-limit>0.48</mach-limit>
        <alt-threshold>13200.0</alt-threshold>
    </airspeed-indicator>
```

The default values are for a Beechcraft B1900D .

<ias-limit> is the aircraft's VNE (never exceed speed) in KIAS

<mach-limit> Mach speed limit.

<alt-threshold> altitude at which these figures were calculated.

Note : <mach-limit> is the mach limit at <alt-threshold>
This was designed for indicated airspeed limits, but could probably be extended
for mach limits.

To initiate additional airspeed indicators, add in your instrumentation
file (for airspeed indicator index 1):

```
  <airspeed-indicator>
    <name>airspeed-indicator</name>
    <number>1</number>
    <total-pressure>/systems/pitot[1]/total-pressure-inhg</total-pressure>
    <static-pressure>/systems/static[1]/pressure-inhg</static-pressure>
    <has-overspeed-indicator>0</has-overspeed-indicator>
  </airspeed-indicator>
```

Note: this airspeed indicator sources its pressures from the second
pitot and static system (with index 1).
and in the aircraft -set file:

```
    <airspeed-indicator n="1">
          <serviceable type="bool" archive="y">true</serviceable>
    </airspeed-indicator>
```

And if "has-overspeed-indicator" = 1, the appropriate limits as explained
above in the airspeed-indicator brackets.

5   Checklists

CHECKLISTS

You can create one or more checklist for an aircraft under /sim/checklists. These
are intended to mimic the checklists of aircraft themselves, and can be found under
the Help->Checklists menu within the simulator.

Tutorials are automatically generated from checklists on startup.

Each checklist is defined as a property tree under /sim/checklists/checklist[n]
or /sim/checklists/group[n]/checklist[m]

with the following tags

```
<title>        - Name of the checklist
<page>         - Zero or more pages for the checklist containing:
 <item>        - One or more checklist items containing:
  <name>       - name of the checklist item (e.g. Carb Heat), to appear on the left
  <value>      - One or more values for the checklist item, to appear on the right
                 hand side
  <marker>     - A tutorial marker (displayed when the user presses the ? button)
                 This can be easily placed using the Help->Display Tutorial Marker.
                 Contains x-m, y-m, z-m and scale tag.
  <condition> - Optional standard FlightGear condition node that evaluates when the
                 checklist item has been completed.
  <binding>   - Zero or more bindings to execute the checklist item.  Allows the user
                 to have their virtual co-pilot perform the action if they select the
                 ">" button next to the checklist item.
```

The <page> tag may be omitted for single-page checklists, with the <item> tags
immediately under the <checklist[n]> node.

Checklists may be grouped under <group> nodes with a <name> tag decribing the
group. For example

```
<group>
 <name>Emergency</name>
 <checklist>...
 <checklist>...
```

```
</group>
<group>
 <name>Normal</name>
 <checklist>...
 <checklist>...
</group>
```

See the c172p for an example of this in action (Aircraft/c172p/c172-checklists.xml).

6   Commands

FlightGear Commands Mini-HOWTO

David Megginson
Started: 2002-10-25
Last revised: 2007-12-01

In FlightGear, a *command* represents an action, while a *property*
represents a state. The trigger for a command can be any kind of user
input, including the keyboard, mouse, joystick, GUI, instrument panel,
or a remote network client.

XML Command Binding Markup

Most of the command-binding in FlightGear is handled through static
XML configuration files such as $FG_ROOT/keyboard.xml for the
keyboard, $FG_ROOT/mice.xml for the mouse, and
$FG_ROOT/gui/menubar.xml for the menubar. In all of these files, you
reference a command through a binding. This binding advances the
first throttle by 1%, up to a maximum value of 1.0:

```
  <binding>
   <command>property-adjust</command>
   <property>/controls/throttle[0]</property>
   <step type="double">0.01</step>
   <max>1.0</max>
```

```
    </binding>
```

A command binding always consists of the XML 'binding' element, with
one subelement named 'command' containing the command name (such as
'property-adjust'). All other subelements are named parameters to the
command: in this case, the parameters are 'property', 'step', and
'max'. Here is a simpler binding, with no parameters:

```
  <binding>
   <command>exit</command>
  </binding>
```

Bindings always appear inside some other kind of markup, depending on
the input type. For example, here is the binding from keyboard.xml
that links the ESC key to the 'exit' command:

```
 <key n="27">
  <name>ESC</name>
  <desc>Prompt and quit FlightGear.</desc>
  <binding>
    <command>exit</command>
  </binding>
 </key>
```

Usually, more than one binding is allowed for a single input trigger,
and bindings are executed in order from first to last. Bindings support
conditions (see README.conditions):

```
 <key n="113">
  <name>q</name>
  <desc>Test</desc>

  <binding>
   <condition>
    <property>/devices/status/mice/mouse/button[0]</property>
   </condition>
   <command>nasal</command>
   <script>print("mouse button 0 pressed")</script>
  </binding>
 </key>
```

Keyboard definitions can embed bindings in tags <mod-up> (key released), <mod-shift>, <mod-ctrl>, <mod-alt>, <mod-meta>, <mod-super>, and <mod-hyper>. Nesting is supported. Meta, Super, and Hyper modifier tags are for local use only, and must be supported by the operating system to work.

```
<key n="113">
 <name>q</name>
 <desc>Test</desc>
 <binding>
  <command>nasal</command>
  <script>print("q pressed")</script>
 </binding>

 <mod-alt>
  <binding>
   <command>nasal</command>
   <script>print("Alt-q pressed")</script>
  </binding>

  <mod-super>
   <binding>
    <command>nasal</command>
    <script>print("Alt-Super-q pressed")</script>
   </binding>

   <mod-meta>
    <binding>
     <command>nasal</command>
     <script>print("Alt-Super-Meta-q pressed")</script>
    </binding>

   </mod-meta>
  </mod-super>
 </mod-alt>
</key>
```

Built-in Commands

As of the last revision date, the following commands were available
from inside FlightGear; the most commonly-used ones are the commands
that operate on property values (FlightGear's internal state):

null - do nothing

script - execute a PSL script
  script: the PSL script to execute

exit - prompt and quit FlightGear

pause - pause/resume the simulation

load - load properties from an XML file
  file: the name of the file to load, relative to the current
    directory (defaults to "fgfs.sav")

save - save properties to an XML file
  file: the name of the file to save, relative to the current
    directory (defaults to "fgfs.sav").

loadxml - load XML file into property tree
  filename: the path & filename of the file to load
  targetnode: the target node within the property tree where to store the XML
  file's structure. If targetnode isn't defined, then the data will be stored
  in a node "data" under the argument branch.

savexml - save property tree node to XML file
  filename: the path & filename for the file to be saved
  sourcenode: the source node within the property tree where the XML file's
  structure is assembled from. If sourcenode isn't defined, then savexml will
  try to save data stored in a node "data" in the argument branch.

panel-load - (re)load the 2D instrument panel
  path: the path of the XML panel file, relative to $FG_ROOT (defaults
    to the value of /sim/panel/path if specified, or
    "Panels/Default/default.xml" as a last resort.

```
panel-mouse-click - pass a mouse click to the instrument panel
  button: the number of the mouse button (0-based)
  is-down: true if the button is down, false if it is up
  x-pos: the x position of the mouse click
  y-pos: the y position of the mouse click

preferences-load - (re)load preferences
  path: the file name to load preferences from, relative to $FG_ROOT.
    Defaults to "preferences.xml".

view-cycle - cycle to the next viewpoint

screen-capture - capture the screen to a file

tile-cache-reload - reload the scenery tile cache

lighting-update - update FlightGear's lighting

property-toggle - swap a property value between true and false
  property: the name of the property to toggle

property-assign - assign a value to a property
  property[0]: the name of the property that will get the new value.
  value: the new value for the property; or
  property[1]: the name of the property holding the new value.

property-interpolate - assign a value to a property, interpolated
                       over time
  property[0]: the name of the property that will get the new value
               and defines the starting point of the interpolation
  value:       the new value for the property; or
  property[1]: the name of the property holding the new value.
  time:        the time in seconds it takes for the transition from the
               old value to the new value of property[0]; or
  rate:        the ammount of change per second the value of property[0] changes
               to transition to the new value

property-adjust - adjust the value of a property
  property: the name of the property to increment or decrement
```

```
  step: the amount of the increment or decrement (defaults to 0)
  offset: input offset distance (used for the mouse; multiplied by
    factor)
  factor: factor for multiplying offset distance (used for the mouse;
    defaults to 1)
  min: the minimum allowed value (default: no minimum)
  max: the maximum allowed value (default: no maximum)
  mask: 'integer' to apply only to the left of the decimal point;
    'decimal' to apply only to the right of the decimal point; 'all'
    to apply to the full value (defaults to 'all')
  wrap: true if the value should be wrapped when it passes min or max;
    both min and max must be specified (defaults to false)

property-multiply - multiply the value of a property
  property: the name of the property to multiply
  factor: the amount by which to multiply (defaults to 1.0)
  min: the minimum allowed value (default: no minimum)
  max: the maximum allowed value (default: no maximum)
  mask: 'integer' to apply only to the left of the decimal point;
    'decimal' to apply only to the right of the decimal point; 'all'
    to apply to the full value (defaults to 'all')
  wrap: true if the value should be wrapped when it passes min or max;
    both min and max must be specified (defaults to false)

property-swap - swap the values of two properties
  property[0]: the name of the first property
  property[1]: the name of the second property

property-scale - set the value of a property based on an axis
  property: the name of the property to set
  setting: the current input setting (usually a joystick axis from -1
    or 0 to 1)
  offset: the offset to shift by, before applying the factor (defaults
    to 0)
  factor: the factor to multiply by (use negative to reverse; defaults
    to 1.0)
  squared: if true will square the resulting value (same as power=2)
  power: the resulting value will be taken to the power of this integer
    value (overrides squared; default=1)
```

```
property-cycle - cycle a property through a set of values
  property: the name of the property to cycle
  value[*]: all of the allowed values


dialog-new - create new dialog from the argument branch


dialog-show - show an XML-configured dialog box
  dialog-name - the name of the dialog to show


dialog-close - close the active dialog box


dialog-update - copy values from FlightGear to the active dialog box
  object-name: the name of the GUI object to update (defaults to all
    objects)


dialog-apply - copy values from the active dialog box to FlightGear
  object-name: the name of the GUI object to apply (defaults to all
    objects)


presets-commit - commit preset values from /sim/presets


open-browser - open the web browser and show given file
   path: name of the local file to be opened.
   url: URL to be opened (http://..., ftp://...).


The following commands are temporary, and will soon disappear or be
renamed; do NOT rely on them:


old-save-dialog - offer to save a flight


old-load-dialog - offer to load a flight


old-reinit-dialog - offer to reinit FlightGear


old-hires-snapshot-dialog - save a hires screen shot


old-snapshot-dialog - save a screenshot


old-print-dialog - print the screen (Windows only)
```

old-pilot-offset-dialog - set pilot offsets graphically

old-hud-alpha-dialog - set the alpha value for the HUD

old-properties-dialog - display the property browser

old-preset-airport-dialog - set the default airport

old-preset-runway-dialog - set the default runway

old-preset-offset-distance-dialog - set the default offset distance

old-preset-altitude-dialog - set the default altitude

old-preset-glidescope-dialog - set the default glidescope

old-preset-airspeed-dialog - set the default airspeed

old-preset-commit-dialog - commit preset values

old-ap-add-waypoint-dialog - add a waypoint to the current route

old-ap-pop-waypoint-dialog - remove a waypoint from the current route

old-ap-clear-dialog - clear the current route

old-ap-adjust-dialog - adjust the autopilot settings

old-lat-lon-format-dialog - toggle the lat/lon format in the HUD

Adding New Commands in C++

To add a new command to FlightGear, you first need to create a
function that takes a single SGPropertyNode const pointer as an
argument:

  void

```
  do_something (SGPropertyNode * arg)
  {
    something();
  }
```

Next, you need to register it with the command manager:

```
  globals->get_commands()->addCommand("something", do_something);
```

Now, the command "something" is available to any mouse, joystick,
panel, or keyboard bindings. If the bindings pass any arguments, they
will be children of the SGPropertyNode passed in:

```
  void
  do_something (const SGPropertyNode * arg)
  {
    something(arg->getStringValue("foo"), arg->getDoubleValue("bar"));
  }
```

That's pretty-much it. Apologies in advance for not making things any
more complicated.

7   Conditions

CONDITIONS IN FLIGHTGEAR PROPERTY FILES

Written by David Megginson, david@megginson.com
Last modified: $Date$

This document is in the Public Domain and comes with NO WARRANTY!

1. Introduction

Some FlightGear property files contain conditions, affecting whether
bindings or animations are applied. For example, the following

binding will apply only when the /sim/input/selected/engine[0]
property is true:

```
  <binding>
   <condition>
    <property>/sim/input/selected/engine[0]</property>
   </condition>
   <command>property-assign</command>
   <property>/controls/starter[0]</property>
   <value type="bool">true</value>
  </binding>
```

Conditions always occur within a property subtree named "condition",
which is equivalent to an "and" condition.

2. Comparison Operators

The simplest condition is "property". It resolves as true when the
specified property has a boolean value of true (i.e. non-zero, etc.)
and false otherwise. Here is an example:

```
  <condition>
   <property>/sim/input/selected/engine[0]</property>
  </condition>
```

For more sophisticated tests, you can use the "less-than",
"less-than-equals", "greater-than", "greater-than-equals", "equals",
and "not-equals" comparison operators. These all take two operands,
either two "property" operands or one "property" and one "value"
operand, and return true or false depending on the result of the
comparison. The value of the second operand is always forced to the
type of the first; for example, if you compare a string and a double,
the double will be forced to a string and lexically compared. If one
of the operands is a property, it is always assumed to be first. Here
is an example of a comparison that is true only if the RPM of the
engine is less than 1500:

```
  <condition>
```

```
    <less-than>
     <property>/engines/engine[0]/rpm</property>
     <value>1500</value>
    </less-than>
   </condition>
```

3. Boolean Operators

Finally, there are the regular boolean operators "and", "or", and
"not". Each one surrounds a group of other conditions, and these can
be nested to arbitrary depths. Here is an example:

```
  <condition>
   <and>
    <or>
     <less-than>
      <property>/engines/engine[0]/rpm</property>
      <value>1500</value>
     </less-than>
     <greater-than>
      <property>/engines/engine[0]/rpm</property>
      <value>2500</value>
     </greater-than>
    </or>
    <property>/engines/engine[0]/running</property>
   </and>
  </condition>
```

The top-level "condition" is an implicit "and".

4. Approximating if...else

There is no equivalent to the regular programming 'else' statement in
FlightGear conditions; instead, each condition separately must take
the others into account. For example, the equivalent of

```
    if (x == 3) ... else if (y == 5) ... else ...
```

in FlightGear conditions is

```
  <condition>
   <equals>
    <property>/x</property>
    <value>3</value>
   </equals>
   <not>
    <equals>
     <property>/y</property>
     <value>5</value>
    </equals>
   </not>
  </condition>
```

and then

```
  <condition>
   <equals>
    <property>/y</property>
    <value>5</value>
   </equals>
   <not>
    <equals>
     <property>/x</property>
     <value>3</value>
    </equals>
   </not>
  </condition>
```

and then

```
  <condition>
   <not>
    <equals>
     <property>/x</property>
     <value>3</value>
    </equals>
```

```
   </not>
   <not>
    <equals>
     <property>/y</property>
     <value>5</value>
    </equals>
   </not>
  </condition>
```

It's verbose, but it works nicely within existing property-based
formats and provides a lot of flexiblity.

5. Syntax Summary

Here's a quick syntax summary:

* <and>...</and>

  Contains one or more subconditions, all of which must be true.

* <condition>...</condition>

  The top-level container for conditions, equivalent to an "and" group

* <equals>...</equals>

  Contains two properties or a property and value, and is true if the
  properties have equivalent values.

* <not-equals>...</not-equals>

  Contains two properties or a property and value, and is true if the
  properties have different values.

* <greater-than>...</greater-than>

  Contains two properties or a property and a value, and is true if
  the second property or the value has a value greater than the first

property.

* <greater-than-equals>...</greater-than-equals>

  Contains two properties or a property and a value, and is true if
  the second property or the value has a value greater than or equal
  to the first property.

* <less-than>...</less-than>

  Contains two properties or a property and a value, and is true if
  the second property or the value has a value less than the first
  property.

* <less-than-equals>...</less-than-equals>

  Contains two properties or a property and a value, and is true if
  the second property or the value has a value less than or equal
  to the first property.

* <not>...</not>

  Contains one subcondition, which must not be true.

* <not-equals>...</not-equals>

  Contains two properties or a property and value, and is true if the
  properties do not have equivalent values.

* <or>...</or>

  Contains one or more subconditions, at least one of which must be
  true.

* <property>...</property>

  The name of a property to test.

* <value>...</value>

A literal value in a comparison.

8   Digitalfilters

```
COMMON SETTINGS
===============================================================================
```

Currently four types of digital filter implementations are supported. They all
serve an individual purpose or are individual implementations of a specific
filter type. Each filter implementation uses the same set of basic configuration
tags and individual configuration elements. These individual elements are
described in the section of the filter.

```
The InputValue
===============================================================================
```
Each filter has several driving values, like the input value itself, sometimes
a reference value, a gain value and others. Most of these input values can be
either a constant value or the value of a property. They all use the same syntax
and will be referred to as InputValue in the remaining document.

The complete XML syntax for a InputValue is

```
<some-element>
  <condition>
    <!-- any condition as defined in README.conditions -->
  </condition>
  <property>/some/property/name</property>
  <value>0.0</value>
  <scale>1.0</value>
  <offset>0.0</offset>
  <max>infinity</max>
  <min>-infinity<min>
  <abs>false</abs>
  <period>
    <min>-180.0</min>
    <max>-180.0</max>
  </period>
</some-element>
```

The enclosing element <some-element> is the element defined in each filter, like
<input>, <u_min>, <reference> etc. These elements will be described later.
The value of the input is calculated based on the given value, scale and offset as
value * scale + offset
and the result is clipped to min/max, if given.
With the full set of given elements, the InputValue will initialize the named
property to the value given, reduced by the given offset and reverse scaled by
the given scale.

Example:
```
<input>
  <property>/controls/flight/rudder</property>
  <value>0.0</value>
  <scale>0.5</scale>
  <offset>0.5</offset>
</input>
```

Will use the property /controls/flight/rudder as the input of the filter. The
property will be initialized at a value of zero and since the property usually is
in the range [-1..+1], the the value of <input> will be in the range
(-1)*0.5+0.5 to (+1)*0.5+0.5 which is [0..1].

The default values for elements not given are:
```
<value/> : 0.0
<scale/> : 1.0
<offset/>: 0.0
<property/> : none
<min/>    : unclipped
<max/>    : unclipped
<abs/>    : false
```

Some examples:
```
<input>
  <property>/position/altitude-ft</property>
  <scale>0.3048</scale>
</input>
```
Gives the altitude in meters. No initialization of the property is performed, no
offset applied.

```
  <value>0.0</value>
</reference>
A constant reference of zero.

A abbreviated method of defining values exist for using a just constant or a
property. The above example may be written as
<reference>0.0</reference>
Or if the reference is defined in a property
<reference>/some/property/name</reference>
No initialization, scaling or offsetting is performed here.
The logic behind this is: If the text node in the element (the text between the
opening and closing tag) can be converted to a double value, it will be interpreted
as a double value. Otherwise the text will
be interpreted as a property name.
Examples:
<reference>3.1415927</reference>              - The constant of PI (roughly)
<reference>/position/altitude-ft</reference> - The property /position/altitude-ft
<reference>3kings</reference>                  - The constant 3. The word kings is
                                                 ignored
<reference>food4less</reference>               - The property food4less

The <property> element may also be written as <prop> for backward compatibility.

There may be one or more InputValues for the same input of a filter which may be
bound to conditions. Each InputValue will have its condition checked in the order
of InputValues given in the configuration file. The first InputValue that returns
true for its condition will be evaluated. Chaining a number of InputValues with
conditions and an unconditioned InputValue works like the C language equivalent
if( condition ) {
  // compute value of first element
} else if( condition2 ) {
  // compute value of second element
} else if( condition3 ) {
  // compute value of third element
} else {
  // compute value of last element
}

Example: Set the gain to 3.0 if /autopilot/locks/heading equals dg-heading-hold or
2.0 otherwise.
```

```
<digital-filter>
  <gain>
    <condition>
      <equals>
        <property>/autopilot/locks/heading</property>
        <value>dg-heading-hold</value>
      </equals>
    </condition>
    <value>3.0</value>
  <gain>
  <!-- Hint: omit a condition here as a fallthru else condition -->
  </gain>
    <value>2.0</value>
  <gain>
<digital-filter>
```

If the element <abs> is used and set to the value "true", only the absolute
value of the input (the positive part) is used for further computations. The
abs function is applied after all other computations are completed.

OutputValue
==
Each filter drives one to many output properties. No scaling or offsetting is
implemented for the output value, these should be done in the filter itself.
The output properties are defined in the <output/> element by adding <property/>
elements within the <output/> element. For just a single output property, the
<property/> element may be ommited. For backward compatibility, <property/> may
be replaced by <prop/>. Non-existing properties will be created with type double.

Example: (Multiple output properties)
```
<output>
  <property>/some/output/property</property>
  <property>/some/other/output/property</property>
  <property>/and/another/output/property</property>
</output>
```

Example: a single output property
```
<output>/just/a/single/property</output>
```

Other Common Settings

```
================================================================================
<name>         String      The name of the filter. Used for debug purpose.
Example:
<name>pressure rate filter</name>


<debug>        Boolean     If true, this filter puts out debug information when
                           updated. Example: <debug>false</debug>


<input>        InputValue  The input property driving the filter.
                           Refer to InputValue for details.


<reference>    InputValue  The reference property for filter that need one.
                           Refer to InputValue for details.


<output>       Complex     Each filter can drive one to many output properties.
                           Refer to OutputValue for details.


<u_min>        InputValue  This defines the optional minimum and maximum value the
<u_max>                    output is clamped to. If neither <u_min> nor <u_max>
                           exists, the output is only limited by the internal limit
                           of double precision float computation.  If either <u_min>
                           or <u_max> is given, clamping is activated. A missing min
                           or max value defaults to 0 (zero).
                           Note: <u_min> and <u_max> may also occour within a <config>
                           element. <min> and <max> may be used as a substitude for
                           the corresponding u_xxx element.
<period>       Complex     Define a periodical input or output value. The phase width
                           is defined by the child elements <min> and <max> which are
                           of type InputValue

Example: Limit the pilot's body temperature to a constant minimum of 36 and a
maximum defined in /pilots/max-body-temperature-degc, initialized to 40.0
<u_max>
  <prop>/pilots/max-body-temperature-degc</prop>
  <value>40.0</
</u_max>
<min>
  <value>36.0</value>
</min
```

Implicit definition of the minimum value of 0 (zero) and defining a maximum of 100.0
```
<config>
  <u_max>100.0</u_max>
</config>
```

This defines the input or output as a periodic value with a phase width of 360, like
the compass rose. Any value reaching the filter's input or leaving the filter at the
output will be transformed to fit into the given range by adding or substracting one
phase width of 360. Values of -270, 90 or 450 applied to this periodical element will
always result in +90. A value of 630, 270 or -90 will be normalized to -90 in the
given example.
```
<period>
  <min>-180.0</min>
  <max>180.0</max>
</period>
```


<enable>   Complex   Define a condition to enable or disable the filter. For
                           disabled filters, no output computations are performed.
                           Only enabled filters fill the output properties. The
                           default for undefined conditions is enabled.
                           Several way exist to define a condition. The most simple
                           case is checking a boolean property. For this, just a
                           <prop> element naming this boolean property is needed.
                           The boolean value of the named property defines the
                           enabled state of the filter. To compare the value of a
                           property with a constant, a <prop> and a <value> element
                           define the property name and the value to be compared.
                           The filter is enabled, if the value of the property equals
                           the given value. A case sensitive string compare is
                           performed here.
                           To define more complex conditions, a <condition> element
                           may be used to define any condition described in
                           README.conditions. If a <condition> element is present
                           and if it contains a valid condition, this conditions has
                           precedence over a given <prop>/<value> condition.
                           The child element <honor-passive>, a boolean flag, may be
                           present within the <enable> element. If this element is
                           true, the property /autopilot/locks/passive-mode is checked
                           and if it is true, the filter output is computed, but the

```
                              output properties are not set.  The default for
                              honor-passive is false
Example: Check a boolean property, only compute this filter if gear-down is true and
          /autopilot/locks/passive-mode is false
<enable>
  <prop>/gear/gear-down</prop>
  <honor-passive>true</honor-passive>
</enable>


Check a property for equality, only compute this filter if the autopilot is locked
in heading mode.
<enable>
  <prop>/autopilot/locks/heading</prop>
  <value>dg-heading-hold</value>
</enable>


Use a complex condition, only compute this filter if the autopilot is serviceable
and the lock is either dg-heading-hold or nav1-heading-hold
<enable>
  <condition>
    <property>/autopilo/serviceable</property>
    <or>
      <equals>
        <property>/autopilot/locks/heading</property>
        <value>dg-heading-hold</value>
      </equals>
      <equals>
        <property>/autopilot/locks/heading</property>
        <value>nav1-heading-hold</value>
      </equals>
    </or>
  </condition>
</enable>


INDIVIDUAL FILTER CONFIGURATION
================================================================================


Digital Filter


Six different types of digital filter can be configured inside the autopilot
```

configuration file. There are four low-pass filter types and two gain filter
types.

The low-pass filter types are:

* Exponential
* Double exponential
* Moving average
* Noise spike filter

The gain filter types are:

* gain
* reciprocal

To add a digital filter, place a <filter> element under the root element. Next to
the global configuration elements described above, the following elements configure
the digital filter:

<filter-time> InputValue This tag is only applicable for the exponential and
                          double-exponential filter types. It controls the
                          bandwidth of the filter. The bandwidth in Hz of the
                          filter is: 1/filter-time. So a low-pass filter with a
                          bandwidth of 10Hz would have a filter time of 1/10 = 0.1

<samples>   InputValue This tag only makes sense for the moving-average filter.
                          It says how many past samples to average.

<max-rate-of-change>
              InputValue This tag is applicable for the noise-spike filter.
                          It says how much the value is allowed to change per
                          second.

<gain>   InputValue This is only applicable to the gain and reciprocal filter
                          types. The output for gain filter is computed as input*gain
                          while the reciprocal filter computes output as gain/input
                          for input values != 0 (zero). Gain may be a constant, a
                          property name defined by a <prop> element within the <gain>
                          element or a property name initialized to a value by using
                          a <prop> and <value> element.

Example: a pressure-rate-filter implemented as a double exponential low pass filter
         with a bandwith of 10Hz

```
<filter>
  <name>pressure-rate-filter</name>
  <debug>false</debug>
  <type>double-exponential</type>
  <enable>
    <prop>/autopilot/locks/pressure-rate-filter</prop>
    <value>true</value>
  </enable>
  <input>/autopilot/internal/pressure-rate</input>
  <output>/autopilot/internal/filtered-pressure-rate</output>
  <filter-time>0.1</filter-time>
</filter>
```

This will filter the pressure-rate property. The output will be to a new
property called filtered-pressure-rate. You can select any numerical property
from the property tree. The input property will not be affected by the filter,
it will stay the same as it would if no filter was configured.

Example 2:

```
<filter>
  <name>airspeed elevator-trim gain reciprocal filter</name>
  <debug>false</debug>
  <enable>
    <prop>/autopilot/locks/airspeed-elevator-trim-gain</prop>
    <value>true</value>
  </enable>
  <type>reciprocal</type>
  <gain>
    <prop>/autopilot/settings/elevator-trim-airspeed-reciprocal-gain</prop>
    <value>7</value>
  </gain>
  <input>/velocities/airspeed-kt</input>
  <output>/autopilot/internal/elevator-trim-gain</output>
  <u_min>0.005</u_min>
  <u_max>0.02</u_max>
</filter>
```

This will use the /velocities/airspeed-kt property to produce a gain factor
that reduces as airspeed increases. At airspeeds up to 350kt the gain will
be clamped to 0.02, at 700kt the gain will be 0.01 and at 1400kt the gain will
be 0.005. The gain will be clamped to 0.005 for airspeeds > 1400kt.

The output from this filter could then be used to control the gain in a PID
controller:

```
<pid-controller>
  <name>Pitch hold</name>
  <debug>false</debug>
  <enable>
    <prop>/autopilot/locks/pitch</prop>
    <value>true</value>
  </enable>
  <input>
    <prop>/orientation/pitch-deg</prop>
  </input>
  <reference>
    <prop>/autopilot/settings/target-pitch-deg</prop>
  </reference>
  <output>
    <prop>/autopilot/internal/target-elevator-trim-norm</prop>
  </output>
  <config>
    <Ts>0.05</Ts>
    <Kp>
      <prop>/autopilot/internal/elevator-trim-gain</prop>
      <value>0.02</value>
    </Kp>
    <beta>1.0</beta>
    <alpha>0.1</alpha>
    <gamma>0.0</gamma>
    <Ti>2.0</Ti>
    <Td>0.2</Td>
    <u_min>-1.0</u_min>
    <u_max>1.0</u_max>
  </config>
</pid-controller>
```

IMPORTANT NOTE: The <Kp> tag in PID controllers has been revised to operate in
the same way as the <gain> elements in filters. However, the original format
of <Kp> will continue to function as before i.e. <Kp>0.02</Kp> will specify a
fixed and unalterable gain factor, but a warning message will be output.

The gain type filter is similar to the reciprocal filter except that the gain
is applied as a simple factor to the input.

Parameters

<name> The name of the filter. Give it a sensible name!

<debug> If this tag is set to true debugging info will be printed on the
console.

<enable> Encloses the <prop> and <value> tags which are used to enable or
disable the filter. Instead of the <prop> and <value> tags, a <condition>
tag may be used to define a condition. Check README.conditions for more
details about conditions. Defaults to enabled if unspecified.

<type> The type of filter. This can be exponential, double-exponential,
moving-average, noise-spike, gain or reciprocal.

<input> The input property to be filtered. This should of course be a
numerical property, filtering a text string or a boolean value does not make
sense.

<output> The filtered value. You can make up any new property.

<u_min> The minimum output value from the filter. Defaults to -infinity.

<u_max> The maximum output value from the filter. Defaults to +infinity.

These are the tags that are applicable to all filter types. The following tags
are filter specific.

<filter-time> This tag is only applicable for the exponential and
double-exponential filter types. It controls the bandwidth of the filter. The
bandwidth in Hz of the filter is: 1/filter-time. So a low-pass filter with a

bandwidth of 10Hz would have a filter time of 1/10 = 0.1

<samples> This tag only makes sense for the moving-average filter. It says how many past samples to average.

<max-rate-of-change> This tag is applicable for the noise-spike filter. Is says how much the value is allowed to change per second.

<gain> This, and it's enclosed <prop> and <value> tags, are only applicable to the gain and reciprocal filter types. The <prop> tag specifies a property node to hold the gain value and the <value> tag specifies an initial default value. The gain defaults to 1.0 if unspecified.

The output from the gain filter type is: input * gain.
The output from the reciprocal filter type is: gain / input.

The gain can be changed during run-time by updating the value in the property node.

9   Effects

```
Effects
-------
```

Effects describe the graphical appearance of 3d objects and scenery in FlightGear. The main motivation for effects is to support OpenGL shaders and to provide different implementations for graphics hardware of varying capabilities. Effects are similar to DirectX effects files and Ogre3D material scripts.

An effect is a property list. The property list syntax is extended with new "vec3d" and "vec4d" types to support common computer graphics values. Effects are read from files with a ".eff" extension or can be created on-the-fly by FlightGear at runtime. An effect consists of a "parameters" section followed by "technique" descriptions. The "parameters" section is a tree of values that describe, abstractly, the graphical characteristics of objects that use the effect. Techniques refer to these parameters and use them to set OpenGL state or to set parameters for shader programs. The names of properties in the

parameter section can be whatever the effects author chooses, although
some standard parameters are set by FlightGear itself. On the other
hand, the properties in the techniques section are all defined by the
FlightGear.

Techniques

A technique can contain a predicate that describes the OpenGL
functionality required to support the technique. The first
technique with a valid predicate in the list of techniques is used
to set up the graphics state of the effect. A technique with no
predicate is always assumed to be valid. The predicate is written in a
little expression language that supports the following primitives:

and, or, equal, less, less-equal
glversion - returns the version number of OpenGL
extension-supported - returns true if an OpenGL extension is supported
property - returns the boolean value of a property
float-property - returns the float value of a property, useful inside equal, less
                 or less-equal nodes
shader-language - returns the version of GLSL supported, or 0 if there is none.

The proper way to test whether to enable a shader-based technique is:
```
  <predicate>
    <and>
    <property>/sim/rendering/shader-effects</property>
    <less-equal>
      <value type="float">1.0</value>
      <shader-language/>
    </less-equal>
    </and>
  </predicate>
```

There is also a property set by the user to indicate what is the level
of quality desired. This level of quality can be checked in the predicate
like this :
```
    <predicate>
      <and>
        <property>/sim/rendering/shader-effects</property>
```

```
  <less-equal>
    <value type="float">2.0</value>
    <float-property>/sim/rendering/quality-level</float-property>
  </less-equal>
  <!-- other predicate conditions -->
      </and>
    </predicate>
```

The range of /sim/rendering/quality-level is [0..5]
 * 2.0 is the threshold for relief mapping effects,
 * 4.0 is the threshold for geometry shader usage.

A technique can consist of several passes. A pass is basically an Open
Scene Graph StateSet. Ultimately all OpenGL and OSG modes and state
attributes will be accessible in techniques. State attributes -- that
is, technique properties that have children and are not just boolean
modes -- have an <active> parameter which enables or disables the
attribute. In this way a technique can declare parameters it needs,
but not enable the attribute at all if it is not needed; the decision
can be based on a parameter in the parameters section of the
effect. For example, effects that support transparent and opaque
geometry could have as part of a technique:

```
    <blend>
    <active><use>blend/active</use></active>
    <source>src-alpha</source>
    <destination>one-minus-src-alpha</destination>
    </blend>
```

So if the blend/active parameter is true blending will be activated
using the usual blending equation; otherwise blending is disabled.

Values of Technique Attributes

Values are assigned to technique properties in several ways:

  * They can appear directly in the techniques section as a
    constant. For example:
    <uniform>

61
```

```
  <name>ColorsTex</name>
  <type>sampler-1d</type>
  <value type="int">2</value>
</uniform>
```
* The name of a property in the parameters section can be
referenced using a "use" clause. For example, in the technique
section:
```
<material>
  <ambient><use>material/ambient</use></ambient>
</material>
```
Then, in the parameters section of the effect:
```
<parameters>
  <material>
    <ambient type="vec4d">0.2 0.2 0.2 1.0</ambient>
  </material>
</parameters>
```

It's worth pointing out that the "material" property in a
technique specifies part of OpenGL's state, whereas "material"
in the parameters section is just a name, part of a
hierarchical namespace.

* A property in the parameters section doesn't need to contain
a constant value; it can also contain a "use" property. Here
the value of the use clause is the name of a node in an
external property tree which will be used as the source of a
value. If the name begins with '/', the node is in
FlightGear's global property tree; otherwise, it is in a local
property tree, usually belonging to a model [NOT IMPLEMENTED
YET]. For example:
```
<parameters>
  <chrome-light><use>/rendering/scene/chrome-light</use></chrome-light>
</parameters>
```
The type is determined by what is expected by the technique
attribute that will ultimately receive the value. [There is
no way to get vector values out of the main property system
yet; this will be fixed shortly.] Values that are declared
this way are dynamically updated if the property node
changes.

```
OpenGL Attributes
-----------------


The following attributes are currently implemented in techiques:
alpha-test - children: active, comparison, reference
     Valid values for comparision:
        never, less, equal, lequal, greater, notequal, gequal,
        always

alpha-to-coverage - true, false

blend - children: active, source, destination, source-rgb,
     source-alpha, destination-rgb, destination-alpha
     Each operand can have the following values:
        dst-alpha, dst-color, one, one-minus-dst-alpha,
        one-minus-dst-color, one-minus-src-alpha,
        one-minus-src-color, src-alpha, src-alpha-saturate,
        src-color, constant-color, one-minus-constant-color,
        constant-alpha, one-minus-constant-alpha, zero

cull-face - front, back, front-back

lighting - true, false

material - children: active, ambient, ambient-front, ambient-back, diffuse,
     diffuse-front, diffuse-back, specular, specular-front,
     specular-back, emissive, emissive-front, emissive-back, shininess,
     shininess-front, shininess-back, color-mode

polygon-mode - children: front, back
     Valid values:
          fill, line, point

program
    vertex-shader
    geometry-shader
    fragment-shader
    attribute
    geometry-vertices-out - integer, max number of vertices emitted by geometry
                             shader
```

```
      geometry-input-type - points, lines, lines-adjacency, triangles,
                            triangles-adjacency
      geometry-output-type - points, line-strip, triangle-strip


render-bin - (OSG) children: bin-number, bin-name


rendering-hint - (OSG) opaque, transparent


shade-model - flat, smooth


texture-unit - has several child properties:
  unit - The number of an OpenGL texture unit
  point-sprite - true, false - Whether this should rendered as a point-sprite
  type - This is either an OpenGL texture type or the name of a
    builtin texture. Currently supported OpenGL types are 1d, 2d,
    3d which have the following common parameters:
      image (file name)
      filter - nearest, linear, [nearest|linear]-mipmap-[nearest|linear]
      mag-filter - nearest, linear, [nearest|linear]-mipmap-[nearest|linear]
      wrap-s - clamp, clamp-to-border, clamp-to-edge, mirror, repeat
      wrap-t - clamp, clamp-to-border, clamp-to-edge, mirror, repeat
      wrap-r - clamp, clamp-to-border, clamp-to-edge, mirror, repeat
      mipmap-control - control the mipmap on a per-channel basis.  Children:
        function-r - auto, average, sum, product, min, max
        function-g - auto, average, sum, product, min, max
        function-b - auto, average, sum, product, min, max
        function-a - auto, average, sum, product, min, max


    The following built-in types are supported:
      white - 1 pixel white texture
      noise - a 3d noise texture. (size parameter defines size of texture)
      light-sprite - a procedurally generated sprite suitable for point lights
      cubemap - build a cube-map.  Children:
        images - build from 6 images. Children: [positive|negative]-[x|y|z]
        image - build from a single cross-image


  environment
    mode - add, blend, decal, modulate, replace
    color
```

```
texenv-combine
  combine-[rgb|alpha] - replace, modulate, add, add-signed, interpolate, subtract,
  source[0|1|2]-[rgb|alpha] - constant, primary_color, previous, texture, texture[
  operand[0|1|2]-[rgb|alpha] -src-color, one-minus-src-color, src-alpha, one-minus-
  scale-[rgb|alpha]
  constant-color

texgen
  mode - object-linear, eye-linear, sphere-map, normal-map, reflection-map
  planes - s, t, r, q

uniform
  name
  type - float, float-vec3, float-vec4, sampler-1d, sampler-2d,
  sampler-3d

vertex-program-two-side - true, false

vertex-program-point-size - true, false
```

Inheritance

One feature not fully illustrated in the sample below is that
effects can inherit from each other. The parent effect is listed in
the "inherits-from" form. The child effect's property tree is
overlaid over that of the parent. Nodes that have the same name and
property index -- set by the "n=" attribute in the property tag --
are recursively merged. Leaf property nodes from the child have
precedence. This means that effects that inherit from the example
effect below could be very short, listing just new
parameters and adding nothing to the techniques section;
alternatively, a technique could be altered or customized in a
child, listing (for example) a different shader program. An example
showing inheritance Effects/crop.eff, which inherits some if its
values from Effects/terrain-default.eff.

FlightGear directly uses effects inheritance to assign effects to 3D
models and terrain. As described below, at runtime small effects are
created that contain material and texture values in a "parameters"

section. These effects inherit from another effect which references
those parameters in its "techniques" section. The derived effect
overrides any default values that might be in the base effect's
parameters section.

Generate

Often shader effects need tangent vectors to work properly. These
tangent vectors, usually called tangent and binormal, are computed
on the CPU and given to the shader as vertex attributes. These
vectors are computed on demand on the geometry using the effect if
the 'generate' clause is present in the effect file. Exemple :

```
  <generate>
    <tangent type="int">6</tangent>
    <binormal type="int">7</binormal>
    <normal type="int">8</normal>
  </generate>
```

Valid subnodes of 'generate' are 'tangent', 'binormal' or 'normal'.
The integer value of these subnode is the index of the attribute
that will hold the value of the vec3 vector.

The generate clause is located under PropertyList in the xml file.

In order to be available for the vertex shader, these data should
be bound to an attribute in the program clause, like this :

```
  <program>
    <vertex-shader>my_vertex_shader</vertex-shader>
    <attribute>
      <name>my_tangent_attribute</name>
      <index>6</index>
    </attribute>
    <attribute>
      <name>my_binormal_attribute</name>
      <index>7</index>
    </attribute>
  </program>
```

attribute names are whatever the shader use. The index is the one
declared in the 'generate' clause. So because generate/tangent has
value 6 and my_tangent_attribute has index 6, my_tangent_attribute
holds the tangent value for the vertex.

Default Effects in Terrain Materials and Models

Effects for terrain work in this way: for each material type in
materials.xml an effect is created that inherits from a single default
terrain effect, Effects/terrain-default.eff. The parameters section of
the effect is filled in using the ambient, diffuse, specular,
emissive, shininess, and transparent fields of the material. The
parameters image, filter, wrap-s, and wrap-t are also initialized from
the material xml. Seperate effects are created for each texture
variant of a material.

Model effects are created by walking the OpenSceneGraph scene graph
for a model and replacing nodes (osg::Geode) that have state sets with
node that uses an effect instead. Again, a small effect is created
with parameters extracted from OSG objects; this effect inherits, by
default, from Effects/model-default.eff. A larger set of parameters is
created for model effects than for terrain because there is more
variation possible from the OSG model loaders than from the terrain
system. The parameters created are:

  * material active, ambient, diffuse, specular, emissive,
    shininess, color mode
    * blend active, source, destination
    * shade-model
    * cull-face
  * rendering-hint
  * texture type, image, filter, wrap-s, wrap-t

Specifying Custom Effects

You can specify the effects that will be used by FlightGear as the
base effect when it creates terrain and model effects.

67

In the terrain materials.xml, an "effect" property specifies the name of the model to use.

In model .xml files, A richer syntax is supported. [TO BE DETERMINED]

Material animations will be implemented by creating a new effect that inherits from one in a model, overriding the parameters that will be animated.

Examples

The Effects directory contains the effects definitions; look there for examples. Effects/crop.eff is a good example of a complex effect.

Application

To apply an effect to a model or part of a model use:

```
  <effect>
    <inherits-from>Effects/light-cone</inherits-from>
    <object-name>Cone</object-name>
  </effect>
```

where <inherits-from> </inherits-from> contains the path to the effect you want to apply. The effect does not need the file extension.

NOTE:

Chrome, although now implemented as an effect, still retains the old method of application:

```
  <animation>
      <type>shader</type>
      <shader>chrome</shader>
      <texture>glass_shader.png</texture>
      <object-name>windscreen</object-name>
  </animation>
```

in order to maintain backward compatibility.

10 Electrical

```
Specifying and Configuring and Aircraft Electrical System
===========================================================


Written by Curtis L. Olson <curt@flightgear.org>

February 3, 2003 - Initial revision.



Introduction
============


The FlightGear electrical system model is an approximation.  We don't
model down to the level of individual electrons, but we do try to
model a rich enough subset of components so that a realistic (from the
pilot's perspective) electrical system may be implemented.  We try to
model enough of the general flow so that typical electrical system
failures can be implimented and so that the pilot can practice
realistic troubleshooting techniques and learn the basic structure and
relationships of the real aircraft electrical system.

An electrical system can be built from 4 major components: suppliers,
buses, outputs, and connectors.  Suppliers are things like batteries
and generators.  Buses collect input from multiple suppliers and feed
multiple outputs.  Outputs are not strictly necessary, but are
included so we can name generic output types and provide a consistent
naming scheme to other FlightGear subsystems.  Finally connectors
connect a supplier to a bus, or a bus to an output, and optionally can
specify a switch property (either a physical switch or a circuit
breaker.)

At run time, the structure specified in the electrical system config
file is parsed and a directional graph (in the computer science sense)
is built.  Each frame, the current is propagated through the system,
```

starting at the suppliers, flowing through the buses, and finally to
the outputs. The system follows the path of connectors laid out in
the config file and honors the state of any connector switch.

Suppliers
=========

A supplier entry could look like the following:

```
  <supplier>
    <name>Battery 1</name>
    <prop>/systems/electrical/suppliers/battery[0]</prop>
    <kind>battery</kind>
    <volts>24</volts>
    <amps>60</amps>    <!-- WAG -->
  </supplier>
```

<name> can be anything you choose to call this entry.
<prop> is the name of a property that will be updated with the state
       of this supplier.
<kind> can be "battery", "alternator", or "external".
<volts> specifies the volts of the source
<amps> specifies the amps of the source

Currently <volts> and <amps> are not really modeled in detail. This
is more of a place holder for the future.

For alternators, you must additionally specify:

```
    <rpm-source>/engines/engine[0]/rpm</rpm-source>
```

The value of the rpm source determines if the generator is able to
produce power or not.

Buses
=====

A bus entry could look like the following:

```
  <bus>
    <name>Essential/Cross Feed Bus</name>
    <prop>/systems/electrical/outputs/bus-essential</prop>
    <prop>/systems/electrical/outputs/annunciators</prop>
    <prop>/systems/electrical/outputs/master-switch</prop>
  </bus>
```

<name> is whatever you choose to call this bus

You can have an arbitrary number of <prop> entries. Each entry is the
name of a property that will be updated with the value of the current
at that bus. This allows you to wire devices directly to the bus but
does not allow you to insert a switch or circuit breaker in between.
See "Outputs" and "Connectors" if you want to do that.

Outputs
=======

An output entry could look like the following:

```
  <output>
    <name>Starter 1 Power</name>
    <prop>/systems/electrical/outputs/starter[0]</prop>
  </output>
```

An output isn't entirely unlike a bus, but it's nice conceptually to
have a separate entity type. This enables us to specify a common set
of output property names so that other subsystems can automatically
work with any electrical system that follows the same conventions. An
output lives on the other side of a switch, so this is how you can
wire in cockpit switches to model things like fuel pump power,
avionics master switch, or any other switch on the panel.

<name> is whatever you choose to call this bus

You can have an arbitrary number of <prop> entries. Each entry is the
name of a property that will be updated with the value of the current
at that bus. This allows you to wire devices directly to the bus but

```

does not allow you to insert a switch or circuit breaker in between.
See "Outputs" and "Connectors" if you want to do that.

Other FlightGear subsystems can monitor the property name associated
with the various outputs to decide how to render an instrument,
whether to run the fuel pump, whether to spin a gyro, or any other
subsystem that cares about electrical power.


Connectors
==========


An connector entry could look like the following:

```
  <connector>
    <input>Alternator 1</input>
    <output>Virtual Bus 1</output>
    <switch>/controls/switches/master-alt</switch>
    <initial-state>off</initial-state> <!-- optional tag -->
  </connector>
```

A connector specifies and input, and output, and any number of
switches that are wired in series. In other words, all switches need
to be true/on in order for current to get from the input to the output
of the connector.

<input> specifies the <name> of the input. Typically you would
specify a "supplier" or a "bus".

<output> specifies the <name> of the output. Typically you would
specify a bus or an output.

You can have an arbitrary number of <switch> entries. The switches
are wired in series so all of them need to be on (i.e. true) in order
for current to pass to the output.

Note: by default the system forces any listed switches to be true.
The assumption is that not every aircraft or cockpit may impliment
every available switch, so rather than having systems be switched off,
with no way to turn them on, we default to switched on.

This is a problem however with the starter switch which we want to be initialized to "off". To solve this problem you can specify
<initial-state>off</initial-state> or
<initial-state>on</initial-state> Switches default to on, so you really only need to specify this tag if you want the connector's switch to default to off.

Summary
=======

The electrical system has a lot of power and flexibility to model a variety of electrical systems. However, it is not yet perfect or finished. One major weakness is that it doesn't yet model degraded battery or generator power, and it doesn't model the "charge" of the batteries in case of a generator failure.

11   Embedded-resources

-*- coding: utf-8; fill-column: 72; -*-

The Embedded Resources System
=============================

This document gives an overview of FlightGear's embedded resources system and related classes. For specific information on the C++ functions, the reference documentation is in the corresponding header files.

Contents
----

1. The CharArrayStream and ZlibStream classes
2. The âembedded resourcesâ system
3. About the XML resource declaration files
4. The EmbeddedResourceProxy class

Introduction

The embedded resources system allows FlightGear to use data from files
without relying on FG_ROOT to be set. This can be used, for instance, to
grab the contents of XML files at FG build time, from any repository[1],
and use said contents in the C++ code. The term âembeddedâ is used to
avoid confusion with the ResourceProvider and ResourceManager classes
provided by SimGear, which have nothing to do with the system described
here.

The embedded resources system relies on classes present in
simgear/io/iostreams/{zlibstream.cxx,CharArrayStream.cxx}, which were
implemented as a way to address a concern that embedding a few XML files
in the fgfs binary could use precious memory. The resource compiler
(fgrcc) compresses resources before writing them in C++ form---except
for some extensions, and it's configurable on a per-resource basis
anyway. Then, the EmbeddedResourceManager instance, which lives in the
fgfs process, can decompress them on-the-fly, incrementally,
transparently. So, there is really no reason to worry about memory
consumption, even for several dozens of XML files.

fgrcc is the resource compiler: it turns arbitrary files into C++ code
the EmbeddedResourceManager can make use of, in order to âserveâ the
files' contents at runtime. It is named this way, because it fulfills
the same role as Qt's rcc tool. It supports a thin superset of the
XML-based format used by rcc for declaring resources[2][3].
'fgrcc --help' gives a lot of info.

1) The CharArrayStream and ZlibStream classes
   ~~~~~~

The CharArrayStream* files in simgear/io/iostreams/ implement
CharArrayStreambuf and related IOStreams classes for working with char
arrays, namely:
  - CharArrayStreambuf   subclass of std::streambuf   stream buffer
  - ROCharArrayStreambuf   subclass of CharArrayStreambuf   stream buffer
  - CharArrayIStream   subclass of std::istream   input stream

```
    - CharArrayOStream      subclass of std::ostream       output stream
    - CharArrayIOStream     subclass of std::iostream      input/output stream
```

(in the 'simgear' namespace, of course)

CharArrayStreambuf is a stream buffer class allowing to read from, and
write to char arrays (std::strstream has been deprecated since C++98).
Contrary to std::strstream, this class does no dynamic allocation: it is
very simple, strictly staying for both reads and writes within the
bounds of the buffer specified in its constructor. Contrary to
std::stringstream, CharArrayStreambuf allows one to work on an array of
char (that could be static data, on the stack, whatever) without having
to make a whole copy of it.

ROCharArrayStreambuf is a read-only subclass of CharArrayStreambuf
(useful for const-correctness). CharArrayIStream, CharArrayOStream and
CharArrayIOStream are very simple convenience stream classes using
either CharArrayStreambuf or ROCharArrayStreambuf as their associated
stream buffer class.

While these classes can be of general-purpose usefulness, the particular
reason they have been written for is to make the embedded resources
system clean and memory-friendly. Concretely, this system supports both
compressed and uncompressed resources, all of which can be read from
their respective static arrays like this (think pipelines):

```
static char array
(uncompressed          --------------->      data available via an std::istream
 resource)           CharArrayIStream          or std::streambuf interface
                or ROCharArrayStreambuf


static char array
(compressed          ---------------> compressed data ------------------->   ditto
 resource)         CharArrayIStream                ZlibDecompressorIStream
                                                or ZlibDecompressorIStreambuf
```

where ditto = uncompressed data available via an std::istream or
             std::streambuf interface

So, whether the resource data stored in static arrays by fgrcc is

compressed or not, end-user code can read it in uncompressed form using an std::istream or std::streambuf interface, which means the resource never needs to be copied in memory a second time. This is particularly interesting with compressed resources, because:

1) The in-memory static data is much smaller in general than the uncompressed contents, and it's the only one we really have to âpayâ for if one uses these stream-based interfaces.

2) The data is transparently decompressed on-demand as the end-user code reads from the ZlibDecompressorIStream or ZlibDecompressorIStreambuf instance.

In other words, these CharArrayStream classes complement the ones in zlibstream.cxx and make it easy to implement all kinds of pipelines to incrementally read or write, and possibly on-the-fly compress or decompress data from or to in-memory buffers (cf. writeCompressedDataToBuffer() in simgear/simgear/embedded_resources/embedded_resources_test.cxx, or ResourceCodeGenerator::writeEncodedResourceContents() in flightgear/src/EmbeddedResources/fgrcc.cxx for examples).

Since all of these provide standard IOStreams interfaces, they can be easily plugged into existing code. For instance, readXML() in simgear/simgear/xml/easyxml.cxx and readProperties() in simgear/props/props_io.cxx can incrementally read and parse data from an std::istream instance, and thus are able to directly read from a resource containing the compressed version of an XML file.

This incremental stuff is of course really interesting with large resources... which probably won't be used in FlightGear, in order not to waste RAM[4][5]. The EmbeddedResourceManager also has a getString() method to simply get an std::string when you don't care about the fact that this operation, by std::string design, will necessarily make a copy of the whole resource contents (in uncompressed form in the case of a compressed resource). This getString() method should be convenient and quite acceptable for reasonably-sized resources.

Finally, all of these classes---CharArray*Stream*, the classes in zlibstream.cxx, the EmbeddedResourceManager and related classes---can

handle text and binary data in exactly the same way (std::string doesn't
care, and neither do the other classes).

2) The âembedded resourcesâ system
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

The embedded resources system works this way:

  (1) The fgrcc resource compiler reads an XML file which has almost the
      same syntax[2] as Qt's .qrc files[3] and writes a .cxx file
      containing:
        - static char arrays initialized with resource contents
          (possibly compressed, this is automatic unless explicitly
          specified in the XML file);
        - a function definition containing calls to
          EmbeddedResourceManager::addResource() that register each of
          these resources with the EmbeddedResourceManager instance.

      If you pass the --output-header-file option to fgrcc, it also
      writes a header file that goes with the generated .cxx file. For
      other options, see the output of 'fgrcc --help'.

      It is quite possible to call fgrcc several times, each time with a
      different (XML input file, .cxx/.hxx output files) tuple: for
      instance, one call for resources present in the FlightGear repo,
      and possibly another call for resources in FGData. The point of
      this is that paths in the XML input file should be relative to
      avoid being system-dependent, and fgrcc accepts a --root option to
      indicate what you want them to be relative to, in order to let it
      find the real files. Thus, on a first invocation of fgrcc, one can
      make --root point to a path to the FlightGear repository when
      building, and on the second call use it to indicate a path to the
      FGData repository. Other variations are possible, of course.

      Notes:

        1) The example given here with FGData would *not* freeze the
           FGData location at FG compile time; this is only to allow
           files from FGData to be turned into generated .cxx files

inside the FG source tree, that will make their contents
available as embedded resources at runtime.

2) At the time of this writing, resources from the FlightGear
repository are compiled at build time, and resources from the
FGData repository are compiled offline using the
'rebuild-fgdata-embedded-resources' script[6] (a
convenience wrapper for fgrcc), before being committed to the
FlightGear repository.

(2) SimGear contains an EmbeddedResourceManager class with, among
others, createInstance() and instance() methods similar to the
ones of NavDataCache. See [7] for the corresponding code.

FlightGear creates an EmbeddedResourceManager instance at startup
and calls the various init functions generated by fgrcc, each of
which registers the resources present in its containing .cxx file
(using EmbeddedResourceManager::addResource()).

End-user FG code can then use EmbeddedResourceManager methods such
as getResource(), getString(), getStreambuf() and getIStream()
to access resource contents:
  - getResource() returns an
    std::shared_ptr<const AbstractEmbeddedResource>
  - getString() returns an std::string
  - getStreambuf() returns an std::unique_ptr<std::streambuf>
  - getIStream() returns an std::unique_ptr<std::istream>

AbstractEmbeddedResource is an abstract base class that you can
think of as a resource descriptor: it points to (not contains!)
the resource data (which is normally of static storage class), and
contains + gives access to metadata such as the compression type
and resource size (compressed and uncompressed).

AbstractEmbeddedResource currently has two derived concrete
classes: RawEmbeddedResource for resources stored as-is
(uncompressed) and ZlibEmbeddedResource for resources compressed by
fgrcc. It's quite easy to add new subclasses if wanted, e.g. for
LZMA compression or other things.

Resource fetching requires two things:

- an std::string key (fgrcc manipulates them with SGPath, but the EmbeddedResourceManager code in SimGear is so far completely agnostic of the kind of data stored in keys; this could be changed, though, if we wanted for example to be able to query at runtime all available resources in a given âvirtual directoryâ);

- a âlocaleâ name, similar to what FlightGear's XML translation files and FGLocale use. We used double quotes here, because fgrcc and the EmbeddedResourceManager expect âlocaleâ names to be of one of these forms:
    * empty string: default locale, typically but not necessarily English (it is âengineering Englishâ in FlightGear, i.e., English written by programmers in the code, before translators possibly fix it up :)
    * en, fr, de, es, it...
    * en_GB, en_US, fr_FR, fr_CA, de_DE, de_CH, it_IT...

    There is no encoding part, contrary to POSIX locales, hence the use of double quotes around the term âlocaleâ in this context.

The FGLocale::getPreferredLanguage() method returns the preferred âlocaleâ in the form described above, according to user choice (from fgfs' --language option) and/or settings (system locale). This allows FG to tell the EmbeddedResourceManager the preferred âlocaleâ for resource fetching (same syntax as in Qt's rcc tool for declaration in the XML file, using the 'lang' attribute on 'qresource' elements).

[Regarding the default locale, the way things are currently set up, I would use no 'lang' attribute for resources suitable for English in the XML input file for fgrcc, except when a country-specific variant is desired (en_GB, en_US, en_AU...). In such a case, there should also be a generic variant with no 'lang' attribute declared for the same resource virtual path. This matches what I did for FGLocale::getPreferredLanguage(), that maps unset locales and locales such as C and C.UTF-8 to the default locale for the EmbeddedResourceManager, which is the

empty string. This is a matter of policy, of course, and could be
changed if desired.]

The EmbeddedResourceManager class has getLocale() and
selectLocale() methods to manage the _selected locale_. Each
resource-fetching method of this class (getResourceOrNullPtr(),
getResource(), getString(), getStreambuf() and getIStream()) has
two overloads:
  - one taking only a virtual path (the key mentioned above);
  - one taking a virtual path and a âlocaleâ name.

(we'll write âlocaleâ without enclosing double-quotes from now on,
otherwise it gets too painful to read; but we're *not* talking
about POSIX-style locales ending with an encoding part)

The first kind of overload uses the selected locale to look up the
resource, whereas the second kind uses the explicitly specified
locale. Then resource lookup behaves as one could expect. For
instance, assuming a resource is looked up for in the "fr_FR"
locale, then the EmbeddedResourceManager tries in this order:
  - "fr_FR";
  - if no resource has been registered for "fr_FR" with the provided
    virtual path, it then tries with the "fr" locale;
  - if this is also unsuccessful, it finally tries with the default
    locale: "";
  - if this third attempt fails, the resource-fetching method
    throws an sg_exception, except for getResourceOrNullPtr(),
    which returns a null
    std::shared_ptr<const AbstractEmbeddedResource> instead.

To see how this is used, you can look at
simgear/simgear/embedded_resources/embedded_resources_test.cxx. The
only difference with real use is that in this file, resource
contents and registering calls with the EmbeddedResourceManager
have been written manually instead of by fgrcc. Apart from
embedded_resources_test.cxx, here are two examples of client usage
of the EmbeddedResourceManager:

(a) With EmbeddedResourceManager::getString():

```
#include <simgear/embedded_resources/EmbeddedResourceManager.hxx>
#include <simgear/debug/logstream.hxx>

[...]

const auto& resMgr = simgear::EmbeddedResourceManager::instance();
SG_LOG(SG_GENERAL, SG_INFO,
       "Resource contents: '" <<
       resMgr->getString("/virtual/path/to/resource") << "'");
```

(b) With EmbeddedResourceManager::getIStream():

```
#include <cstddef>                  // std::size_t
#include <simgear/io/iostreams/sgstream.hxx>
#include <simgear/embedded_resources/EmbeddedResourceManager.hxx>

[...]

sg_ofstream outFile(SGPath("/tmp/whatever"));
if (!outFile) {
  <handle open error>
}

const auto& resMgr = simgear::EmbeddedResourceManager::instance();
auto resStream = resMgr->getIStream("/virtual/path/to/resource");
// One possible way of handling errors from resStream[8]:
// resStream->exceptions(std::ios_base::badbit);

constexpr std::size_t bufSize = 4096;
std::unique_ptr<char[]> buf(new char[bufSize]); // intermediate buffer

do {
  resStream->read(buf.get(), bufSize);
  outFile.write(buf.get(), resStream->gcount());
} while (*resStream && outFile); // resStream *points* to an std::istream

<handle possible errors that might have caused to loop to stop
prematurely>
```

3) About the XML resource declaration files
   ~~~~~~

You may want to read the output of 'fgrcc --help', which explains a few
things, in particular how to write an XML resource declaration file that
fgrcc can use. At the time of this writing, such files are already
present as flightgear/src/EmbeddedResources/FlightGear-resources.xml and
flightgear/src/EmbeddedResources/FGData-resources.xml in the FlightGear
repository. In case you need resources from elsewhere, it's easy to add
other XML resource declaration files:

  1) If you want the .cxx/.hxx resource files to be automatically
     generated as part of the FlightGear build:

     Copy and adapt the add_custom_command() call in
     flightgear/src/Main/CMakeLists.txt[9] that invokes fgrcc on
     flightgear/src/EmbeddedResources/FlightGear-resources.xml.

  2) In flightgear/src/Main/CMakeLists.txt, add paths for your new
     fgrcc-generated .cxx and .hxx files to the SOURCES and HEADERS
     CMake variables for the 'fgfs' target.

  3) Assuming you passed for instance
     --init-func-name=initFoobarEmbeddedResources in step 1, add a call
     to initFoobarEmbeddedResources() after this code in fgMainInit()
     (flightgear/src/Main/main.cxx):

      simgear::EmbeddedResourceManager::createInstance();
      initFlightGearEmbeddedResources();

4) The EmbeddedResourceProxy class
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

SimGear contains an EmbeddedResourceProxy class that allows one to
access real files or embedded resources in a unified way. When using it,
one can switch from one data source to the other with minimal code
changes, possibly even at runtime (in which case there is obviously no
code change at all).

Sample usage (from FlightGear):

```
simgear::EmbeddedResourceProxy proxy(globals->get_fg_root(), "/FGData");
proxy.setUseEmbeddedResources(false); // can also be set via the constructor

std::string s = proxy.getString("/some/path");
std::unique_ptr<std::istream> streamp = proxy.getIStream("/some/path");
```

This example would retrieve contents from the real file
$FG_ROOT/some/path. If true had been passed in the
proxy.setUseEmbeddedResources() call, it would instead have used the
default-locale version of the embedded resource whose virtual path is
/FGData/some/path.

For more information about this class, see [10] and [11].

Footnotes
=========

[1] E.g., FlightGear or FGData, as long as the path to the latter is
    provided to the FG build system, which is currently possible but not
    required (passing '-D FG_DATA_DIR:PATH=...' to CMake when
    configuring the FlightGear build).

[2] The differences with the QRC format[3] are explained in the output
    of 'fgrcc --help'. Here is the relevant excerpt:

```
,----
| 1. The <!DOCTYPE RCC> declaration at the beginning should be omitted (or
|    replaced with <!DOCTYPE FGRCC>, however such a DTD currently doesn't
|    exist). I suggest to add an XML declaration instead, for instance:
|
|       <?xml version="1.0" encoding="UTF-8"?>
|
| 2. <RCC> and </RCC> must be replaced with <FGRCC> and </FGRCC>,
|    respectively.
|
| 3. The FGRCC format supports a 'compression' attribute for each 'file'
|    element. At the time of this writing, the allowed values for this
```

```
|     attribute are 'none', 'zlib' and 'auto'. When set to a value that is
|     not 'auto', this attribute of course bypasses the algorithm for
|     determining whether and how to compress a given resource (algorithm
|     which relies on the file extension).
|
| 4. Resource paths (paths to the real files, not virtual paths) are
|     interpreted relatively to the directory specified with the --root
|     option. If this option is not passed to 'fgrcc', then the default root
|     directory is the one containing INFILE, which matches the behavior of
|     Qt's 'rcc' tool.
'----
```

[3] http://doc.qt.io/qt-5/resources.html

[4] The main reason why I wrote the classes in
    simgear/simgear/io/iostreams/{CharArrayStream,zlibstream}.cxx is
    thus not to maximize memory-efficiency with very large resources;
    rather, it is to make the implementation of the following parts
    simple, clean and modular:
      - the resource compiler (fgrcc);
      - the EmbeddedResourceManager.

[5] The EmbeddedResourceManager architecture would make it quite easy to
    also support runtime loading of resources from files (a thing the Qt
    resource system supports), but it is not very clear how interesting
    this would be, compared to having the files loaded from $FG_ROOT.
    Well, maybe for large files [apt.dat.gz & Co] that we would want to
    load but not see in the FGData repository at all. But then there
    would be the requirement, of course, that âsomethingâ puts the files
    in a clearly-defined, platform-dependent location known to the
    EmbeddedResourceManager.

[6] https://sourceforge.net/p/flightgear/fgmeta/ci/next/tree/python3-flightgear/rebu

[7] https://sourceforge.net/p/flightgear/simgear/ci/next/tree/simgear/embedded_resou

[8] We know that in some buggy C++ implementations, the
    std::ios_base::failure exception can't be caught, at least not under
    its name, due to some ABI compatibility mess:

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=66145

    However, it stills causes the program to abort, and since this
    error handling technique makes for much more readable and less
    error-prone code, I think it's still a good way to handle IOStreams
    errors even now, unless you really need to *catch* the
    std::ios_base::failure exception.

[9] flightgear/CMakeModules/GenerateFlightgearResources.cmake in my
    'i18n-and-init-work-v2' branch (not merged into 'next' at the time
    of this writing).

[10] https://sourceforge.net/p/flightgear/simgear/ci/next/tree/simgear/embedded_resou

[11] https://sourceforge.net/p/flightgear/simgear/ci/next/tree/simgear/embedded_resou

12   Fgjs

```
fgjs -- a small program for creating a basic FlightGear joystick
        configuration

fgjs requires plib to be installed on your system.  If you've
successfully installed and built FlightGear then you should be
all set


Build instructions
At this point, fgjs has only been built and tested under Linux,
so the makefile is a simple one. cd into the directory in which
the fgjs source resides and type 'make' and, if you are lucky,
all will go well.  You can e-mail me (apeden@earthlink.net) any
changes needed to make it work on other systems.  It's quite
possible that this program will become part of the regular
FlightGear package so


Running
Set up your joystick and make sure it works with js_demo from the
FlightGear distribution.  Upon executing fgjs, it will prompt you
to move the control you wish to use for elevator, ailerons, etc.
Note that when being prompted for an analog control, you can skip
```

the current one by pressing any button and vice-versa when being
prompted for a button. You may want to do this if for, as an
example, rudder if you have only one joystick or your joystick
doesn't have as many analog axes as FlightGear supports.

Once you've run with this configuration, you may wish to tune
the dead-band a bit (see fgfsrc.js) as the default, 0.02, may
be too narrow for your particular hardware/taste.

And last, but not least, this thing needs a GUI!!!! Hopefully,
the joystick handling code and interface code are separate
enough that using that a GUI version could be built using this
source as a starting point.

13   Flightrecorder

FlightGear Flight Recorder Mini-HOWTO

Thorsten Brehm
Started in August 2011
Last revised: 2011-09-26

FlightGear provides a customizable flight recorder capable of capturing
any selection of properties described via XML configuration files.
The recorder is currently used for the replay system.

```
Feature Brief
-------------
* Generic recording system, adaptable to any aircraft/data, provided that
  data is accessible via the property tree. No hard-coded selections or
  assumptions on properties to be recorded.
* Configuration read from XML files or the property tree itself.
* Interpolation method configurable per recorded/replayed signal.
* Adaptable recording resolution per signal.
* Multiple configurations supported.


Quick Start: Basic Configuration
--------------------------------
To configure and adapt the flight recorder, add a "/sim/flight-recorder"
section to your aircraft -set.xml file.

Example:

<sim>

  <!-- ... -->

  <flight-recorder>
    <replay-config type="int">0</replay-config>
    <config n="0"
        include="/Aircraft/Generic/flightrecorder/generic-piston-propeller-1.xml">
      <name type="string">My Aircraft's Flight Recorder</name>
      <!-- Custom properties -->
      <signal>
        <type>float</type>
        <property type="string">/controls/gear/nosegear-steering-cmd-norm</property>
        <interpolation>linear</interpolation>
      </signal>
      <!-- More custom signals here -->
    </config>
  </flight-recorder>

      <!-- ... -->

</sim>
```

Default type for each signal is "float". Default "interpolation" method
is "linear" (for float/double). Default values may be omitted. See
configuration details below.

Generic Configuration Files

Select one of the default configuration files to specify the basic
properties to be recorded. It's not recommended to specify all
properties to be recorded individually.
The following generic files are provided:

* /Aircraft/Generic/flightrecorder/generic-piston-propeller-4.xml
  Matches propeller aircraft with 4 piston engines, 4 tanks,
  3 retractable gear.
  It is the same configuration that was hard-coded for the replay system
  up to FlightGear 2.4.0. To provide backward compatibility this
  configuration is loaded by default, unless an aircraft provides a
  specific flight recorder configuration.

* /Aircraft/Generic/flightrecorder/generic-piston-propeller-1.xml
  Matches propeller aircraft with 1 piston engines, 2 tanks, 3 fixed
  gear.

* /Aircraft/Generic/flightrecorder/generic-turboprop-2.xml
  Matches turboprop aircraft with 2 turbines/propellers, 4 tanks,
  3 retractable gear.

* /Aircraft/Generic/flightrecorder/generic-jet.xml
  Matches jet aircraft with 2 jet engines, 4 tanks.

* /Aircraft/Generic/flightrecorder/generic-glider.xml
  Matches gliders (no engines, no tanks, single fixed gear).

* /Aircraft/Generic/flightrecorder/generic-helicopter.xml
  Matches helicopters with main and tail rotor (tested with YASim).

If none of the generic files matches your aircraft, simply use a
configuration which covers more than you need. Alternatively, copy the

contents of one of these generic files to your aircraft, and adapt as
needed (see below).

FDM experts are welcome to add more generic configuration files to
/Aircraft/Generic/flightrecorder - such as YASim-/JSBSim-specific
configurations, and configurations for other types of aircraft
(balloons, airships, ...).

Generic Components

The generic configuration files in turn include a set of generic
components. If you copy the contents of a generic file to your aircraft,
you can adapt the components to your needs. See examples.
It is not recommended to copy the contents of the _component_ files to
an aircraft though (causes too much hassle and dependencies).

Engine Selection:
 * /Aircraft/Generic/flightrecorder/components/engine-jet.xml
   Records properties of a single jet engine.
   For multiple jet engines, use "count". Example for 4 jets:
     <signals include="/Aircraft/Generic/flightrecorder/components/engine-jet.xml">
       <count>4</count>
     </signals>

 * /Aircraft/Generic/flightrecorder/components/engine-piston.xml
   Records properties of a single piston engine and propeller.
   For multiple piston engines, use "count" (see "jet" example).

 * /Aircraft/Generic/flightrecorder/components/rotor.xml
   Records properties of a single helicopter rotor (tested with YASim).
   To use this, provide the base property path to the rotor as "prefix".
   Example recording the rotor below "/rotors/main":
     <signals include="/Aircraft/Generic/flightrecorder/components/rotor.xml">
         <prefix type="string">/rotors/main</prefix>
     </signals>

Gear Selection:
 * /Aircraft/Generic/flightrecorder/components/gear-fixed.xml
   Records properties of a single non-retractable gear.

For multiple fixed gear, use "count" (see "jet" example).

  * /Aircraft/Generic/flightrecorder/components/gear-retractable.xml
    Records properties of a single retractable gear.
    For multiple retractable gear, use "count" (see "jet" example).

Tanks:
  * /Aircraft/Generic/flightrecorder/components/tanks.xml
    Records properties of a single fuel tank.
    For multiple fuel tanks, use "count" (see "jet" example).

Other:
  * /Aircraft/Generic/flightrecorder/components/surfaces.xml
    Records properties of flight control surfaces. Include this
    for aircraft (with wings). Not useful for helicopters,
    balloons, ...

  * /Aircraft/Generic/flightrecorder/components/faults-engines.xml
    Records fault properties of a single engine. Only include this
    if your aircraft supports fault simulation.
    For multiple engines, use "count" (see "jet" example). If used,
    it should be compined with piston or jet engine.

  * /Aircraft/Generic/flightrecorder/components/environment.xml
    Records properties of environment/weather (visibility,
    temperature - but _not_ cloud position...).

  * /Aircraft/Generic/flightrecorder/components/position.xml
    Records properties of a the aircrafts main position (latitude,
    longitude, velocities, ...).
    This is the most important component. Always include this.

  * /Aircraft/Generic/flightrecorder/components/controls.xml
    Records most important flight controls (rudder, aileron,
    elevator, ...). Always include this.

Custom Properties

When the generic or component files are not be sufficient to record or

replay aircraft-specific effects, you can add custom properties (signals
to be recorded) to the configuration.
Each signal consits of a recording type/resolution (which does _not_
need to match the actual type in the property tree!), the path to the
property and interpolation type.

Example recording some additional custom properties:

```
  <sim>
    <flight-recorder>
      <config n="0"
        include="/Aircraft/Generic/flightrecorder/generic-piston-propeller-1.xml">
        <!-- Add custom properties here -->
        <signal>
          <type>float</type>
          <property type="string">/controls/gear/nosegear-steering-cmd-norm</property
        </signal>
        <signal>
          <type>double</type>
          <interpolation>rotational-deg</interpolation>
          <property type="string">/ai/model/carrier/alpha-angle-deg</property>
        </signal>
        <signal>
          <type>bool</type>
          <property type="string">/controls/panel/custom-switch</property>
        </signal>
      </config>
    </flight-recorder>
  </sim>
```

Signal Configuration

Template:

```
  <signal>
    <type>bool</type>
    <interpolation>angular-deg</interpolation>
    <property type="string">/controls/panel/custom-switch</property>
  </signal>
```

* type: The signal's type specifies the recording resolution - not the

type of the original property. The following types are supported:

- double: 8 byte/sample
- float: 4 byte/sample (default)
- int: 4 byte/sample, integer
- int16: 2 byte/sample, integer
- int8:   1 byte/sample, integer
- bool: 1 bit/sample (yes, 1 bit. 8 bools per byte).

String type is unsupported (too expensive).

* interpolation: Specifies how values are interpolated during replay, i.e.
  when replay is in slow-motion mode and more frames/second are required
  than recorded, or when replaying data from the medium/long term memory.
  Supported methods:
    - discrete: No interpolation. Default for integer/bool types.
    - linear: Standard linear interpolation. Default for float/double.
    - angular-rad (or angular): Angular values in radians (0-2pi).
    - angular-deg: Angular values in degrees (0-360).

* property: Path to the property to be recorded.

Advanced Configuration

- Multipe recorder configurations for a single aircraft are supported
  (multiple "<config n=..>" sections for n=0,1,...).
  Active configuration to be used for the replay system is selected via
     /sim/flight-recorder/replay-config (= 0,1,...).
  This can be useful for specific recorders for specific scenarios,
  which should not be used by default. For example, a specific recorder
  configuration could be provided which also records the position of
  an aircraft carrier, of other AI aircraft, ...
  This may also be useful for future use, i.e. to select another flight
  recorded configuration for a different purpose, such as for the
  multiplayer system.

- Flight recorder configuration can be adapted during run-time
  (configuration is visible in the property browser below
  /sim/flight-recorder). However it is necessary to reset (reinit) the

replay subsystem first – which also erases earlier recordings. It is
not possible to mix recordings of different configurations on to a
single "tape".

- Each configuration should be given a name. Useful for a (future)
  selection GUI, when multiple configurations are available.

Optimizing Performance

- Recording properties consumes memory and also CPU time. A few
  additional properties don't matter much, but avoid execessive numbers.
  Reduce the resolution (type) of signals to the minimum necessary to
  save space.
- Use "bool" types where possible, they are most efficient.
- Avoid recording with "double" resolution (type "double"). Use "float"
  instead – even if the original property in the property tree is a
  "double" (almost all of them do). "float" precision is almost always
  sufficient for recording/replay purposes, with few exceptions (like
  latitude/longitude properties).
- Use int16/int8 for "small" integer values.

Recording/Replay Limits

- All properties can be recorded, however, only writable properties can
  be replayed. Properties marked as read-only, or tied properties not
  implementing the "set" method cannot be replayed.
- Replaying a property overwrites the property's value. However, other
  sources may also write to the same property – such as Nasal code,
  autopilot rules etc. When multiple sources "fight" over a property's
  value then the last update "wins" – resulting in a dependency to an
  unknown/random sequence. Hence, during deplay, try to disable other
  sources writing to properties which were recorded and should be
  replayed.
  If the other source cannot be disabled, check if you're recording the
  right property. It may be better to record the input properties of the
  other source instead (i.e. the inputs processed by the Nasal or
  autopilot rule).

```
__end__
```

14 Gui

FlightGear GUI Mini-HOWTO

David Megginson
Started: 2003-01-20
Last revised: 2003-01-20

FlightGear creates its drop-down menubar and dialog boxes from XML
configuration files under $FG_ROOT/gui. This document gives a quick
explanation of how to create or modify the menubar and dialogs. The
toolkit for the FlightGear GUI is PUI, which is part of plib.

All of the XML files use the standard FlightGear PropertyList format.

MENUBAR

FlightGear reads the configuration for its menubar from
$FG_ROOT/gui/menubar.xml. The file consists of a series of top-level
elements named "menu", each of which defines on of the drop-down
menus, from left to right. Each menu contains a series of items,
representing the actual items a user can select from the menu, and
each item has a series of bindings that FlightGear will activate when
the user selects the item.

Here's a simplified grammar:

```
  [menubar] : menu*

  menu : label, item*

  item : label, enabled, binding*
```

The bindings are standard FlightGear bindings, the same as the ones
used for the keyboard, mouse, joysticks, and the instrument panel.
Any commands allowed in those bindings are allowed here as well.

Here's an example of a simple menubar with a "File" drop-down menu and
a single "Quit" item:

```
<PropertyList>

 <menu>
  <label>File</label>

  <item>
   <label>Quit</label>
   <binding>
    <command>exit</command>
   </binding>
  </item>

</PropertyList>
```

PUI menus do not allow advanced features like submenus or checkmarks.
The most common command to include in a menu item binding is the
'dialog-show' command, which will open a user-defined dialog box as
described in the next section.

DIALOGS

The configuration files for XML dialogs use a nested structure to set
up dialog boxes. The top-level always describes a dialog box, and the
lower levels describe the groups and widgets that make it up. Here is
a simple, "hello world" dialog:

```
<PropertyList>

 <name>hello</name>

 <width>150</width>
```

```
    <height>100</height>
    <modal>false</modal>
    <draggable>true</draggable>
    <resizable>true</resizable>

    <text>
     <x>10</x>
     <y>50</y>
     <label>Hello, world</label>
     <color>
      <red>1.0</red>
      <green>0.0</green>
      <blue>0.0</blue>
     </color>
    </text>

    <button>
     <x>40</x>
     <y>10</y>
     <legend>Close</legend>
     <binding>
      <command>dialog-close</command>
     </binding>
    </button>

  </PropertyList>
```

The dialog contains two sub-objects: a text field and a button. The
button contains one binding, which closes the active dialog when the
user clicks on the button.

Coordinates are pseudo-pixels. The screen is always assumed to be
1024x768, no matter what the actual resolution is. The origin is the
bottom left corner of the screen (or parent dialog or group); x goes
from left to right, and y goes from bottom to top.

All objects, including the top-level dialog, accept the following
properties, though they will ignore any that are not relevant:

 x - the X position of the bottom left corner of the object, in

pseudo-pixels. The default is to center the dialog.

y - the Y position of the bottom left corner of the object, in
  pseudo-pixels. The default is to center the dialog.

width - the width of the object, in pseudo-pixels. The default is
  the width of the parent container.

height - the height of the object, in pseudo-pixels. The default is
  the width of the parent container.

border - the border thickness, in pseudo-pixels. The default is 2.

color - a subgroup to specify the dialogs color:
 red   - specify the red color component of the color scheme.
 green - specify the green color component of the color scheme.
 blue - specify the blue color component of the color scheme.
 alpha - specify the alpha color component of the color scheme.

font - a subgroup to specify a specific font type
 name - the name of the font (excluding it's .txf extension)
 size - size of the font
 slant - the slant of the font (in pseudo-pixels)

legend - the text legend to display in the object.

label - the text label to display near the object.

property - the name of the FlightGear property whose value will
  be displayed in the object (and possibly modified through it).

binding - a FlightGear command binding that will be fired when the
  user activates this object (more than one allowed).

keynum - the key code of a key that can be used to trigger the
  widget bindings via keyboard (e.g. <keynum>97</keynum> for
  the "a" key.

key - like "keynum", but takes a character ("a", "A", "Shift-a",
  "Shift-A", "Ctrl-a", "%", etc.), or symbolic key name ("Tab",

```
     "Return" = "Enter", "Esc" = "Escape", "Space", "&amp;" = "and",
     "&lt;", "&gt;", "F1" -- "F12", "Left", "Up", "Right", "Down",
     "PageUp", "PageDn", "Home", "End", "Insert"). Note that you
     can't use "<", ">", and "&" directly.

 default - true if this is the default object for when the user
    presses the [RETURN] key.

 visible - if set to false, hides the whole widget that it is used
    in, along with its children. There's no empty space reserved
    for such widgets. The "visible" property can also be used to hide
    other XML groups from the layouter.
```

Objects may appear nested within the top-level dialog or a "group"
or a "frame" object. Here are all the object types allowed, with their
special properties:

```
dialog
------
```

The top-level dialog box; the name does not actually appear in the
file, since the root element is named PropertyList.

```
  name - (REQUIRED) the unique name of the dialog for use with the
     "dialog-show" command.

  modal - true if the dialog is modal (it blocks the rest of the
     program), false otherwise.  The default is false.

  draggable - false if the dialog is not draggable. The default is true.

  resizable - false if the dialog is not resizable. The default is false.

  nasal - Nasal definition block
     open - Nasal script to be executed on dialog open
     close - Nasal script to be executed on dialog close
```

All Nasal code runs in a dialog namespace. Nasal bindings can
directly access variables and functions defined in an <open> block.

settimer() and setlistener() functions have to be removed manually
in the <close> block if they shouldn't remain active.

Example:

```
<PropertyList>

 <name>sample</name>
 <width>500</width>
 <height>210</height>
 <modal>false</modal>

 <text>
  ...
 </text>

 <button>
  ...
 </button>

</PropertyList>
```


group and frame

A group of subobjects. This object does not draw anything on the
screen, but all of its children specify their coordinates relative to
the group; using groups makes it easy to move parts of a dialog
around.

A frame is a visual representation of a group and has a border and an
adjustable background color.

Example:

```
  <group>
   <x>0</x>
   <y>50</y>
```

```
  <text>
   ...
  </text>

  <input>
   ...
  </input>

  <button>
   ...
  </button>

 </group>
```

input

A simple editable text field.

Example:

```
<input>
 <x>10</x>
 <y>60</y>
 <width>200</width>
 <height>25</height>
 <label>sea-level temperature (degC)</label>
 <property>/environment/temperature-sea-level-degc</property>
</input>
```

text

A non-editable text label.

Example:

```
<text>
 <x>10</x>
 <y>200</y>
 <label>Heading</label>
</text>

<text>
 <x>10</x>
 <y>200</y>
 <label>-9.9999</label> <!-- placeholder for width -->
 <format>%-0.4f m</format>
 <property>/foo/altitude</property>
</text>
```

checkbox

A checkbox, useful for linking to boolean properties.

Example:

```
<checkbox>
 <x>150</x>
 <y>200</y>
 <width>12</width>
 <height>12</height>
 <property>/autopilot/locks/heading</property>
</checkbox>
```

button

A push button, useful for firing command bindings.

  one-shot - true if the button should pop up again after it is
    pushed, false otherwise. The default is true.

```
<button>
 <x>0</x>
 <y>0</y>
 <legend>OK</legend>
 <binding>
  <command>dialog-apply</command>
 </binding>
 <binding>
  <command>dialog-close</command>
 </binding>
 <default>true</default>
</button>
```

combo

A pop-up list of selections.

  value - one of the selections available for the combo. There may be
  any number of "value" fields.

Example:

```
<combo>
 <x>10</x>
 <y>50</y>
 <width>200</width>
 <height>25</height>
 <property>/environment/clouds/layer[0]/type</property>
 <value>clear</value>
 <value>mostly-sunny</value>
 <value>mostly-cloudy</value>
 <value>overcast</value>
 <value>cirrus</value>
</combo>
```

```
list
----


like "combo", but displays all values in a scrollable list box with
slider on the right side. Updates the <property> to the selected
entry. On <dialog-update> re-scans the <value> nodes and updates
the list.




airport-list
------------


like "list", but fills the list automatically with all airports known
to FlightGear. Calls bindings on airport selection and returns the
selected entry in <property> on dialog-apply. Interprets <property>
as search term on dialog-update.




property-list
-------------


like "list", but shows a list of properties from the global property
tree. The widget handles navigation in the property tree. It calls its
bindings on property selection and returns the path of the selected
property in <property> on dialog-apply. It's up to the caller to check
if the path belongs to a dir node or a value node. The widget shows
the contents of the dir property given in <property> on dialog-apply.
It does *not* handle setting of property values! Clicking on some
entries with the "control" or "shift" key pressed has a special meaning:

Ctrl +
  "."     -> toggle verbose mode (shows flags, listeners, dir-values)
  ".."    -> go to root node
  (bool)  -> toggle bool value

Shift +
  "."     -> dump contents of that tree level to the terminal
```

The flags printed after the node type have the following meaning:

```
  r        ->  read protected
  w        ->  write protected
  R        ->  trace read operations    (in the terminal window)
  W        ->  trace write operations
  A        ->  archive bit set
  U        ->  user archive bit set
  P        ->  preserved bit set (value is preserved on sim-reset)
  T        ->  property is "tied"

  Ln       ->  number of listeners attached to this node
```

select

A box with arrow buttons that cycle through a list of values.

Example:

```
  <select>
   <x>10</x>
   <y>50</y>
   <width>200</width>
   <height>25</height>
   <property>/sim/aircraft</property>
   <value>bo105</value>
   <value>ufo</value>
  </select>
```

slider

A horizontal or vertical slider for setting a value.

  vertical - true if the slider should be vertical, false if it should

be horizontal. The default is false.

  min - the minimum value for the slider. The default is 0.0.

  max - the maximum value for the slider. The default is 1.0.

  step - set to non-null if slider should move in steps. The default is 0.0 (off).

  pagestep - set to non-null to enable page-stepping. The default is 0.0 (off).

  fraction - size of the slider handle. Range: 0..1. The default is 0.0 (minimum).

Example:

```
<slider>
 <x>10</x>
 <y>50</y>
 <width>200</width>
 <property>/environment/visibility-m</property>
 <min>5</min>
 <max>50000</max>
</slider>
```

dial

A circular dial for choosing a direction.

  wrap - true if the dial should wrap around, false otherwise. The
    default is true.

  min - the minimum value for the dial. The default is 0.0.

  max - the maximum value for the dial. The default is 1.0.

Example:

```
<dial>
 <x>10</x>
```

105

```
  <y>50</y>
  <width>20</width>
  <property>/environment/wind-from-direction-deg</property>
  <min>0</min>
  <max>360</max>
</dial>
```

textbox

The text will be retrieved/buffered from/within a specified
property tree, like:

```
<textbox>
    <!-- position -->
    <x>100</x>
    <y>100</y>

    <!-- dimensions -->
    <width>200</width>
    <height>400</height>

    <property>/gui/path-to-text-node/contents</property>

    <slider>15</slider> <!--width for slider -->
    <wrap>false</wrap> <!-- don't wrap text; default: true -->
    <top-line>0</top-line <!-- line to show at top, -ve numbers: show last line -->

    <editable>true</editable> <!-- if the puLargeInput is supposed to be editable -->
</textbox>
```

hrule/vrule

Draws a horizontal/vertical line that, by default, expands to full width/height.
Its thickness can be set with <pref-height>/<pref-width>.

```
  <hrule>
```

```
      <color>
        <red>1.0</red>
        <green>0.0</green>
        <blue>0.0</blue>
      </color>
      <pref-height>2</pref-height>
    </hrule>
```

GLOBAL SETTINGS ("THEMES")

FlightGear reads GUI style information from /sim/gui/, which is by default
loaded from file $FG_ROOT/gui/style.xml. This file contains one <font> and
one <colors> group:

global font settings

```
  <sim>
    <gui>
      <font>
        <name type="string">Helvetica.txf</name>
        <size type="float">15</size>
        <slant type="float">0</slant>
      </font>
    </gui>
  <sim>
```

<name> can either be the name of a built-in bitmap font (one of:
"FIXED_8x13", "FIXED_9x15", "TIMES_10", "TIMES_24", "HELVETICA_10",
"HELVETICA_12", "HELVETICA_14", "HELVETICA_18", "SANS_12B"), or the
name of a texture font in the $FG_FONT directory. $FG_FONT is by
default set to $FG_ROOT/Fonts/. Properties <size> and <slant> are
only applied to texture fonts, and otherwise ignored.

global color settings

These define the color of the splash screen font, and the color of the
GUI elements. All colors are in /sim/gui/colors/ and follow the same
pattern:

```
  <sim>
    <gui>
      <colors>
        <!-- splash screen font color; ignores <alpha> value -->
        <splash>
          <red type="float">1.0</red>
          <green type="float">0.9</green>
          <blue type="float">0.0</blue>
        </splash>
      </colors>
    </gui>
  </sim>
```


As listed above, FlightGear implements several GUI elements:

```
(1)    "dialog"       "group"        "frame"     "hrule"      "vrule"
       "list"         "airport-list" "input"     "text"       "checkbox"
       "radio"        "button"       "combo"     "slider"     "dial"
       "textbox"      "select"
```


The underlying plib library uses six colors for each GUI element.
These are:

```
(2)    "background" "foreground"   "highlight"
       "label"      "legend"       "misc"
```


"button", for example, uses the first four colors from (2), while it
ignores "legend" and "misc" color. "text" only uses "label", and ignores
the rest. In some cases the use of colors isn't obvious and you have to

try or look up the plib sources to be sure. GUI colors can be defined
for each of the categories from (1) and (2), and for combinations of
them:

(3)   "button-legend"   "input-misc"   etc.

FlightGear has default colors for (2) built-in. Let's call them (0).
And this is how colors for individual GUI elements are determined,
if, for example, a button is to be drawn:

```
  For the button's background:
  a. read the hard-coded default "background" color from (0) as base
  b. merge the global "background" color from (2) in (if defined)
  c. merge a global color "button-background" from (3) in (if defined)
  d. merge a specific <color> from the dialog's XML file in (if defined)

  Repeat the four steps for the button's "foreground", "highlight",
  etc. color.
```

If you write a style file, you'll most likely start with the colors
from (2):

```
 <sim>
   <gui>
     <colors>
       <background>
         <red type="float">0.6</red>
         <green type="float">0.0</green>
         <blue type="float">0.0</blue>
         <alpha type="float">1.0</alpha>
       </background>

       <foreground>
       ...
```

This makes all dialogs dark red. But you don't, for example, want buttons

to be red, but yellow. So you define another color for buttons:

```
<button>
  <red type="float">1.0</red>
  <green type="float">0.9</green>
  <blue type="float">0.0</blue>
  <alpha type="float">1.0</alpha>
</button>
...
```

This sets all of a button's six colors (2) to some shades of red. plib
does this automatically. The lower and right border ("foreground") will
be darker, the upper and left border will be lighter ("highlight").
If you aren't happy with plib's choice, you can set each of the colors
explicitly. Let's say, we want the text on the button blue (3):

```
<button-legend>
  <red type="float">0.3</red>
  <green type="float">0.3</green>
  <blue type="float">1.0</blue>
  <alpha type="float">1.0</alpha>
</button-legend>
...
```

To set the cursor color from input fields, you'd define "input-misc",
etc.

You can change colors and font at runtime. Just open the property
browser, go to /sim/gui/colors and change whatever you like. The
new color will only take effect, though, if you re-init the GUI.
There's a menu entry for that, and you can define a key binding
for it:

```
<key n="99">
  <name>c</name>
  <desc>Re-init GUI</desc>
  <binding>
    <command>reinit</command>
    <subsystem>gui</subsystem>
```

```
        </binding>
      </key>

  Note that this will currently close all open dialogs!


  __end__
```

15   Hud

```
This document describes the *new* HUD system that will be first released
with fgfs >0.9.10. For the old system see $FG_ROOT/Docs/README.xmlhud.
Note that the old system is scheduled for removal, and that the new system
is work in progress. So it's up to you to choose the lower risk.   :-)




##############################################################################



A HUD configuration file may contain 3 types of information:

  (1) global settings
  (2) HUD instrument definitions
  (3) imports of further HUD config files




(1) global settings ========================================================

These can be used to override settings in the global property tree. Currently
only bool <enbale-3d> is supported. It allows a HUD to define itself if it is
a 2D HUD (false) or a 3D HUD (true). 2D HUDs always remain in the screen plane,
while 3D HUDs always remain in a position relative to the aircraft.
```

Example:

```
<enable-3d>true</enable-3d>
```

(2) HUD instrument definitions ===

These define one single HUD "item" (instrument or label), and consist of several
properties. Some of those are standardized property groups that can be used
in many places. These shall be explained first.

(2.1) standardized property groups --

1. <condition> group
2. input channel group
3. <option>s

(2.1.1) <condition> ....

These define conditions that are either "true" or "false". They are used to
hide/unhide whole items, or to set other item states (blinking on/off) etc.
You find detailed documentation about them in $FG_ROOT/Docs/README.conditions.

(2.1.2) input channel groups ....

These define an input channel to the HUD instrument and serve as interface
between property system and the instrument. A complete channel definition
looks like this (defaults in comments):

```
<input>
```

```
      <property>/position/altitude-agl-ft</property>   <!-- no default -->
      <factor>0.3048</factor>                           <!-- 1.0 -->
      <offset>2.0</offset>                              <!-- 0.0 -->
      <damp>1.5</damp>                                  <!-- 0.0 (no damping) -->
      <min>0.0</min>                                    <!-- -infinity -->
      <max>10000</max>                                  <!-- +infinity -->
  </input>
```

Input channels are only called <input> for instruments that only have one
channel. Other instruments may have two or more channels, called <bank-input>,
<pitch-input> etc. All of them will have the same member properties and behave
the same.

An input channel will preprocess the raw property value for the HUD instrument.
The property may be of any type (bool, int, long, float, double, string), but
not all types will make sense in every situation. The HUD instrument will only
see the final value, which is calculated as:

```
  v = <property> * <factor> + <offset>
  if (<damp>)      v = EWMA_lowpass(v, <damp>)
  if (v < <min>)  v = <min>
  if (v > <max>)  v = <max>
```

The EWMA_lowpass filter (Exponentially Weighted Moving Average) is calculated
like so:

```
  coeff = 1.0 - 1.0 / 10^<damp>
  v = average = (average * coeff) + (v * (1.0 - coeff))
```

That is, a <damp> value of 0 will cause no damping. A damping value of 1 will
make a coefficient of 0.9, which means that the resulting value will be 9/10
of the average plus 1/10 of the new value. A damping value of 2 will make
a coefficient of 0.99 and hence result in a value of 99/100 the average plus
1/100 the new value etc. The higher the <damp> value, the more damped will
the output value be.

```
2.1.3 <option> ...........................................................

Most HUD instruments accept one or more options from a common set. It will be
explaind in the respective intrument descriptions which options are actually
used by that instrument. Possible values are:

  <option> autoticks </option>
  <option> vertical </option>   \___orientation of <tape>
  <option> horizontal </option> /
  <option> top </option>            \
  <option> left </option>           |___place of numbers in <tape>, <gauge>
  <option> bottom </option>         |   top/bottom for turn-bank-indicator, etc.
  <option> right </option>          /
  <option> both </option>            _left/right for vert. and top/bottom for hor.
  <option> noticks </option>
  <option> arithtic </option>
  <option> decitics </option>
  <option> notext </option>        ___no numbers on <tape>


Example:

  <tape>
      <option>left</option>
      <option>vertical</option>
      ...
  </tape>
```

(2.1) properties common to all instruments -----------------------------------

All HUD instruments will accept the following common properties (shown on
a <tape> instrument):

```
<tape>
    <name>foo tape</name>
    <x>-100</x>                          <!-- 0 == center -->
    <y>-60</y>                           <!-- 0 == center -->
    <width>20</width>                    <!-- 0 -->
    <height>120</height>                 <!-- 0 -->
    <condition>...</condition>           <!-- see section 2.1.1; default: true -->
    ...
</tape>
```

The <name> is only a description for the instrument to make reading the config
easier. It's output in --log-level=info, but not otherwise used. The coordinates
define the place and size of the instrument. They are relative to the origin of
their parent, which is the middle of the HUD/screen by default. Positive <x> are
on the right, positive <y> in the upper half. The <condition> hides/reveals the
whole instrument.

(2.2) HUD instruments ---

(2.2.1) <label> ....

Draws a formatted string or number.

Text:
  <format>   ... printf-style format with only one % item. Example: "%2.3lf ft"
  <prefix>   ... prefix text ___ in addition to the <format>
  <postfix> ... postfix text /
  <halign>   ... one of "left", "center" (default), "right".

Box:

```
<box>        ... draw box around label (default: false)
<option>     ... one of (left|right|top|bottom)  ... draw arrow on this side
<pointer-width>  ... size of pointer base
<pointer-lenfth> ... distance of base--peak

<blinking>
    <interval>                     ... on/off-time in seconds (default: -1 == off)
    <condition>...</condition>  ... see secion 2.1.1      (default: true)
</blinking>

TODO:
 <digit>      ... number of insignificant digits (those will be printed smaller)



Example:

 <label>
     <name>G Load</name>
     <x>-40</x>
     <y>25.5</y>
     <width>1</width>
     <height>1</height>

     <input>
         <property>/accelerations/pilot/z-accel-fps_sec</property>
         <factor>-0.03108095</factor>
         <damp>1.3</damp>
     </input>

     <format>%2.1f</format>
     <halign>right</halign>
     <box>true</box>
     <option>bottom</option>    <!-- pointer on the lower edge -->

     <blinking>
         <interval>0.25</interval>
         <condition>
             <or>
                 <less-than>      <!-- G load > 2.0 -->
```

```
                        <property>/accelerations/pilot/z-accel-fps_sec</property>
                        <value>-64.3481</value>
                  </less-than>

                  <greater-than>    <!-- G load < -1.0 -->
                        <property>/accelerations/pilot/z-accel-fps_sec</property>
                        <value>31.17405</value>
                  </greater-than>
            </or>
        </condition>
      </blinking>
  </label>
```

SCALE:
    input
    major-divisions
    minor-divisions
    modulo
    display-span

TAPE:
    tick-bottom
    tick-top
    tick-right
    tick-left
    cap-bottom
    cap-top
    cap-right
    cap-left
    marker-offset
    enable-pointer

```
    zoom

    pointer-type    (moving|fixed)
    tick-type       (circle|line)
    tick-length     (constant|variable)
```

(2.2.3) <dial> ....
SCALE:
    input
    major-divisions
    minor-divisions
    modulo
    display-span

TAPE:
    radius
    divisions

(2.2.4) <gauge> ....
SCALE:
    input
    major-divisions
    minor-divisions
    modulo
    display-span

```
    type        (pitch|climb-dive)
```

(2.2.7) <runway> ....
```
    arrow-scale
    arrow-radius
    line-scale
    scale-dist-nm
    outer_stipple
    center-stipple
    arrow-always
```

```
reads directly:
    /position/altitude-agl-ft,
    /sim/view[0]/config/pitch-pitch-deg
    /sim/view[0]/config/pitch-heading-deg
```

(2.2.8) <aiming-reticle> ....

Draws MIL-STD-1787B aiming reticle. Size of bullet and inner circle are
determined from <width>. The outer circle radius is changeable at runtime.

```
  <active-condition> ... true:  stadiametric (4.2.4.4)  (default)
                         false: standby (4.2.4.5)
  <diameter-input>   ... input channel: diameter of outer circle relative to
                         inner circle; default: 2.0 (= twice as big)
```

```
(3) <import> ================================================================
```

Imports another HUD config into the current one. This can be a file defining
a single instrument ($FG_ROOT/Huds/Instruments/*.xml), a set of instruments
($FG_ROOT/Huds/Sets/*.xml) or a mixture of both (for example a complete HUD
on its own). The x/y offets moves the reference point for the included items
relative to the current reference point.

```
    <import>
        <path>Huds/Sets/controls.xml</path>
        <x-offset>-100</x-offset>
        <y-offset>70</y-offset>
    </import>
```

Imported files can import further files. This is allowed for up to 10 levels.
This is an arbitrary number and can easily be changed in the code if necessary.

When fgfs is called with --log-level=info, then it outputs a graphical trees
of all loaded/imported files, with the instruments shown as leafs.

16 Introduction

```
Internals
---------
```

The core of FlightGear is the property system. This is a tree like internal
representation of global variables. The property system is explained more
in detail later on.

FlightGear' way of doing things is breaking it up into small pieces. There is
(for example) animation code that reacts on property changes. There is also a
Flight Dynamics model (FDM) that (amongst other things) updates properties.
There is a menu system that can display and alter properties. Then we have

sound code that plays sound based on ... properties.

Maybe you see a pattern evolve by now.

All subsystems are almost self containing. Most of the time they only read the values of some properties, and sometimes they alter other properties. This is the basic way of communicating between subsystems.

Property System

The property system is best described as an in-memory LDAP database which holds the state of global variables. The system has a tree like hierarchy (like a file system) and has a root node, sub nodes (like subdirectories) and end-nodes (variables).

All variables are kept internally as raw values and can be converted to any other supported type (boolean, int, float double and string).

Like a file system, every node can be accessed relative to the current node, or absolute to the root node.

The property system also allows aliasing nodes to other nodes (like symbolic linking files or directories to other files or directories) and may be assigned read-only or read-write.

If necessary it would be possible for parts of the program to hold it's own property tree, which is inaccessible from the global property tree, by keeping track of it's own root-node.

Property I/O code allows one to easily read the tree from, or write the tree to an XML file.

Subsystems

To add a new subsystem you would have to create a derived class from SGSubsystem and define at least a small set of functions:

```
class FGFX : public SGSubsystem
{
public:

  FGFX ();
  virtual ~FGFX ();

  virtual void init ();
  virtual void reinit ();
  virtual void bind ();
  virtual void unbind ();
  virtual void update (double dt);
}
```

The init() functions should make sure everything is set and ready so the
update() function can be run by the main loop. The reinit() function handles
everything in case of a reset by the user.

The bind() and unbind() functions can be used to tie and untie properties.

After that you can register this class at the subsystem manager:

```
globals->add_subsystem("fx", new FGFX);
```

Now the subsystem manager calls the update() function of this class every
frame. dt is the time (in seconds) elapsed since the last call.

Scripting

The scripting langage Nasal can also read and modify properties but it can also
be incorporated into the menu system. The documentation for Nasal can be found
here: http://www.plausible.org/nasal/flightgear.html

17 IO

This document describes how to invoke FlightGear's generic IO subsystem.

FlightGear has a fairly flexible generic IO subsystem that allows you to "speak" any supported protocol over any supported medium. The IO options are configured at runtime via command line options. You can specify multiple entries if you like, one per command line option.

The general form of the command line option is as follows:

```
--protocol=medium,direction,hz,medium_options,...

protocol = { native, nmea, garmin, fgfs, rul, pve, ray, etc. }
medium = { serial, socket, file, etc. }
direction = { in, out, bi }
hz = number of times to process channel per second (floating
     point values are ok.
```

Generic Communication:

```
--generic=params
```

With this option it is possible to output a pre-configured ASCII string or binary sequence using a predefined separator. The configuration is defined in an XML file located in the Protocol directory of the base package.

```
params can be:
serial port communication:     serial,dir,hz,device,baud,protocol
socket communication:          socket,dir,hz,machine,port,style,protocol
i/o to a file:                 file,dir,hz,filename,protocol
```

See README.protocol for how to define a generic protocol.

Serial Port Communication:

```
--nmea=serial,dir,hz,device,baud

device = OS device name of serial line to be open()'ed
baud = {300, 1200, 2400, ..., 230400}

example to pretend we are a real gps and output to a moving map application:

--nmea=serial,out,0.5,COM1,4800

Note that for unix variants you might use a device name like "/dev/ttyS0"


Socket Communication:

--native=socket,dir,hz,machine,port,style

machine = machine name or ip address if client (leave empty if server)
port = port, leave empty to let system choose
style = tcp or udp

example to slave one copy of fgfs to another

fgfs1:  --native=socket,out,30,fgfs2,5500,udp
fgfs2:  --native=socket,in,30,,5500,udp --fdm=external

This instructs the first copy of fgfs to send UDP packets in the
native format to a machine called fgfs2 on port 5500.

The second copy of fgfs will accept UDP packets (from anywhere) on
port 5500.  Note the additional --fdm=external option.  This tells
the second copy of fgfs to not run the normal flight model, but
instead set the FDM values based on an external source (the
network in this case.)


File I/O:

--garmin=file,dir,hz,filename

filename = file system file name
```

example to record a flight path at 10 hz:

--native=file,out,10,flight1.fgfs

example to replay your flight

--native=file,in,10,flight1.fgfs --fdm=external

You can make the replay from a file loop back to the beginning
when it reaches the end of the file with the "repeat" flag:

--generic=file,in,20,flight.out,playback,repeat

With a numeric argument, FlightGear will exit after that number of repeats.
--generic=file,in,20,flight.out,playback,repeat,5

Moving Map Example:

Per Liedman has developed a moving map program called Atlas
(atlas.sourceforge.net) The initial inspiration and much code came
from Alexei Novikov.

The moving map supports NMEA format input either via network or
via serial port. Either way will work, but this example
demonstrates the use of a socket connection.

Start up fgfs with:

fgfs --nmea=socket,out,0.5,atas-host-name,5500,udp

Start up the Atlas program with:

Atlas --udp=5500 --fgroot=path-to-fg-root --glutfonts

Once both programs are running, the Atlas program should display
your current location. Atlas is a really nifty program with many
neat options such as the ability to generate and use background
bitmaps that show the terrain, cities, lakes, oceans, rivers, etc.

```
HTTP Server Example

    You can now interact with a running copy of FlightGear using your
    web browser.  You can view all the key internal variables and even
    change the ones that are writable.  If you have support in your
    favorite [scripting] language for interacting with an http server,
    you should be able to use this as a mechanism to interface your
    script with FlightGear.

    Start up fgfs with the --httpd=<port#> option:

    For example:

        fgfs --httpd=5500

    Now point your web browser to:

        http://host.domain.name:5500/

    When a value is displayed, you can click on it to bring up a form
    to assign it a new value.


ACMS flight data recorder playback

  fgfs --fdm=acms --generic=file,in,1,<path_to_replay_file>,acms
```

18   Joystick

```
Replaced by Docs/README.Joystick.html in the base package.
```

19   JSBsim

```
JSBSim
```

JSBSim is an ongoing attempt at producing an OO Flight Dynamics Model (FDM) to replace LaRCsim as the default FDM for FlightGear. It can also be used standalone.

JSBSim uses config files to represent aircraft, engines, propellers, etc. Also, the flight control system is described in the config file. Normally, for use with FlightGear, the config files are named this way [case is significant]:

<FG_ROOT>/Aircraft/<aircraft name>/<aircraft name>.xml

Engines are named like this:

<FG_ROOT>/Engines/<engine name>.xml

Aircraft and engine config files are present in the FGFS Base package which must be downloaded. See the FlightGear web site for more information.

How to run FGFS using JSBSim

All the various FDMs are currently compiled into FGFS. You can specify which FDM you want at run time. You can also specify which aircraft you want. Currently, for JSBSim only the X-15 and C-172 aircraft are available. Here is an example command line used to start up FlightGear using JSBSim as the FDM:

fgfs --fdm=jsb --aircraft=X15 --units-feet --altitude=60000 --uBody=2000 --wBody=120

or,

fgfs --fdm=jsb --aircraft=c172

[Note: uBody is the forward velocity of the aircraft, wBody is the downward velocity - from the aircraft point of view. This essentially means that the aircraft is going forward fast and has an angle of attack of about 4 degrees or so]

The first command line sets up the initial velocity and altitude to allow the X15 to glide down. Note that if you fire up the engine, it

will burn for only about two minutes and then run out of fuel – but
you will go very, very fast! The second command line example will
start up the C172 on the end of the runway.

Check out the JSBSim home page at http://jsbsim.sf.net. Please report
any bugs to jsb@hal-pc.org, or apeden@earthlink.net, or post on the
jsbsim web site using the SourceForge bug tracking system for the
project.

JSBSim is written by Jon S. Berndt and Tony Peden with contributions
by other FlightGear programmers, as well.

20   Jsclient

```
Start flightgear with
fgfs --jsclient=socket,in,<hz>,,<port>,udp \
     --prop:/jsclient/axis[i]="/property/you/want/to/control" \
     --prop:/jsclient/axis[i+1]="/another/property/you/want/to/control" ...
eg:
# fgfs --aircraft=yf23-yasim --airport=KEMT --jsclient=socket,in,5,,16759,udp \
  --prop:/jsclient/axis[0]="/controls/flight/spoilers" \
  --prop:/jsclient/axis[1]="/radios/comm/volume"

Start the server on the machine with the remote gameport:
JsServer <host> <port>
eg:
# JsServer 192.168.1.1 16759

(JsServer can be started before or after fgfs)
```

21   Layout

I just commited an implementation of GUI layout management, ported
over from my game project last year*. What this means is that you no
longer need to position your widgets manually in dialogs, and can
instead lay them out in tables and boxes like the pros do. :) I've
redone a few of the dialogs using the new scheme (I'm especially proud
of the autopilot dialog: http://plausible.org/andy/autopilot-new.png),

so you can see what the possibilities look like.

* FWIW, this is almost the last of my useful code from last spring.
  Nasal and the Plib vertex splitting code are two other bits that
  were useful in isolation. I also had a terrain engine and stencil
  shadow implementation, but those weren't really production quality.

Basically, the implementation is a preprocessor on top of the existing
dialog properties, which sets x/y/width/height values based on
constraints. The <group> objects, including the top-level one which
represents the whole dialog, can now have a <layout> property, which
can be "hbox", "vbox", or "table".

The boxes simply lay out their children in order, either top-to-bottom
or left-to-right. The box name comes from Qt and Gtk, but this is
also the same thing that Java calls a "flow layout", or what the Tk
"packer" does. You can set "constraint" properties on the children,
to give the layout manager hints as to how to place the children. For
the boxes, these are:

```
  equal:   The box manager makes sure that all the widgets with this
           constraint set to true get equal sizes big enough to fit the
           largest one.  This is very useful for button boxes to make
           the "OK" and "Cancel" buttons match, for example.
  stretch: Cells with "stretch" set to true get all the extra space,
           if any, the box has to allocate.  These are useful for
           alignment purposes, especially when combined with <empty>
           "widgets" (which are ignored by the dialog creation code,
           but honored by the layout engine).
```

The table layout will be a little more familiar to anyone with HTML
experience. Children of tables get the following constraints:

```
  row:     The row number containing the upper left corner of the widget.
           Table rows are zero-indexed.
  col:     The column number containing the upper left corner of the widget.
           Table columns are zero-indexed.
  rowspan: The number of rows spanned by the widget. Defaults to one.
  colspan: The number of columns spanned by the widget.  Defaults to
           one.
```

130

Inside of each "cell", regardless of parent layout, there are some
constraints that are used to position the widget within the space
available:

```
  halign:      The horizontal alignment.  Can be "left", "right",
               "center", or "fill" (i.e. stretch to available space).
  valign:      The vertical alignment.  Can be "top", "bottom",
               "center", or "fill".
  padding:     The number of pixels to leave between the edge of the
               cell and the widget.
  pref-height:
  pref-width:  Overrides the default preferred size of the widget.
               Note that this is the size of the widget only, not the
               cell (which includes padding).
```

Also, the padding values for cells in a group can be set to a default
value with a <default-padding> property on the group widget.

Some will ask why didn't I implement this as part of Pui. The problem
is the pui just isn't set up for it. Not only is there no notion of
"preferred" size for a widget, there isn't anything remote like a
"constraint" system for attaching arbitrary values to widgets. With
the property system, I have that for free (the original code was
written to work with Nasal objects, btw). I can do the layout with
the properties and on the properties, and our existing dialog code
hardly needs to change at all.

Anyway, give it a try and see if I've broken anything. Also, note
that some of these changes *do* modify the visual appearance of the
GUI. I think it looks better, but opinions will no doubt vary. Shout
if you hate it.

And finally, the text alignment doesn't quite look right with current
plib due to some minor rendering bugs. Bug Steve to apply the patch I
submitted a week or so ago. :)

Andy

22   Logging

```
Logging in FlightGear
---------------------
```

[Note: JSBSim also has its own independent logging facilities, which are not discussed here.]

FlightGear can log any property values at any interval to one or more CSV files (which can be read and graphed using spreadsheets like Gnumeric or Excel). Logging is defined in the '/logging' subbranch of the main property tree; under '/logging', each '/log' subbranch defines a separate log with its own output file and interval. Here is a simple example that logs the rudder and aileron settings every second (1000ms) to the file steering.csv, using a comma (the default, anyway) as the field delimiter:

```
<PropertyList>
 <logging>
  <log>
   <enabled>true</enabled>
   <filename>steering.csv</filename>
   <interval-ms>1000</interval-ms>
   <delimiter>,</delimiter>
   <entry>
    <enabled>true</enabled>
    <title>Rudder</title>
    <property>/controls/flight/rudder</property>
   </entry>
   <entry>
    <enabled>true</enabled>
    <title>Ailerons</title>
    <property>/controls/flight/aileron</property>
   </entry>
  </log>
 </logging>
</PropertyList>
```

Each 'log' subbranch contains a required 'enabled' property, an optional 'filename' property (defaults to "fg_log.csv"), an optional

'delimiter' property (defaults to a comma), an optional 'interval-ms'
property (defaults to 0, which logs every frame), and a series of
'entry' subbranches. The 'delimiter' property uses only the first
character of the property value as the delimiter. Note that the
logger does no escaping, so you must choose a delimiter that will not
appear in the property values (that's not hard, since most of the
values are numeric, but watch for commas in the titles).

Each 'entry' subbranch contains a required 'enabled' property, a
'property' property specifying the name of the property to be logged,
and an optional 'title' property specifying the title to use in the
CSV file (defaults to the full path of the property). The elapsed
time in milliseconds since the start of the simulation is always
included as the first entry with the title "Time", so there is no need
to include it explicitly.

Here's a sample of the logging output for the above log:

```
Time,Rudder,Ailerons
6522,0.000000,0.000000
7668,-0.000000,0.000000
8702,-0.000000,0.000000
9705,-0.000000,0.000000
10784,-0.000000,0.000000
11792,-0.000000,0.000000
12808,-0.000000,-0.210000
13826,-0.000000,-0.344000
14881,-0.000000,-0.066000
15901,-0.000000,-0.806000
16943,-0.000000,-0.936000
17965,-0.000000,-0.534000
19013,-0.000000,-0.294000
20044,-0.000000,0.270000
21090,-0.000000,-1.000000
22097,-0.000000,-0.168000
```

Note that the requested interval is only a minimum; most of the time,
the actual interval is slightly longer than the requested one.

The easiest way for an end-user to define logs is to put the log in a

separate XML file (usually under the user's home directory), then
refer to it using the --config option, like this:

```
  fgfs --config=log-config.xml
```

The output log files are always relative to the current directory.

--

David Megginson, last updated 2002-02-01

23 Materials

README.materials

This README describes the materials.xml file format. It is targeted at those
wanting to change the appearance of the scenery in FlightGear.

As is the norm in FG, the materials.xml file is a properties file. However,
it is only read on startup, is not part of the main property tree and cannot
be changed at runtime.

The properties file consists of a number of <material> entries, each of which
describes a single visually distinct terrain material in the FG world.

The rest of this document describes the children tags of the <material> entry.

name : Scenery type names that map to this material. These are typically taken
      from landclass definitions created by TerraGear. Multiple scenery types
      may map to a single material. This is recommended to minimize texture
      memory usage.

condition : A condition statement used to activate the material. Note that this
      if evaluated once at start-up.

texture : A relative path to an SGI RGB, PNG or DDS file containing a texture
      for the material. RGB and PNG are recommended for platform compatibility.
      You may define more than one <texture> element, in which case the scenery
      loader will choose one texture for each contiguous set of scenery

triangles.

texture-set : If using an effect (see below), it may be necessary to define more
        than one texture. The texture-set element has multiple <texture> element
        children which may then be referenced by the effect. You may define more
        than one <texture-set> element, in which case the scenery loader will
        choose one texture for each contiguous set of scenery triangles.

object-mask : An optional bitmap file used to control random placement of lights,
        buildings and vegetation on the terrain. The green channel mask is used
        for random vegetation placement, the blue channel for buildings and lights.
        and the red channel controls the rotation of buildings (0.0 is North, 0.5
        is South). Fractional colour values can be used to give a probability of
        placement. Multiple object-masks may be defined to match up with <texture>
        or <texture-set> elements.

effect : The effect to be used for this material. (default:
        Effects/terrain-default)

        ambient, diffuse, specular, emissive, and shininess are copied into the
        parameter section of the effect created for this material.

parameters : Additional parameters to be used in the effect. See README.effects
            for format information.

wrapu : True if the texture should repeat horizontally over a surface, false if
        it should not repeat (default: true).

wrapv : True if the texture should repeat vertically over a surface, false if
        it should not repeat (default: true).

mipmap : True if the texture should be mipmapped, false otherwise. (default: true).

xsize : The horizontal size of a single texture repetition, in meters.

ysize : the vertical size of a single texture repetition, in meters

light-coverage : The coverage of a single point of light in m^2. 0 indicates no
            lights at all. Minimum value is 1000m^2. May be masked by the blue channel
            of an object-mask. Lights are all generated 3m above the surface, and

135

have random colour (50% yellow, 35% white, 10% orange, 5% red)

ambient : The ambient light colour for the material, specified as separate
        r, g, b, a components (default: all color components 0.2, alpha 1.0).

diffuse : The diffuse light colour for the material, specified as separate
        r, g, b, a components (default: all color components 0.8, alpha 1.0).

specular : The specular light colour for the material, specified as separate
        r, g, b, a components (default: all color components 0.0, alpha 1.0).

emissive : The emissive light colour for the material, specified as separate
        r, g, b, a components (default: all color components 0.0, alpha 1.0).

solid : Whether the surface is solid from an FDM perspective. If it is not
        solid, it is assumed that the material models a fluid (water) surface.
        (default: true).

friction-factor : The friction factor for that material. The normalized
        factor can be used by a FDM to post-multiply all contact friction forces
        with that factor. That is the more slippery a material is the smaller
        this value should be. (default: 1.0 for Dry concrete/Asphalt).

rolling-friction : the gear rolling rolling-friction coefficient for this
        particular material. (default: 0.02 for Dry concrete/Asphalt).

bumpiness : normalized bumpiness factor for this particular terrain.
        (default: 0.0 for a smooth surface).

load-resistance : a pressure value how much force per surface area this
        surface can carry without deformation. The value should be in N/m^2
        (default: 1e30).

glyph : group that defines one letter/digit/symbol in a font texture
        sub-entries: name, left (default: 0.0), right (default: 1.0)
        (left and right describe the horizontal position in the texture.)

wood-coverage : The coverage of trees in areas marked as woodland in
        m^2. A lower number means a higher density of trees. A value of
        0 indicates no woods. May be masked by the green channel of an

object-mask. (default: 0)

tree-range-m : The range at which trees become visible. Note that this
        is not absolute, as trees are loaded in blocks. A lower number means
        trees will not become visible until you are closer.

tree-texture : A texture to use for the trees. Typically this will contain around
        8 different trees in a row, duplicate 4 times. From bottom to top, the
        rows contain
        * summer textures
        * summer snow texture
        * winter texture
        * winter snow texture

        Each tree must have space at the top. For a 512x512 texture sheet, this
        should be 8 pixels. Otherwise subsequent rendering results in "top hats"
        above trees in the distance where the trunk of the tree above in the
        textures sheet bleeds downwards when the mipmaps are generated.

tree-varieties : The number of different trees defined in the tree-texture
        horizontally. (default: 1)

tree-height-m : The average height of the trees. Actual tree height will
        vary by +/- 50%. (default: 0)

tree-width-m : The average width of the tree cover. Actual tree width will
        vary by +/- 50%. (default 0)

tree-max-density-angle-deg : The slope angle at which trees begin to thin out
        as the slope is too steep to support the full coverage. Shallower
        slopes have maximum wood-coverage. Steeper slopes have fewer trees.
        (default : 45)

tree-zero-density-angle-deg : The angle at which the slope is too steep to
        support significant vegetation. Steeper slopes have no trees.
        (default : 60)

object-max-density-angle-deg : The angle at which objects and buildings become
        less dense due to a steep slope. (default : 20)

```
object-zero-density-angle-deg : The angle at which the slope is too steep to build
          on.  No object/buildings will be placed on slopes steeper than this.
          (default : 30)

object-group : A group of random objects to be placed on the surface. Contains
          <range-m> and one or more <object> children.

range-m : The distance at which objects within this object-group become
          visible. Note that for realism, 60% of the objects will become visible
          at <range-m>, 30% at 1.5*<range-m>, and 10% at 2*<range-m>.
          (default: 2000)

object : A set of random objects to be placed. Contains <coverage-m2>, <path>
          and <heading> children.

coverage-m2 : The coverage of a single object in m2. Lower values mean a higher
          density. Minimum value is 1000.

spacing-m : The minimum space between this object and any other on the surface in
          meters. This helps to avoid objects being placed ontop of each other.
          (default 20)

path :    Path relative to FG_ROOT to a model definition, usually .ac or .xml file.
          More than one <path> may be included within the <object> tag, in which
          case a single <path> is chosen at random for each individual object
          placement.

heading-type : Indicator of how the heading of the random objects should be
          determined. Valid values are:
             fixed     - Objects all point North. Default.
             random    - Objects are assigned an individual random heading
             mask      - Rotation is taken from the red channel of the object-mask
             billboard - Object is always rotated to face camera - expensive


Random Buildings
================

Random Buildings come in three sizes, with individual constraints.
```

138

Small buildings. These have different textures on the sides compared to the front
and back. Small buildings are never deeper than they are wide.

Medium buildings, which are never taller than they are wide.

Large buildings. There are no constraints on their width, depth or height.

building-coverage : The coverage of random buildings in areas marked for random
          objects in m^2. A lower number means a higher density of buildings. A
          value of 0 indicates no buildings. May be masked by the blue channel of an
          object-mask. (default: 0)

building-spacing-m : The minimum spacing between random buildings and other buildings
          or random objects. This helps avoid objects being placed on top of each
          other. (default: 5)

building-small-ratio: Ratio of small buildings. These buildings are 1-3 stories
          in height, and may have a pitched roof. Fraction of small buildings is
          (<building-ratio-small> / (<building-ratio-small> + <building-ratio-medium>
          + <building-ratio-large>). (default: 0.8)

building-medium-ratio : Ratio of medium buildings. These buildings are 3-6 stories
          in height, and have a flat roof. (default: 0.15)

building-large-ratio : Ratio of large buildings. These buildings are 5-10 stories in
          height, and have a flat roof. (default 0.05)

building-small-pitch : Fraction of small buildings with pitched roofs. (default 0.8)
building-medium-pitch : Fraction of small buildings with pitched roofs. (default 0.2)
building-large-pitch : Fraction of small buildings with pitched roofs. (default 0.1)

building-small-min-floors : Min. number of floors for a small building. (default 1)
building-small-max-floors : Max. number of floors for a small building. (default 3)

building-medium-min-floors : Min. number of floors for a medium building. (default 3)
building-medium-max-floors : Max. number of floors for a medium building. (default 8)

building-large-min-floors : Min. number of floors for a medium building. (default 5)
building-large-max-floors : Max. number of floors for a medium building. (default 20)

```
building-small-min-width-m : Min. width of small buildings. (default 15)
building-small-max-width-m : Max. width of small buildings. (default 60)
building-small-min-depth-m : Min. depth of small buildings. (default 10)
building-small-max-depth-m : Max. depth of small buildings. (default 20)

building-medium-min-width-m : Min. width of medium buildings. (default 25)
building-medium-max-width-m : Max. width of medium buildings. (default 50)
building-medium-min-depth-m : Min. depth of medium buildings. (default 20)
building-medium-max-depth-m : Max. depth of medium buildings. (default 50)

building-large-min-width-m : Min. width of large buildings. (default 50)
building-large-max-width-m : Max. width of large buildings. (default 75)
building-large-min-depth-m : Min. depth of large buildings. (default 50)
building-large-max-depth-m : Max. depth of large buildings. (default 75)

building-texture : The texture used for all buildings. See Docs/buildings.png for
                   details. (default Texture/buildings.png)

building-lightmap: Emissive texture for all buildings, which is faded in at night to
                   provide illusion of lit windows.  Same texture coordinates and
                   format at building-texture above.

building-range-m: Range at which all buildings are visible.  Beyond this point fewer
                   and fewer buildings are rendered, with no buildings rendered at
                   2*building-range-m (default 10000)
```

24   Mingw

```
How to compile FlightGear with mingw
====================================


MinGW & MSYS
============

You need to install mingw & msys:

http://www.mingw.org
```

You need at least:

MinGW: binutils, gcc-core, gcc-g++, mingw-runtime, mingw-utils, w32api
I would recommed the gcc-3.4.4 versions.
MSYS: msys-1.0.10.exe, msys-autoconf, msys-automake, msys-libtool, msys-DTK.

Please read instructions carefully.

Set the follwing environment variables within msys shell.

```
export CFLAGS="-I/usr/local/include -O2"
export CXXFLAGS="-I/usr/local/include -O2"
export CPPFLAGS=-I/usr/local/include
export LDFLAGS=-L/usr/local/lib
```

Pthread-win32
=============

http://sources.redhat.com/pthreads-win32/

compile:
make GCE-inlined

Install:
cp pthread.h sched.h semaphore.h /usr/local/include
cp linpthreadGCE2.a /usr/local/lib/libpthread.a
cp pthread-GCE.dll /usr/local/bin

patch header:

```
--- pthread.h   Sat Oct  1 20:56:43 2005
**************
*** 210,218 ****
  * ----------------
  */

! #if HAVE_CONFIG_H
! #include "config.h"
! #endif /* HAVE_CONFIG_H */
```

```
  #ifndef NEED_FTIME
  #include <time.h>
--- 210,218 ----
   * ----------------
   */


! //#if HAVE_CONFIG_H
! //#include "config.h"
! //#endif /* HAVE_CONFIG_H */

  #ifndef NEED_FTIME
  #include <time.h>
```

GLUT
====

use precompiled in order to avoid conflicts with glut32.dll already installed.

http://www.xmission.com/~nate/glut.html
http://www.xmission.com/~nate/glut/glut-3.7.6-bin.zip

The header has to be updated with respect to MINGW.

```
*** glut.h      Tue Dec 12 22:22:52 2000
--- /local_old/include/GL/glut.h        Thu Aug 18 20:41:15 2005
**************
*** 20,26 ****
    /* XXX This is from Win32's <windef.h> */
  #  ifndef APIENTRY
  #   define GLUT_APIENTRY_DEFINED
! #   if (_MSC_VER >= 800) || defined(_STDCALL_SUPPORTED) || defined(__BORLANDC__) |
  #    define APIENTRY    __stdcall
  #   else
  #    define APIENTRY
--- 20,26 ----
    /* XXX This is from Win32's <windef.h> */
  #  ifndef APIENTRY
  #   define GLUT_APIENTRY_DEFINED
! #   if (_MSC_VER >= 800) || defined(_STDCALL_SUPPORTED) || defined(__BORLANDC__) |
  #    define APIENTRY    __stdcall
```

```
  #   else
  #     define APIENTRY


install:
cp glut.h /usr/local/include
cp glut32.dll /usr/local/bin

reimp glut32.lib
cp libglut32.a /usr/local/lib

OpenAL
======


Get OpenAL for instance from Creative

OpenAL win32 package
install Redist


install:

cd libs
reimp  OpenAL32.lib
cp libopenal32.a  /usr/local/lib
cp alut.lib  /usr/local/lib/libalut.a
cd ..
mkdir /usr/local/include/AL
cp Include/* /usr/local/include/AL


zlib-1.2.3
==========


configure --prefix=/usr/local
make
make install

plib-1.6.8
==========
```

```
configure --prefix=/usr/local
make
make install

simgear
=======
get simgear from CVS
configure --prefix=/usr/local
make
make  install

flightgear
==========
configure --prefix=/usr/local --with-threads
make
make install
```

25   Minipanel

```
Mini Panels for c172

List of files:

./keyboard.xml - same as release key bindings with "s" added to swap panels.
./preferences.xml - same as release preferences.xml with "panel2" added.
./Aircraft/c172/Panels/c172-panel-mini.xml - mini with sacred six, compass,
        mixture knob, flaps, and control indicators.
./Aircraft/c172/Panels/c172-panel-trans-mini.xml - same mini panel with
        plexiglass (transparent) background.
./Aircraft/c172/Panels/Textures/panel-mini-bg.rgb - grey background.
./Aircraft/c172/Panels/Textures/panel-trans-mini-bg.rgb - transparent background.


USAGE NOTES:

Hitting "s" will switch between the standard panel and the default.

You may choose other panels for the two that get toggled by changing the
```

preferences or adding command line parameters.

The property for the panels are:
Normal (Primary) panel: /sim/panel/path=<path to xml file>
Mini (Secondary) panel: /sim/panel2/path=<path to xml file>

The new property is /sim/panel2/path, it does not need to be a mini panel, you
can use whatever you want.

For example: to use the grey mini panel change preferences.xml or add this to
your command line:

runfgfs --prop:/sim/panel2/path=Aircraft/c172/Panels/c172-panel-mini.xml

26   Multiplayer

The commands are of the form:

--multiplay=in | out,Hz,destination address,destination port
--callsign=a_unique_name

Below are some examples of startup commands that demonstrate the use of the
multiplayer facilities.

For two players on a local network or across the internet:

Player1:
--multiplay=out,10,192.168.0.3,5500 --multiplay=in,10,192.168.0.2,5501
--callsign=player1

Player2:
--multiplay=out,10,192.168.0.2,5501 --multiplay=in,10,192.168.0.3,5500
--callsign=player2

For multiple players on a local network:

Player1:

```
--multiplay=out,10,255.255.255.255,5500
--multiplay=in,10,255.255.255.255,5500 --callsign=player1

Playern:
--multiplay=out,10,255.255.255.255,5500
--multiplay=in,10,255.255.255.255,5500 --callsign=playern
```

Note that the callsign is used to identify each player in a multiplayer game
so the callsigns must be unique. The multiplayer code ignores packets that
are sent back to itself, as would occur with broadcasting when the rx and tx
ports are the same.

Multiple players sending to a single player:

Player1:
```
--multiplay=out,10,192.168.0.2,5500 --callsign=player1
```

Player2:
```
--multiplay=out,10,192.168.0.2,5500 --callsign=player2
```

Player3:
```
--multiplay=out,10,192.168.0.2,5500 --callsign=player3
```

Player4 (rx only):
```
--multiplay=in,10,192.168.0.2,5500 --callsign=player4
```

This demonstrates that it is possible to have multiple instances of
Flightgear that send to a single instance that displays all the traffic. This
is the sort of implementation that we are considering for use as a tower
visual simulator.

For use with a server:

Oliver Schroeder has created a server for multiplayer flightgear use.
The server acts as a packet forwarding mechanism. When it
receives a packet, it sends it to all other active players
in the vicinity (the server is configured to use 100nm by default).

Check out the server homepage <http://www.o-schroeder.de/fg_server/>
for the current status. You can either download the server for
some local use, or join the developers flying at the existing servers.
As with flightgear, the server is free software, released under GPL.

Pigeon <http://pigeond.net> has created a web page monitoring
two such servers, showing the traffic in a Google map environment.
See <http://pigeond.net/flightgear/fg_server_map.html>.

Options needed to enable multiplayer game with a server:
Player1:
--multiplay=out,10,serveraddress,5000 --multiplay=in,10,myaddress,5000
--callsign=player1

Player2:
--multiplay=out,10,serveraddress,5000 --multiplay=in,10,myaddress,5000
--callsign=player2

...

PlayerN:
--multiplay=out,10,serveraddress,5000 --multiplay=in,10,myaddress,5000
--callsign=playerN

Note that if every player using a particular server, such as one of those
listed on the Pigeon's page, needs to have a unique callsign, not
already in use on that server.

If you are sitting behind a NAT'ting firewall, then you need to forward
the incoming traffic on the firewall outer (visible to the internet)
address arriving at the UDP port you use (5000 in the case above)
over to your private LAN address. In this case, use your PRIVATE LAN address
as <myaddress>. Example (if your private LAN address is 10.0.0.1,
in order to play on pigeond.net):

fgfs --multiplay=in,10,10.0.0.1,5000 --multiplay=out,10,pigeond.net,5000
        --callsign=...UNIQUE callsign here...

If you and the server are in the same address space (i.e., both have a public
IP address or both are on the same private LAN), you hopefully don't need to

mess with any firewalls.

If you don't see other players playing on the same server in your flightgear, check that you have followed the above router configuration guidelines. Check that you don't have any LOCAL firewall running on your computer preventing the flightgear network traffic flow.

Finally, use ethereal(1) or tethereal(1) to capture the UDP traffic on the port that you are using, and see if you observe both incoming and outgoing packets.

It's a good idea to talk to the IRC channel #flightgear on irc.flightgear.org while flying on one of the public servers. Also, it makes sense for every user on the same server to use the same weather setup, e.g., the real weather METAR feed, selected by setting to true the real-world-weather-fetch and control-fdm-atmosphere properties.

Further reading (a must if you have a problem):
----
[1] The flightgear server homepage <http://fgms.sourceforge.net/>
[2] The wiki howto <http://wiki.flightgear.org/index.php/Howto:_Multiplayer>
[3] If everything else fails, ask for help on
the IRC channel #flightgear on irc.flightgear.org

27 Multiscreen

The Open Scene Graph library, which current FlightGear uses for its 3D graphics, provides excellent support for multiple views of a scene. FlightGear uses the osgViewer::Viewer class, which implements a "master" camera with "slave" cameras that are offset from the master's position and orientation. FlightGear provides the "camera group" abstraction which allows the configuration of slave cameras via the property tree.

Slave cameras can be mapped to windows that are open on different screens, or all in one window, or a combination of those two schemes, according to the video hardware capabilities of a machine. It is not advisable to open more than one window on a single graphics card due to the added cost of OpenGL context switching between the windows. Usually, multiple monitors attached to a single graphics card

are mapped to different pieces of the same desktop, so a window can be opened that spans all the monitors. This is implemented by Nvidia's TwinView technology and the Matrox TripleHead2Go hardware.

The camera group is configured by the /sim/rendering/camera-group node in the property tree. It can be set up by, among other things, XML in preferences.xml or in an XML file specified on the command line with the --config option.

Here are the XML tags for defining camera groups.

camera-group
For the moment there can be only one camera group. It can contain window, camera, or gui tags.

 window
 A window defines a graphics window. It can be at the camera-group
 level or defined within a camera. The window contains these tags:

  name - string
  The name of the window which might be displayed in the window's
  title bar. It is also used to refer to a previously defined
  window. A window can contain just a name node, in which case
  the whole window definition refers to a previously defined window.

  host-name - string
  The name of the host on which the window is opened. Usually this is
  empty.

  display - int
  The display number on which the window is opened.

  screen - int
  The screen number on which the window is opened.

  x, y - int
  The location on the screen at which the window is opened. This is in
  the window system coordinates, which usually puts 0,0 at the upper
  left of the screen XXX check this for Windows.

width, height - int
The dimensions of the window.

decoration - bool
Whether the window manager should decorate the window.

fullscreen - bool
Shorthand for a window that occupies the entire screen with no
decoration.

overrideRedirect - bool
Only effective when fullscreen = true.
Provides an extra hint for Gnome-based linux systems that we insist that
the full screen window span *all* physical displays, not just the current
physical display.

camera
The camera node contains viewing parameters.

window
This specifies the window which displays the camera. Either it
contains just a name that refers to a previous window definition, or
it is a full window definition.

viewport
The viewport positions a camera within a window. It is most useful
when several cameras share a window.

x, y - int
The position of the lower left corner of the viewport, in y-up
coordinates.

width, height - int
The dimensions of the viewport

view
The view node specifies the origin and direction of the camera in
relation to the whole camera group. The coordinate system is +y up,
-z forward in the direction of the camera group view. This is the
same as the OpenGL viewing coordinates.

```
 x,y,z - double
 Coordinates of the view origin.

 heading-deg, pitch-deg, roll-deg - double
 Orientation of the view in degrees. These are specified using the
 right-hand rule, so a positive heading turns the view to the left,
 a positive roll rolls the view to the left.

perspective
This node is one way of specifying the viewing volume camera
parameters. It corresponds to the OpenGL gluPerspective function.

 fovy-deg - double
 The vertical field-of-view

 aspect-ratio - double
 Aspect ratio of camera rectangle (not the ratio between the
 vertical and horizontal fields of view).

 near, far - double
 The near and far planes, in meters from the camera eye point. Note
 that FlightGear assumes that the far plane is far away, currently
 120km. The far plane specified here will be respected, but the sky
 and other background elements may not be drawn if the view plane is
 closer than 120km.

 offset-x, offset-y - double
 Offsets of the viewing volume specified by the other parameters in
 the near plane, in meters.

frustum
This specifies the perspective viewing volume using values for the near
and far planes and coordinates of the viewing rectangle in the near
plane.

 left, bottom - double
 right, top - double
 The coordinates of the viewing rectangle.
```

```
 near, far - double
  The near and far planes, in meters from the camera eye point.

 ortho
  This specifies an orthographic view. The parameters are the sames as
  the frustum node's.

 gui
 This is a special camera node that displays the 2D GUI.

 viewport
  This specifies the position and dimensions of the GUI within a
  window, *however* at the moment the origin must be at 0,0.

Here's an example that uses a single window mapped across 3
displays. The displays are in a video wall configuration in a
horizontal row.

<PropertyList>
  <sim>
    <rendering>
      <camera-group>
        <window>
          <name>wide</name>
          <host-name type="string"></host-name>
          <display>0</display>
          <screen>0</screen>
          <width>3840</width>
          <height>1024</height>
          <decoration type = "bool">false</decoration>
        </window>
        <camera>
          <window>
            <name>wide</name>
          </window>
          <viewport>
            <x>0</x>
            <y>0</y>
            <width>1280</width>
            <height>1024</height>
```

```
    </viewport>
    <view>
      <heading-deg type = "double">0</heading-deg>
    </view>
    <frustum>
      <top>0.133</top>
      <bottom>-0.133</bottom>
      <left>-.5004</left>
      <right>-.1668</right>
      <near>0.4</near>
      <far>120000.0</far>
    </frustum>
  </camera>
  <camera>
    <window>
      <name type="string">wide</name>
    </window>
    <viewport>
      <x>1280</x>
      <y>0</y>
      <width>1280</width>
      <height>1024</height>
    </viewport>
    <view>
      <heading-deg type = "double">0</heading-deg>
    </view>
    <frustum>
      <top>0.133</top>
      <bottom>-0.133</bottom>
      <left>-.1668</left>
      <right>.1668</right>
      <near>0.4</near>
      <far>120000.0</far>
    </frustum>
  </camera>
  <camera>
    <window>
      <name>wide</name>
    </window>
    <viewport>
```

```
            <x>2560</x>
            <y>0</y>
            <width>1280</width>
            <height>1024</height>
          </viewport>
          <view>
            <heading-deg type = "double">0</heading-deg>
          </view>
          <frustum>
            <top>0.133</top>
            <bottom>-0.133</bottom>
            <left>.1668</left>
            <right>.5004</right>
            <near>0.4</near>
            <far>120000.0</far>
          </frustum>
        </camera>
        <gui>
          <window>
            <name type="string">wide</name>
          </window>
        </gui>
      </camera-group>
    </rendering>
  </sim>
</PropertyList>
```

Here's a complete example that uses a seperate window on each
display. The displays are arranged in a shallow arc with the left and
right displays at a 45.3 degree angle to the center display because,
at the assumed screen dimensions, the horizontal field of view of one
display is 45.3 degrees. Each camera has its own window definition;
the center window is given the name "main" so that the GUI definition
can refer to it. Note that the borders of the displays are not
accounted for.

```
<PropertyList>
  <sim>
    <rendering>
      <camera-group>
```

```
<camera>
  <window>
    <host-name type="string"></host-name>
    <display>0</display>
    <screen>0</screen>
    <fullscreen type = "bool">true</fullscreen>
  </window>
  <view>
    <heading-deg type = "double">45.3</heading-deg>
  </view>
  <frustum>
    <top>0.133</top>
    <bottom>-0.133</bottom>
    <left>-.1668</left>
    <right>.1668</right>
    <near>0.4</near>
    <far>120000.0</far>
  </frustum>
</camera>
<camera>
  <window>
    <name type="string">main</name>
    <host-name type="string"></host-name>
    <display>0</display>
    <screen>1</screen>
    <fullscreen type = "bool">true</fullscreen>
  </window>
  <view>
    <heading-deg type = "double">0</heading-deg>
  </view>
  <frustum>
    <top>0.133</top>
    <bottom>-0.133</bottom>
    <left>-.1668</left>
    <right>.1668</right>
    <near>0.4</near>
    <far>120000.0</far>
  </frustum>
</camera>
<camera>
```

```
            <window>
              <host-name type="string"></host-name>
              <display>0</display>
              <screen>2</screen>
              <fullscreen type = "bool">true</fullscreen>
            </window>
            <view>
              <heading-deg type = "double">-45.3</heading-deg>
            </view>
            <frustum>
              <top>0.133</top>
              <bottom>-0.133</bottom>
              <left>-.1668</left>
              <right>.1668</right>
              <near>0.4</near>
              <far>120000.0</far>
            </frustum>
          </camera>
          <gui>
            <window>
              <name type="string">main</name>
            </window>
          </gui>
        </camera-group>
      </rendering>
    </sim>
</PropertyList>
```

28   Osgtext

```
This document describes the syntax for text objects in the scene graph.
Text nodes are configured using XML and may appear within a model description
file, like other models or the particlesystem.

For the anxious reader, here is a complete example of a text node:

<!-- Must be enclosed by a <text/> node
<text>
  <!-- It should have a name. Can be used for other animations -->
```

```
<name>My first Text</name>
<!-- Use offsets for the initial placement -->
<offsets>
  <pitch-deg>0</pitch-deg>
  <heading-deg>0</heading-deg>
  <roll-deg>0</roll-deg>
  <x-m>0</x-m>
  <y-m>0</y-m>
  <z-m>0</z-m>
</offsets>

<!-- instead of using pitch/heading/roll offset, one may use
     axis-alignment -->
<!-- remember: x backwards, y right and z up -->
<axis-alignment>xy-plane</axis-alignment>
<!--
<axis-alignment>reversed-xy-plane</axis-alignment>
<axis-alignment>xz-plane</axis-alignment>
<axis-alignment>reversed-xz-plane</axis-alignment>
<axis-alignment>yz-plane</axis-alignment>
<axis-alignment>reversed-yz-plane</axis-alignment>
<axis-alignment>screen</axis-alignment>
-->

<!-- what type of text to draw, use on of literal, text-value or number-value -->
<!-- A simple constant, never changing string -->
<type type="string">literal</type>
<text type="string">Hello, world!</text>

<!-- The string value of a property -->
<type type="string">text-value</type>
<property type="string">some/property</property>
<format type="string">%s</format> <!-- the printf() format to display the value -->

<!-- A number from a property -->
<type type="string">number-value</type>
<property type="string">position/latitude-deg</property>
<factor type="double">1.0</factor> <!-- optional, scale the propertie's value -->
<offset type="double">0.0</offset> <!-- optional, shift the propertie's value -->
<format type="string">%5.2lf</format> <!-- printf() format to display -->
```

```xml
<truncate type="bool">false</truncate> <!-- truncate to an integer value -->

<layout>left-to-right</layout> <!-- default -->
<!--
<layout>right-to-left</layout>
<layout>vertical</layout>
-->


<draw-text type="bool">true</draw-text> <!-- draw the text itself -->
<draw-alignment type="bool">false</draw-alignment> <!-- draw crosshair at object ce
<draw-boundingbox type="bool">false</draw-boundingbox> <!-- draw a bounding box -->

<font>led.txf</font> <!-- The font file name, relative to data/Fonts -->
<character-size type="double">0.01</character-size> <!-- size (height) im meters --
<character-aspect-ratio type="double">1.0</character-aspect-ratio>
<max-height>0.012</max-height> <!-- the maximum height of the text -->
<max-width>0.040</max-width> <!-- the maximum width of the text -->
<font-resolution>
  <width type="int">32</width>
  <height type="int">32</height>
</font-resolution>

<!-- chose one of the kerning types or omit for default -->
<kerning>default</kerning>
<!--
<kerning>unfitted</kerning>
<kerning>none</kerning>
-->


<alignment>center-center</alignment> <!-- alignment of the text itself -->
<!-- possible values are
<alignment>left-top</alignment>
<alignment>left-center</alignment>
<alignment>left-bottom</alignment>
<alignment>center-top</alignment>
<alignment>center-center</alignment>
<alignment>center-bottom</alignment>
<alignment>right-top</alignment>
<alignment>right-center</alignment>
<alignment>right-bottom</alignment>
```

```
    <alignment>left-baseline</alignment>
    <alignment>center-baseline</alignment>
    <alignment>right-baseline</alignment>
    <alignment>baseline</alignment>
    -->

</text>
```

The <text/> node may appear within <model/> or <PropertyList/> nodes. If you place
your text directly within your model file, use <text></text> nodes. You can also put
your <text> configuration into a separate file using the well known include directive

```
Your model.xml file:
<model>
  <path>may-fancy-model.ac</path>
  <text include="HelloWorld.xml"/>
</model>

Your HelloWorld.xml:
<PropertyList>
  <name>Hello World</name>
  <font>Helvetica.txf</font>
  <type type="string">literal</type>
  <text type="string">Hello, world!</text>
  <!-- etc. - you get the idea -->
</PropertyList>
```

Animation can be applied to text nodes like any other object. To give your text some
color, use the material animation, or translate, rotate, scale or spin your text as
you like.

29   Properties

```
================================================================================
CONTROLS
================================================================================


Flight Controls
---------------
```

```
/controls/flight/aileron
/controls/flight/aileron-trim
/controls/flight/elevator
/controls/flight/elevator-trim
/controls/flight/rudder
/controls/flight/rudder-trim
/controls/flight/flaps
/controls/flight/slats
/controls/flight/BLC                          // Boundary Layer Control
/controls/flight/spoilers
/controls/flight/speedbrake
/controls/flight/wing-sweep
/controls/flight/wing-fold
/controls/flight/drag-chute

Engines
-------
/controls/engines/throttle_idle
/controls/engines/engine[%d]/throttle
/controls/engines/engine[%d]/starter
/controls/engines/engine[%d]/fuel-pump
/controls/engines/engine[%d]/fire-switch
/controls/engines/engine[%d]/fire-bottle-discharge
/controls/engines/engine[%d]/cutoff
/controls/engines/engine[%d]/mixture
/controls/engines/engine[%d]/propeller-pitch
/controls/engines/engine[%d]/magnetos
/controls/engines/engine[%d]/boost
/controls/engines/engine[%d]/WEP
/controls/engines/engine[%d]/cowl-flaps-norm
/controls/engines/engine[%d]/feather
/controls/engines/engine[%d]/ignition
/controls/engines/engine[%d]/augmentation
/controls/engines/engine[%d]/afterburner
/controls/engines/engine[%d]/reverser
/controls/engines/engine[%d]/water-injection
/controls/engines/engine[%d]/condition

Fuel
----
```

```
/controls/fuel/dump-valve
/controls/fuel/tank[%d]/fuel_selector
/controls/fuel/tank[%d]/to_engine
/controls/fuel/tank[%d]/to_tank
/controls/fuel/tank[%d]/boost-pump[%d]

/consumables/fuel/tank[%d]/level-lbs
/consumables/fuel/tank[%d]/level-gal_us
/consumables/fuel/tank[%d]/capacity-gal_us
/consumables/fuel/tank[%d]/density-ppg
/consumables/fuel/total-fuel-lbs
/consumables/fuel/total-gal_us
```

```
Gear
----
/controls/gear/brake-left
/controls/gear/brake-right
/controls/gear/brake-parking
/controls/gear/steering
/controls/gear/gear-down
/controls/gear/antiskid
/controls/gear/tailhook
/controls/gear/tailwheel-lock
/controls/gear/wheel[%d]/alternate-extension
```

```
Anti-Ice
--------
/controls/anti-ice/wing-heat
/controls/anti-ice/pitot-heat
/controls/anti-ice/wiper
/controls/anti-ice/window-heat
/controls/anti-ice/engine[%d]/carb-heat
/controls/anti-ice/engine[%d]/inlet-heat
```

```
Hydraulics
----------
/controls/hydraulic/system[%d]/engine-pump
/controls/hydraulic/system[%d]/electric-pump
```

```
Electric
--------
/controls/electric/battery-switch
/controls/electric/external-power
/controls/electric/APU-generator
/controls/electric/engine[%d]/generator
/controls/electric/engine[%d]/bus-tie

Pneumatic
---------
/controls/pneumatic/APU-bleed
/controls/pneumatic/engine[%d]/bleed

Pressurization
--------------
/controls/pressurization/mode
/controls/pressurization/dump
/controls/pressurization/outflow-valve
/controls/pressurization/pack[%d]/pack-on

Lights
------
/controls/lighting/landing-lights
/controls/lighting/turn-off-lights
/controls/lighting/formation-lights
/controls/lighting/taxi-light
/controls/lighting/logo-lights
/controls/lighting/nav-lights
/controls/lighting/beacon
/controls/lighting/strobe
/controls/lighting/panel-norm
/controls/lighting/instruments-norm
/controls/lighting/dome-norm

Armament
--------
/controls/armament/master-arm
/controls/armament/station-select
/controls/armament/release-all
/controls/armament/station[%d]/stick-size
```

```
/controls/armament/station[%d]/release-stick
/controls/armament/station[%d]/release-all
/controls/armament/station[%d]/jettison-all

Seat
----
/controls/seat/vertical-adjust
/controls/seat/fore-aft-adjust
/controls/seat/cmd_selector_valve
/controls/seat/eject[%d]/initiate
/controls/seat/eject[%d]/status

APU
---
/controls/APU/off-start-run
/controls/APU/fire-switch

Autoflight
----------
/controls/autoflight/autopilot[%d]/engage
/controls/autoflight/autothrottle-arm
/controls/autoflight/autothrottle-engage
/controls/autoflight/heading-select
/controls/autoflight/altitude-select
/controls/autoflight/bank-angle-select
/controls/autoflight/vertical-speed-select
/controls/autoflight/speed-select
/controls/autoflight/mach-select
/controls/autoflight/vertical-mode
/controls/autoflight/lateral-mode


===============================================================================
FDM (Aircraft settings)
===============================================================================

Position
--------------
/position/latitude-deg
/position/longitude-deg
/position/altitude-ft
```

```
Orientation
-----------
/orientation/roll-deg
/orientation/pitch-deg
/orientation/heading-deg

/orientation/roll-rate-degps
/orientation/pitch-rate-degps
/orientation/yaw-rate-degps

/orientation/side-slip-rad
/orientation/side-slip-deg
/orientation/alpha-deg

Velocities
----------
/velocities/airspeed-kt
/velocities/mach
/velocities/speed-north-fps
/velocities/speed-east-fps
/velocities/speed-down-fps

/velocities/uBody-fps
/velocities/vBody-fps
/velocities/wBody-fps

/velocities/vertical-speed-fps
/velocities/glideslope

Acceleration
------------
/accelerations/nlf

/accelerations/ned/north-accel-fps_sec
/accelerations/ned/east-accel-fps_sec
/accelerations/ned/down-accel-fps_sec

/accelerations/pilot/x-accel-fps_sec
/accelerations/pilot/y-accel-fps_sec
```

```
/accelerations/pilot/z-accel-fps_sec

Engines
-------

common:
/engines/engine[%d]/fuel-flow-gph
/engines/engine[%d]/fuel-flow_pph
/engines/engine[%d]/thrust_lb
/engines/engine[%d]/running
/engines/engine[%d]/starter
/engines/engine[%d]/cranking

piston:
/engines/engine[%d]/mp-osi
/engines/engine[%d]/egt-degf
/engines/engine[%d]/oil-temperature-degf
/engines/engine[%d]/oil-pressure-psi
/engines/engine[%d]/cht-degf
/engines/engine[%d]/rpm

turbine:
/engines/engine[%d]/n1
/engines/engine[%d]/n2
/engines/engine[%d]/epr
/engines/engine[%d]/augmentation
/engines/engine[%d]/water-injection
/engines/engine[%d]/ignition
/engines/engine[%d]/nozzle-pos-norm
/engines/engine[%d]/inlet-pos-norm
/engines/engine[%d]/reversed
/engines/engine[%d]/cutoff

propeller:
/engines/engine[%d]/rpm
/engines/engine[%d]/pitch
/engines/engine[%d]/torque


===============================================================================
```

```
LIGHT
================================================================================
/sim/time/sun-angle-rad
/rendering/scene/ambient/red
/rendering/scene/ambient/ggreen
/rendering/scene/ambient/blue
/rendering/scene/diffuse/red
/rendering/scene/diffuse/green
/rendering/scene/diffuse/blue
/rendering/scene/specular/red
/rendering/scene/specular/green
/rendering/scene/specular/blue
```

30   Protocol

The generic communication protocol for FlightGear provides a powerful way
of adding a simple ASCII based or binary input/output protocol, just by
defining an XML encoded configuration file and placing it in the
$FG_ROOT/Protocol/ directory.

```
== file layout ====================================================================

A protocol file can contain either or both of <input> and <output>
definition blocks. Which one is used depends on how the protocol
is called  (e.g. --generic=file,out,1,/tmp/data.xml,myproto would
only use the <output> definitions block).

  <?xml version="1.0"?>
  <PropertyList>
      <generic>

          <output>
              <binary_mode>false</binary_mode>
              <line_separator></line_separator>
              <var_separator></var_separator>
              <preamble></preamble>
```

```
                  <postamble></postamble>

                  <chunk>
                          ... first chunk spec ...
                  </chunk>

                  <chunk>
                          ... another chunk etc. ...
                  </chunk>
          </output>

          <input>
              <line_separator></line_separator>
              <var_separator></var_separator>

              <chunk>
                      ... chunk spec ...
              </chunk>
          </input>

      </generic>
  </PropertyList>
```

```
== input/output parameters ==================================================

Both <output> and <input> blocks can contain information about
the data mode (ascii/binary) and about separators between fields
and data sets, as well as a list of <chunk>s. Each <chunk> defines
a property that should be written (and how), or a variable and which
property it should be written to.

--- ASCII protocol parameters ---

output only:
  <preamble>        STRING  default: ""    file header put on top of the file
  <postamble>       STRING  default: ""    file footer put at the end of the file
```

```
input & output:
  <binary_mode>      BOOL     default: false (= ASCII mode)
  <var_separator>  STRING  default: ""    field separator
  <line_separator> STRING  default: ""    separator between data sets
```

<var_separator> are put between every two output properties, while
<line_separator> is put at the end of each data set. Both can contain
arbitrary strings or one of the following keywords:

```
  Name               Character

  newline            '\n'
  tab                '\t'
  formfeed           '\f'
  carriagereturn     '\r'
  verticaltab        '\v'
```

Typical use could be:

```
  <var_separator>tab</var_separator>
  <line_separator>newline</var_separator>
```

or

```
  <var_separator>\t</var_separator>
  <line_separator>\r\n</line_separator>
```

--- Binary protocol parameters ---

To enable binary mode, simply include a <binary_mode>true</binary_mode> tag in
your XML file. The format of the binary output is tightly packed, with 1 byte
for bool, 4 bytes for int, and 8 bytes for double. At this time, strings are not
supported. A configurable footer at the end of each "line" or packet of binary
output can be added using the <binary_footer> tag. Options include the length
of the packet, a magic number to simplify decoding. Examples:

```
  <binary_footer>magic,0x12345678</binary_footer>
```

```
<binary_footer>length</binary_footer>
<binary_footer>none</binary_footer>                      <!-- default -->




== variable parameters (chunk spec) =========================================

Both <input> and <output> block can contain a list of <chunk> specs,
each of which describes the properties of on variable to write/read.


   <name>          for ease of use (not tranferred)
   <node>          the property tree node which provides the data
   <type>          the value type (needed for formatting)
                   one of string, float, bool, int (default: int)
   <format>        (ASCII protocol only, not used or needed in binary mode)
                   defines the actual piece of text which should be sent.
                   it can include "printf" style formatting options like:
                              <type>
                     %s       string
                     %d       integer (default)
                     %f       float

   <factor>        an optional multiplication factor which can be used for
                   unit conversion. (for example, radians to degrees).
   <offset>        an optional offset which can be used for unit conversion.
                   (for example, degrees Celcius to degrees Fahrenheit).


Chunks can also consist of a single constant <format>, like in:
  <format>Data Section</format>



== examples ================================================================

Writes log of this form:

V=16
H=3.590505
```

```
P=3.59
V=12
H=3.589020
P=3.59
```

```xml
<?xml version="1.0"?>

<PropertyList>
  <generic>

    <output>
      <line_separator>newline</line_separator>
      <var_separator>newline</var_separator>
      <binary_mode>false</binary_mode>

      <chunk>
        <name>speed</name>
        <format>V=%d</format>
        <node>/velocities/airspeed-kt</node>
      </chunk>

      <chunk>
        <name>heading (rad)</name>
        <format>H=%.6f</format>
        <type>float</type>
        <node>/orientation/heading-deg</node>
        <factor>0.0174532925199433</factor>  <!-- degrees to radians -->
      </chunk>

      <chunk>
        <name>pitch angle (deg)</name>
        <format>P=%03.2f</format>
        <node>/orientation/pitch-deg</node>
      </chunk>
    </output>

 </generic>
</PropertyList>
```

```
-- writing data in XML syntax ------------------------------------------------

Assuming the file is called $FG_ROOT/Protocol/xmltest.xml, then it could be
used as   $ fgfs --generic=file,out,1,/tmp/data.xml,xmltest


<?xml version="1.0"?>

<PropertyList>
  <generic>
    <output>
      <binary_mode>false</binary_mode>
      <var_separator>\n</var_separator>
      <line_separator>\n</line_separator>
      <preamble>&lt;?xml version="1.0"?&gt;\n\n&lt;data&gt;\n</preamble>
      <postamble>&lt;/data&gt;\n</postamble>

      <chunk>
        <format>\t&lt;set&gt;</format>
      </chunk>

      <chunk>
        <node>/position/altitude-ft</node>
        <type>float</type>
        <format>\t\t&lt;altitude-ft&gt;%.8f&lt;/altitude-ft&gt;</format>
      </chunk>

      <chunk>
        <node>/velocities/airspeed-kt</node>
        <type>float</type>
        <format>\t\t&lt;airspeed-kt&gt;%.8f&lt;/airspeed-kt&gt;</format>
      </chunk>

      <chunk>
        <format>\t&lt;/set&gt;</format>
      </chunk>
```

```
        </output>
    </generic>
</PropertyList>


-- Analyzing the resulting binary packet format -----------------------------

A utility called generic-protocol-analyse can be found under
FlightGear/utils/xmlgrep which can be used to analyze the resulting
data packet for the binary protocol.

The output would be something like:

bintest.xml
Generic binary output protocol packet description:

 pos | size |  type  | factor      | description
-----|------|--------|------------|-----------------------
   0 |    4 |    int |            | indicated speed (kt)
   4 |    4 |  float |            | pitch att (deg)
   8 |    4 |  float |            | magnetic heading (deg)
  12 |    4 |    int |            | outside air temperarure (degF)
  16 |    1 |   bool |            | autocoord

total package size: 17 bytes
```

31   Scenery

This document describes how FlightGear searches and loads scenery, how to
add static objects to the scenery as well as the syntax of *.stg files.

Contents --

1 scenery path
2 terrasync

1 scenery path ---

FlightGear loads scenery by default from the Scenery/ subdirectory of its
data directory. The path to this data directory can be set via environment
variable FG_ROOT or the --fg-root option. The scenery path can be set
independently via environment variable FG_SCENERY or option --fg-scenery.
The order of precedence is as follows:

```
  --fg-scenery=/some/dir        ... highest priority
  $FG_SCENERY
  $FG_ROOT/Scenery/             ... lowest priority
```

A scenery specification may be a list of paths, separated by the OS-specific
path separator (colon on Unix/OSX, semicolon on MS Windows). The paths are
searched in the order from left to right:

```
    FG_SCENERY=/first/dir:/second/dir:/third/dir
                                    (likewise with --fg-scenery option)
```

Each of the scenery paths can follow one of two possible layouts: with or
without Terrain/ and Objects/ subdirectories. As soon as either or both
of these subdirectories are found, scenery is only searched *in* these two,
but not in any other directory on the same hierarchy level!

This example shows which directories are used to search for scenery:

```
  $ ls /first/dir
  w130n30/                  searched

  $ ls /second/dir
  Objects/                  searched
  Terrain/                  searched
  w130n30/                  *not* searched

  $ ls /third/dir
  Terrain/                  searched
  w130n30/                  *not* searched
```

If FlightGear searches for a particular "tile" file, let's say for
"w130n30/w123n37/942050.stg", then (using the above examples) it looks
into

```
  /first/dir/w130n30/w123n37/942050.stg           (A)

  /second/dir/Terrain/w130n30/w123n37/942050.stg  (B)\__ same path element
  /second/dir/Objects/w130n30/w123n37/942050.stg  (C)/      /second/dir

  /third/dir/Terrain/w130n30/w123n37/942050.stg   (D)
```

but as soon as it finds an OBJECT_BASE entry it only finishes this
path element and then stops scanning. So, if (B) contains an entry
"OBJECT_BASE 942050.btg, then the twin Objects/ directory (C) will
be read, too. But (D) will *not*! Objects/ and Terrain/ directories

174

are laid out equally. Airport and elevation data, as well as airport
inventory objects are usually put into Terrain/, while other objects
are put into Objects/.

This searching behavior is usually used to collect user-added
custom objects first, then to read in standard scenery and objects
that came with the distribution (San Francisco Bay area), and to
use locally added scenery everywhere else. So a typical scenery
path specification could look like this:

```
FG_SCENERY=$HOME/.fgfs/Scenery:$FG_ROOT/Scenery:$FG_ROOT/WorldScenery
```

The third path would then be populated by the user with unpacked scenery
archives downloaded from http://www.flightgear.org/Downloads/scenery.html,
or by using terrasync (see next section).
Additional objects can be downloaded from the FlightGear Objects
Database (http://scenemodels.flightgear.org/download/). (Note that
those objects are occasionally merged into the flightgear.org/terrasync
packages, so you may end up with doubled entries!)

Using a private directory for downloaded add-on scenery and adding
that path to FG_SCENERY is the preferred way. This separates default
data from locally added data, and makes administration and later updates
easier.

HINT: if you want to see where FlightGear is searching and finding
terrain/objects, start it with the --log-level=info option.

2 terrasync --

FlightGear comes with a utility "terrasync" that allows downloading
scenery (literally) "on-the-fly. Given the scenery path setup from
section 1 you could use terrasync with a script like this:

```
#!/bin/bash
PORT=5503
```

```
  nice terrasync -p $PORT -d $FG_ROOT/WorldScenery&
  fgfs --atlas=socket,out,1,localhost,$PORT,udp $*
  killall terrasync
```

If you name it "fgfsterra", then you can use it just like you would use
"fgfs", but behind the scenes it would update your scenery everywhere in
sight and save the files to $FG_ROOT/WorldScenery. Example:

```
  $ ./fgfsterra --aircraft=ufo --airport=LOXZ
```

Note, however, that if it downloads scenery for the area around your
starting location, then you'll only see that after the next start, or
after you flew or teleported to a distant location and then back.
terrasync depends on the rsync application and an open port 873,
so it may not be available/usable on MS Windows.

3 stg files --

stg files ("static terragear") define the static elements of a scenery
"tile", including the terrain elevation data, airport geometry, and all
static objects placed on this tile. (See section 5 for how to find out which
geo coordinates belong to which tile.) Four of the available key words
are followed by a string and four numbers. The meaning of these numbers
is always the same and described in section 3.3.

3.1 OBJECT_BASE
----

specifies the terrain elevation data file. These files are generated with
the TerraGear tools (http://www.terragear.org/) and have file extension
".btg" ("binary terragear"; there used to be an "*.atg" file, too, where
the 'a' stood for ASCII).

Example:

```
  OBJECT_BASE 942050.btg
```

The entry may be anywhere in the 942050.stg file, on a separate line.

3.2 OBJECT
----

specifies an airport geometry 'drop-in' file. The scenery elevation file
has cut out holes for airports, that are filled with such objects. They
are usually called after the airport ICAO id:

Example:

```
  OBJECT KSFO.btg
```

These files are, again, created by TerraGear tools and are usually gzipped,
so you'll find that file stored as KSFO.btg.gz.

3.3 OBJECT_SHARED
----

add static object to the tile.

Example:

```
  OBJECT_SHARED Models/Airport/tower.xml -122.501090 37.514830 15.5 0.00 0.00 0.00
```

Syntax:

```
  OBJECT_SHARED <object-path> <lon> <lat> <elev-m> <hdg-deg> <pitch-deg> <roll-deg>
```

The <object-path> is relative to the data directory (FG_ROOT).

<elev-m> is in meter and relative to mean sea-level (in the fgfs world).
<hdg-deg> is in degree, counter-clockwise with North being 0.0. Note
that this differs from about every other place in FlightGear, most notably
the /orientation/heading-deg entry in the property system, which is clockwise.
<pitch-deg> and <roll-deg> are in degree and optional.
OBJECT_SHARED models are cached and reused. They are only once in memory
and never freed. (See also the next section.)

3.4 OBJECT_STATIC
----

add static objects to the tile, just like OBJECT_SHARED. There are three
differences to OBJECT_SHARED (apart from the name):

(A) the path is relative to the tile directory where the *.stg file with
    this entry is located. For example, relative to 130n30/w123n37/. This
    usually means that all 3D object files, textures, and XML animation
    files are in this tile directory, too.

(B) these objects are *not* cached and kept loaded, but rather freed with
    the tile (that is, when you leave that area).

(C) the animation XML files may contain Nasal blocks <nasal><load> and
    <nasal><unload> which are executed on loading/unloading.

Example:

      OBJECT_STATIC ggb-fb.xml -122.4760494 37.81876042 0 105 0.00 0.00

3.5 OBJECT_SIGN
----

defines taxiway or runway sign. The syntax is much like that of OBJECT_SHARED
entries, except that the path is replaced with a sign contents specification

and that there is an additional size value at the end of the line.

Example:

```
OBJECT_SIGN {@R}10L-28R{@L}C -122.35797457 37.61276290 -0.5398 74.0 2
```

The sign specification defines the sign contents. We try to resemble the apt.dat 850 specifications in our implementation.
In the simplest form it contains just 'normal' text, for example:

```
EXIT
```

This will create a black panel of 1m height with "EXIT" written on it in white versal letters. Actually, each of those characters are single-letter glyph names that are looked up in the <glyph> map of a texture font <material> entry in $FG_ROOT/materials.xml. It just happens that the <glyph> entry for <name> 'E' maps to a drawn 'E' in the font texture. This isn't true for all ASCII characters. Many aren't mapped at all (and thus not available), others are mapped to non-standard drawings. The '_', for example, is mapped to an empty black area and can therefore be used as a space. (The sign specification must not contain real spaces.) The '*' is mapped to a raised period.

Some glyph names consist of more than one character, and can't, thus, be used directly. They have to be put in a pair of curly braces:

```
{^rd}
```

This creates an arrow that points to the right and down. Braces can really contain a list of glyph names, separated by commas (no space!). Single-letter glyph names can be used that way, too, or in any mixture of both methods:

```
EXIT
{E,X,I,T}
{E}{X}{I}{T}
EX{I,T}
E{X,I}T{^lu,^rd}
{^u}EXIT{^u}
```

Multi-letter glyph names are usually used for symbols. Arrow symbol names
always start with a caret ("arrow head") and the left or right direction
always comes first (like the x in a Cartesian coordinate system). Here's
a list of some of the available names (see $FG_ROOT/materials.xml for
more):

```
  ^l        left arrow
  ^r        right arrow
  ^u        up arrow
  ^d        down arrow
  ^lu       left-up arrow
  ^ld       left-down arrow
  ^ru       right-up arrow
  ^rd       right-down arrow
  no-entry "no entry" symbol
  critical runway critical area
  safety    ils safety area
  hazard    end of taxiway
```

There are commands for pre-defined sign types according to the FAA
specification (5345-44; see http://www.google.com/search?q=5345-44g).

```
  @Y    "Direction, Destination, Boundary" sign (black on yellow)
  @R    "Mandatory Instruction" sign (white on red with black outline)
  @L    "Location" sign  (yellow text and frame on black)
  @B    "Runway Distance Remaining" sign  (white on black)
```

Examples:

```
  {@R}10L-28R{@L}C
  {@Y,^l}P|{^lu}N{@L}F{@Y}F{^ru}
  {@Y,^ld}C   ...  same as any of {@Y}{@ld}C  {@Y,@ld,C}
  {@B}17
```

Syntax errors are reported in --log-level=debug, in the SG_TERRAIN
group. You can use this command line to filter out such messages:

```
$ fgfs --log-level=debug 2>&1|grep OBJECT_SIGN
```

4 model manager ("/models/model") --

4.1 static objects

Another way to add objects to the scenery is via the "model manager".
It reads all /models/model entries at startup and places these objects
in the scenery. Just load a definition like the following into the
property tree, for example by putting it into $FG_ROOT/preferences.xml, or
better: an XML file that you load with e.g. --config=$HOME/.fgfs/stuff.xml:

```
  <models>
      <model n="0">
          <name>pony</name>
          <path>Local/pony.ac</path>
          <longitude-deg>-115.8352869</longitude-deg>
          <latitude-deg>37.24302849</latitude-deg>
          <elevation-ft>4534.691321</elevation-ft>
          <heading-deg>0</heading-deg>
          <pitch-deg>0</pitch-deg>
          <roll-deg>0</roll-deg>
      </model>
  </models>
```

The <path> is relative to $FG_ROOT, the <name> is optional. One can leave the
heading/pitch/roll entries away, in which case they are set to zero. The values
are fixed and unchangeable at runtime.

4.2 dynamic objects

Any of the model properties can be made changeable at runtime by appending
"-prop" and using a property path name instead of the fixed value:

```
  <local>
      <pony>
          <longitude-deg>-115.8352869/<longitude-deg>
          <latitude-deg>37.24302849</latitude-deg>
          <elevation-ft>4534.691321</elevation-ft>
          <heading-deg>0</heading-deg>
      </pony>
  </local>

  <models>
      <model n="1">
          <name>pony</name>
          <path>Local/pony.ac</path>
          <longitude-deg-prop>/local/pony/longitude-deg</longitude-deg-prop>
          <latitude-deg-prop>/local/pony/latitude-deg</latitude-deg-prop>
          <elevation-ft-prop>/local/pony/elevation-ft</elevation-ft-prop>
          <heading-deg-prop>/local/pony/heading-deg</heading-deg-prop>
          <pitch-deg>1.234</pitch-deg>  <!-- static, just for fun -->
      </model>
  </models>
```

Then one can move the pony around by changing the values in /local/pony/ in
the property system. One can, of course, use other animals, too.

4.3 loading/unloading at runtime

Both dynamic and static model-manager-models can be loaded and unloaded
at runtime. For loading you first create a new <model> entry under <models>,
initialize all properties there (<longitude-deg> or <longitude-deg-prop>,

etc.), and finally you create a child <load> of any type in this group.
This is the signal for the model manager to load the object. You can
remove the <load> property after that. It has no further meaning.

To remove a model-manager model at runtime, you simply delete the whole
<model> group.

5 tools for object placing --

5.1 calc-tile.pl

For finding out the tile number for a given geo coordinate pair there's
a script "scripts/perl/scenery/calc-tile.pl" in the FlightGear sources.
You feed longitude and latitude to it and it returns the path to the
*.stg file where you have to add the object entry.

```
  $ perl calc-tile.pl 16.1234 48.5678
  Longitude: 16.1234
  Latitude:  48.5678
  Tile:      3220128
  Path:      "e010n40/e016n48/3220128.stg"
```

5.2 ufo scenery object editor

The ufo has a scenery object editor built-in. It uses the model manager
described in section 4. To place objects with it, start fgfs, optionally
with specifying an initial model type ("cursor") and a list of subdirectories
of $FG_ROOT where the ufo should search for available 3D models ("source"):

```
  $ fgfs --aircraft=ufo --prop:cursor=Models/Airport/radar.xml  \
```

```
                    --prop:source=Models,Scenery/Objects
```

Then click anywhere on the terrain to add a model (left mouse button).
You can open the adjustment dialog (Tab-key) to make adjustments to
position and orientation. Click as often as you like, choose further
models from the space-key dialog. You can select an already placed object
by Ctrl-clicking at its base (not at the object itself, but the surface
point where it's located!). By also holding the Shift key down, you
can select several objects or add them to a selection. You can remove
the selected object(s) with the Backspace-key. (See the ?-key dialog
for futher available keys.) After clicking on the input field right
over the status line (invisible if there's no text in it) you can enter
a comment/legend for the selected object.

And finally, you dump the object data to the terminal (d-key) or export
them to a file $HOME/.fgfs/ufo-model-export.xml (Unix) or
%APPDATA%\flightgear.org\ufo-model-export.xml (MS Windows).

You can now put the generated object entries into the specified *.stg
file to make them permanent. Or load the whole exported *.xml file
via --config option:

```
  $ fgfs --config=$HOME/.fgfs/ufo-model-export.xml
```

If you choose the sign placeholder object from the m-key dialog (first
entry; "Aircraft/ufo/Models/sign.ac"), then an OBJEC_SIGN *.stg line
will be generated with the legend used as sign contents. If you didn't
insert any legend, then the sign text will be: NO CONTENTS and a 4 digits
random number for later identification in the *.stg file.

Unfortunately, objects added with this method are kept in memory, no
matter where you are actually flying, so the *.stg method is preferable.

6 embedded Nasal in XML files (static objects and AI) ------------------------

6.1 static models

Objects loaded via OBJECT_STATIC in *.stg files as well as AI models loaded
via scenarios may contain embedded Nasal code. This can be used to drive
more advanced animations. An example is a lighthouse with specific light
signals, or hangar doors that open when the "player"'s aircraft is nearby.
The Nasal code is added to the object's XML wrapper/animation file, anywhere
on the top level, for example:

```
<PropertyList>
    <path>lighthouse.ac</path>

    <nasal>
        <load>
            var loop_id = 0;
            var light = aircraft.light.new("
                    "/models/static/w120n30/w118n35/lighthouse/light",
                    [2, 1, 2, 1, 2, 1, 2, 5]);

            var loop = func(id) {
                id == loop_id or return;
                light.switch(getprop("/sim/time/sun-angle-rad") > 1.37);
                settimer(func { loop(id) }, 30);
            }
            loop(loop_id += 1);
        </load>

        <unload>loop_id += 1</unload>
    </nasal>

    <animation>
        <type>select</type>
        <object-name>light-halo</object-name>
        <property>/models/static/w120n30/w118n35/lighthouse/light/state</property>
    </animation>

    ...
</PropertyList>
```

The <load> part is executed when the scenery tile on which the model is placed
is loaded into memory. It can start timers or listeners that modify properties,
which are then queried by the <animation>. As a convention developers are requested
to use "/models/static/" + <tile-path> + <file-basename>. So, in the above example
file "$FG_ROOT/Scenery/Objects/w120n30/w118n35/lighthouse.xml" all properties
are stored under "/models/static/w120n30/w118n35/lighthouse/". That way collisions
with other models are quite unlikely.

An optional <unload> part is executed when the tile and model is removed from
memory. Note that this is only when the "player" is already far away! To
cause minimal impact on the framerate it is recommended to do as few
calculations as possible, to use as large timer intervals as possible, and to
stop all timers and listeners in the <unload> part, as shown in the example.

All Nasal variables/functions are in a separate namespace, which is named
after the file name. It's recommended not to access this namespace from
outside for other than development purposes.

What the above code does: as soon as the model is loaded, an aircraft.light
is created with a specific light sequence. Then, in half-minute intervals,
the light is turned on or off depending on the sun angle. On <unload> the
loop identifier is increased, which makes the loop terminate itself. For
more info about this technique, see the Nasal wiki.

6.2 AI models

Here the syntax is the same like for static models. The only two differences
are:

- these models are currently only removed at program end, so it's more
  important to consider effects on performance.

- AI models don't need to store their properties in /models/static/...,
  but get a separate node under /ai/models/, for example /ai/models/carrier[1].

The embedded Nasal code can access this dynamically assigned property
via cmdarg() function, which returns a props.Node hash. Example:

```
<nasal>
    <load>print("my data are under ", cmdarg().getPath())</load>
    <unload>print("Currently I'm only called at fgfs exit!")</unload>
</nasal>
```

32   Sound

```
OpenAL setup for general use (Linux)
------------------------------------
As of the July 2004 release of OpenAL it is best to add at least the
following line to your ~/.openalrc file on Linux because it wil find out
what audio backend to use, starting with the most appropriate:

(define devices '(native alsa sdl esd arts null))



ALSA surround sound (5.1) setup
------------------------------------
(taken from http://floam.ascorbic.com/how-to/alsa5.1)

Make a ~/.openalrc, we are telling OpenAL that we want surround sound and
we want to use ALSA instead of OSS.

(define speaker-num 4)
(define devices '(alsa))
(define alsa-out-device "surround40:0,0")



IRIX surround sound (5.1) setup
------------------------------------
```

To add 4 channel surround sound on IRIX hardware that supports in directly you can just add the following line to your ~/.openalrc file:

(define speaker-num 4)

To add 4 channel surround sound to IRIX systems that have more than one stereo output you can add the following section to your ~/.openalrc file (for a typical O2 configuration):

(define speaker-num 4)
(define native-out-device "Analog Out")
(define native-rear-out-device "Analog Out 2")

or alternatively:

(define speaker-num 4)
(define native-out-device "A3.Speaker")
(define native-rear-out-device "A3.LineOut2")

(Note the following section is obsolete as of the July 2004 release of OpenAL, since your could command OpenAL to use ALSA or Arts directly)

ALSA and Arts

I'm using kernel 2.6.5 with alsa, my sound module is snd-intel8x0. When I ran fgfs, I'd get quite 'choppy' sound (wasn't smooth, there'd be a couple of breaks in the sound every second or so). Running arts, and starting fgfs with "artsdsp fgfs" (from the artsdsp website: "When an application is run under artsdsp all accesses to the /dev/dsp audio device are intercepted and mapped into aRts API calls. While the device emulation is not perfect, most applications work this way, albeit with some degradation in performance and latency.") would improve the situation, but it seemed to still be choppy.

This command:
echo "fgfs 0 0 direct" >/proc/asound/card0/pcm0p/oss

(from the alsa kernel OSS emulation website:

"The direct option is used, as mentioned above, to bypass the automatic
  conversion and useful for MMAP-applications")

made my sound work beautifully when fgfs was run with artsdsp. Running without
artsdsp however (with artsd suspended or killed), would give me no sound at all
(which I find a bit strange)

The following websites might help people with similar troubles:
http://www.alsa-project.org/~iwai/OSS-Emulation.html
http://www.arts-project.org/doc/handbook/artsdsp.html

Computer info:

kernel 2.6.5

flightgear 0.9.4
simgear 0.3.5
plib 1.8.3

soundcard is onboard an asus p4p800-e deluxe mobo (using snd-intel8x0), alsa,
related modules from lsmod:

```
Module                    Size  Used by
snd_pcm_oss              53252  1
snd_mixer_oss            19968  1 snd_pcm_oss
snd_intel8x0             33476  1
snd_ac97_codec           63492  1 snd_intel8x0
snd_pcm                  97408  2 snd_pcm_oss,snd_intel8x0
snd_timer                26112  1 snd_pcm
snd_page_alloc           11396  2 snd_intel8x0,snd_pcm
snd_mpu401_uart           7936  1 snd_intel8x0
snd_rawmidi              24832  1 snd_mpu401_uart
snd_seq_device            8324  1 snd_rawmidi
snd                      53892  9 snd_pcm_oss,snd_mixer_oss,snd_intel8x0,
                                   snd_ac97_codec,snd_pcm,snd_timer,snd_mpu401_uart,
                                   snd_rawmidi,snd_seq_device
soundcore                10208  2 snd
```

33 Submodels

```
<?xml version="1.0"?>

<!--

Submodels are objects which can be dropped or launched from the user
aircraft. The trigger is a boolean property, which you define, which when
"true" causes the submodel to be released/launched.

A submodel will create an AIBallistic object which will follow a ballistic
path. By default one submodel will be released when the corresponding
trigger is "true".

Notes:
   1.  This utility is intended for ballistic objects which align to the
   trajectory. Drag is applied based on this assumption: no allowance is
   for changes in drag for objects which do not conform to this asumption.
   made

   2.  While Inertia is calculated properly, Moment of Inertia and rotational
   aerodynamic damping are simulated. It is assumed that the object is a cylinder
   of uniform density - if your object does not conform to this, there will be
   inaccuracies.

   3.  The program does not calculate windage for ballistic objects well. While
   adequate for smoke effects, etc., for bullets, bombs, droptanks this is probably
   best left at "False". Since the effects of wind on various ballistic objects is
   uncertain, there is no plan to change this situation.

   4. Submodels can be ensted to any depth, thus a submodel on expiry or impact etc,
   can launch a child submodel, which in turn can launch a submodel. and so on. This
   is the basis for Persistent Contrails, but any use is possible.

The initial conditions (IC) define the object's starting point (relative
to the user aircraft's "reported position"), and its initial speed and
direction (relative to the user aircraft).  If you want to release many
similar objects with similar IC, then you may use the <repeat>, <delay>
and <count> properties to define this.  The allowed properties are:
```

```
<name>          The name of the submodel.
<model>         The path to the visual model.
<trigger>       The property which will act as the trigger. If this tag is not
                included, the submodels will be released continuously, provided
                <count> is set to -1.
<speed>         Initial speed, in feet/sec, relative to user aircraft.
<speed-prop>    The property containing the Initial speed, in feet/sec, relative to
                user aircraft. If this path is found, <speed> will be overwritten.
<repeat>        Set "true" if you want multiple releases of this submodel.
<delay>         Time, in seconds, between repeated releases.
<count>         Number of submodels available for multiple release.
                -1 defines an unlimited number.
<slaved>        If true, the submodel is slaved to the parent model.
<x-offset>      Submodel's initial fore/aft position (in feet), relative to user
                aircraft.  Fore is positive.
<y-offset>      Submodel's initial left/right position (in feet), relative to user
                aircraft.  Right is positive.
<z-offset>      Submodel's initial up/down position (in feet), relative to user
                aircraft.  Up is positive.
<yaw-offset>    Submodel's initial azimuth, in degrees, relative to user
                aircraft'snose. Right is positive.
<pitch-offset>  Submodel's initial elevation, in degrees, relative to user aircraft's
                pitch. Up is positive.
<life>          Life span in seconds.
                Default is 900.0.
<buoyancy>      In ft/sec/sec.  Works opposite acceleration of gravity.
                For example, if set to 32 the submodel will feel no
                gravity. If greater than 32 the object will rise.
                Default is 0.
<wind>          Set to true if you want the submodel to react to the wind. Default
                is "false".
<cd>            The Coeffient of Drag. Varies with submodel shape - 0.295 for a
                bullet, 0.045 for an airfoil. Enter an appropriate value. Defaults to
                0.295.
<random>        When this is true the Cd is varied by +- 5%. Useful for smoke or
                contrails.
<eda>           Effective drag area (sq ft). Usually the cross-sectional area of the
                submodel normal to the airflow.
<weight>        The weight of the submodel (lbs). NOT set to 0 on submodel release.
                You may wish to set this value to 0 by means of key bindings or Nasal
```

```
                    script. Defaults to 0.25.
<contents>          The path to the contents of a submodel. The contents must be in lbs.
                    Intended for use with drop tanks. The property value will be set to
                    0 on release of the submodel: do not also set to 0 elsewhere e.g. in
                    key bindings. Defaults to 0.
<random>            Varies CD by +- 10%, initial azimuth by +- 10 degs, and life by
                    <randomness>
<randomness>        If <random> is true, <randomness> is applied to <life>. 0 > Value < 1
                    are valid. Defaults to 0.5.
<no-roll>           If true the submodel does not roll.
<impact>            If true, the impact location (lat/lon) on the terrain is calculated.
                    The Material (e.g Grass)of the terrain, load resistance, and impact
                    velocity. Altitude agl is calculated.
<collision>         If true, collisions with other objects is tested. If a collision is
                    detected then the position data are written to the "Report Node".
<fuze-range>        Used in detecting collisions. The distance in feet between an object
                    and a submodel at which a collision is deemed to have occurred.
<expiry>            If true, the current position of the submodel is written to the
                    "Report Node" when the submodel life expires.
<impact-reports> Defines a "Report Node". When an impact happens, then the path of
                    the submodel will be written to this node. An attached listener
                    function can evaluate the impact properties. If unset, reports go to
                    /ai/models/model-impact.

***** experimental ****

<external-force> If true the submodel is subjected to an external force
<force-path> A string describing the property where the magnitude, azimuth and
              elevation of the external force are to be found. The following child
               properties are instantiated:
                ~/force-lb
               ~/force-azimuth-deg
               ~/force-elevation-deg

You will have to set these values by some means (Nasal script etc.) to make use of th
utility.

<PropertyList>

  <submodel>
```

```
    <name>left gun</name>
    <model>Models/Geometry/tracer.ac</model>
    <trigger>ai/submodels/submodel[0]/trigger</trigger>
    <speed>2750.0</speed>
    <repeat>true</repeat>
    <delay>0.25</delay>
    <count>100</count>
    <x-offset>1.0</x-offset>
    <y-offset>-7.0</y-offset>
    <z-offset>-2.0</z-offset>
    <yaw-offset>0.4</yaw-offset>
    <pitch-offset>1.8</pitch-offset>
    <life>2.0</life>
</submodel>

<submodel>
    <name>right gun</name>
    <model>Models/Geometry/tracer.ac</model>
    <trigger>ai/submodels/submodel[0]/trigger</trigger>
    <speed>2750.0</speed>
    <repeat>true</repeat>
    <delay>0.25</delay>
    <count>100</count>
    <x-offset>1.0</x-offset>
    <y-offset>7.0</y-offset>
    <z-offset>-2.0</z-offset>
    <yaw-offset>-0.4</yaw-offset>
    <pitch-offset>1.8</pitch-offset>
    <life>2.0</life>
</submodel>

<submodel>
    <name>droptank-l</name>
    <model>Aircraft/Hunter/Models/droptank-100gal.ac</model>
    <trigger>controls/armament/station[0]/jettison-all</trigger>
    <speed>0</speed>
    <repeat>false</repeat>
    <count>1</count>
    <x-offset>0.820</x-offset>
    <y-offset>-9.61</y-offset>
```

```
    <z-offset>-2.39</z-offset>
    <yaw-offset>0</yaw-offset>
    <pitch-offset>0</pitch-offset>
    <wind>false</wind>
    <eda>2.11348887</eda>
    <weight>170</weight>
    <cd>0.045</cd>
    <contents>consumables/fuel/tank[2]/level-lbs</contents>
</submodel>

<submodel>
    <name>droptank-r</name>
    <model>Aircraft/Hunter/Models/droptank-100gal.ac</model>
    <trigger>controls/armament/station[1]/jettison-all</trigger>
    <speed>0</speed>
    <repeat>false</repeat>
    <count>1</count>
    <x-offset>0.820</x-offset>
    <y-offset>9.61</y-offset>
    <z-offset>-2.39</z-offset>
    <yaw-offset>0</yaw-offset>
    <pitch-offset>0</pitch-offset>
    <wind>false</wind>
    <eda>2.11348887</eda>
    <weight>170</weight>
    <cd>0.045</cd>
    <contents>consumables/fuel/tank[3]/level-lbs</contents>
</submodel>

<submodel>
    <name>engine exhaust r</name>
    <model>Aircraft/seahawk/Models/exhaust_s.xml</model>
    <trigger>sim/ai/aircraft/exhaust</trigger>
    <speed-node>engines/engine/n1</speed-node>
    <speed>10</speed>
    <repeat>true</repeat>
    <delay>0.1</delay>
    <count>-1</count>
    <x-offset>-3.5</x-offset>
    <y-offset>2.6768</y-offset>
```

```
      <z-offset>-0.3937</z-offset>
      <yaw-offset>170</yaw-offset>
      <life>10</life>
      <buoyancy>128</buoyancy>
      <aero-stabilised>0</aero-stabilised>
      <wind>true</wind>
      <eda>1</eda>
      <cd>0.95</cd>
      <weight>1</weight>
      <random>1</random>
   </submodel>

</PropertyList>

-->
```

34   Systems

```
By Default systems are initialized by the Aircraft/generic/generic-system.xml
This initializes the following:
- The generic electrical system
- 1 pitot system, index [0]
- 1 static system index [0]
- 2 vacuum systems [0] and [1], depending on engine rpm of engine[0] and
  engine[1] respectfully

If you want to define more systems, copy the generic-system file to your
aircraft-name/Systems folder and rename it systems.xml

In your aircraft -set file add the path to the system.xml file:
<sim>
    ....
    <systems>
        ....
        <path>Aircraft/aircraft-name/Systems/systems.xml</path>
        ....
    </systems>
    ....
</sim>
```

```
** Adding a second pitot system.
In your systems.xml, you should already have

  <pitot>
    <name>pitot</name>
    <number>0</number>
    <stall-deg>60</stall-deg>          # optionnal, default to 60 degrees
  </pitot>


and you need to add for a pitot system with index 1:

  <pitot>
    <name>pitot</name>
    <number>1</number>
    <stall-deg>60</stall-deg>  #optionnal
  </pitot>


For the any pitot system except for the first (with index 0)
add in the aircraft -set file (below for index 1):

<systems>
      <pitot n="1">
              <serviceable>1</serviceable>
      </pitot>
</systems>


Of course you can add a third or fourth etc.



** Adding a second static system
Absolutely analog with the pitot system. So add in systems.xml:

  <static>
    <name>static</name>
    <number>1</number>
    <tau>1</tau>
    <type>0</type>                      #optionnal: 0,1 or 2 default is 0
    <error-factor>0.5</error-factor>    #optionnal see below default = 0
```

```
        </static>
```

and in the aircraft -set file:

```
<systems>
      <static n="1">
                <serviceable>1</serviceable>
      </static>
</systems>
```

Now you can source your instrumentation relying on static and pitot
pressure (airspeed, altimeter, vertical speed indicator) from different
and independent systems

** The PITOT System

The pitot system measures impact pressure and is basically a tube pointing forward.
Small aircraft have one, small IFR aircraft have one or two (of which at least
one is heated) and larger commercial aircraft have three or even more. In those large
aircraft the left pitot serves the pilot instruments, the right the co-pilot and
the third system the back-up instruments. This might be different for each type
of aircraft of course.
In Flightgear the pitot system outputs the total pressure to the following property:
/systems/pitot[n]/total-pressure-inhg and
/systems/pitot[n]/measured-total-pressure-inhg
which are the same except at supersonic speeds. For supersonic aircraft use the "meas
property. See also the README.airspeed-indicator.
However it is advised for every aircraft to use the measured property. In future
this will be the property where all the measurement faults are reflected.
the following "measurement failures" are currently applied:
1) decrease of total and measured pressure due to side-slip and angle of attack
2) at 60 deg the pitot tube will stall and the value will fall back to static pressur
3) for the "measured" property only: at Mach>1, a shock wave is assumed in front of t
   pitot tube, decreasing the total pressure.

The stall angle may be (optionally) set to any angle between 0 and 90 deg
(default = 60 deg) like so:
```
  <pitot>
    <name>pitot</name>
```

```
      <number>0</number>
      <stall-deg>45</stall-deg>
   </pitot>
```

Both the decrease of the pitot pressure and the default stall angle are based on a me
on an AN5812 pitot tube.

** The STATIC system

The static system measures the static pressure. So all influences of airspeed are eli
In real life this is however not always easy. Effects from angle of attack, side-slip
gear extension, engine power setting and airspeed are present and for the aircraft
designer it is not alway easy to find a good position for the static port.
Usually the number of static systems are equal to the number of pitot systems.
In Flightgear there are 3 types of static systems modelled.
Type 0 (default): the perfect sensor. No measurement failures.
Type 1: Dual static ports on the fuselage sides. Side-slip angle influence only. this
Type 2: Static port on the pitot tube. Both angle of attack and side-slip influence.
If you want to use type 1 or 2:

```
   <static>
      <name>static</name>
      <number>0</number>
      <tau>0.1</tau>
      <type>1</type>
      <error-factor>0.5</error-factor>
   </static>
```

The output property /systems/static[n]/pressure-inhg is filtered. Therefore, if you w
the effect of the measurement failure, "tau" should be 0.1 or smaller.
The "error-factor" should be between 0.2 and 0.7. Setting it to 0 equals a "perfect s
A setting of 1 means the whole (projected on static port face) impact pressure is app
This is not realistic as usually there are more than one static pick-up points
and so the pressure increase gets "flattened".

35 Tutorials

== README.tutorials ==

FlightGear offers a flexible tutorial system, entirely written in the Nasal

language. Tutorials can be started and stopped from the "Help" menu. They are
defined in XML files. Each of them has to be loaded into /sim/tutorials/ under
a separate tutorial[n]/ branch:

```
<sim>
    <tutorials>
        <tutorial include="Tutorials/take-off.xml"/>
        <tutorial include="Tutorials/landing.xml"/>
    </tutorial>
</sim>
```

Alternatively, all tutorials can be defined in one file, with <tutorial> tags
around each tutorial. This is then included like so:

```
<sim>
    <tutorials include="foo-tutorials.xml"/>
</sim>
```

Finally, tutorials are automatically generated from any valid checklists
on startup. See README.checklists for details.

== TUTORIAL STRUCTURE ===

A tutorial has this structure, where some of the elements are described
in detail below:


```
<tutorial>
  <name>...</name>              mandatory; short identifier, also shown in the
                                    tutorial selection dialog
  <description>...</description> mandatory; longer description for the dialog
  <audio-dir>...</audio-dir>  optional; defines where to load sound samples
  <timeofday>noon</timeofday> optional; defines daytime; any of "dawn",
                                    "morning", "noon", "afternoon",
                                    "evening", "dusk", "midnight", "real"

  <step-time>                   optional; period between each step being executed.
                                Default 5
```

```
<exit-time>                     optional; period between exit/abort conditions being
                                checked. Default 1

<nasal>
    ...                         optional; initial Nasal code; see below
</nasal>

<models>
    ...                         optional; scenery objects; see below
</models>

<targets>
    ...                         optional; targets; see below
</targets>

<presets>
    ...                         optional; initial simulator state; see below
</presets>


<init>                          optional; initial settings; see below
  <set>
    ...                           optional; property settings; allowed multiple
  </set>                          times
  <view>
    ...                           optional; view settings
  </view>
  <marker>
    ...                           optional; marker coordinates
  </marker>
  <nasal>
    ...                           optional; Nasal code
  </nasal>
</init>

<step>                          mandatory; well, not really, but if there's not
                                at least one <step>, then the whole tutorial
                                won't do anything; see below for details

  <message>...</message>          optional; message to be displayed/spoken when
```

```
                                <step> is entered; allowed multiple times, in
                                which case one is chosen at random

<message-param>                 optional; allowed up to 4 times.
  <property>...</property>      property to substitute into the <message> string
</message-param>                using sprintf() formatting.  E.g. %d, %.2f

<audio>...</audio>              optional; file name of *.wav sample to be played;
                                        may be used multiple times (random)
<set>
    ...                         optional; allowed several times
</set>
<view>
    ...                         optional
</view>
<marker>
    ...                         optional
</marker>
<nasal>
    ...                         optional; Nasal code that is executed when the
</nasal>                                  step is entered

<wait>10</wait>                 optional; wait period after initial messages etc.

<error>                         optional; allowed several times
    <message>..</message>           optional; text displayed/spoken
    <audio>...</audio>              optional; name of *.wav sample to be played

    <condition>
        ...                         optional, but one should be there to make sense
    </condition>                            see $FG_ROOT/Docs/README.conditions

    <nasal>
        ...                         optional; Nasal code that is executed when the
    </nasal>                                error condition was fulfilled
</error>

<exit>                          optional; defines when to leave this <step>
    <condition>                         see $FG_ROOT/Docs/README.conditions
        ...
```

```
        </condition>

        <nasal>
            ...                        optional; Nasal code that is executed when the
        </nasal>                              exit condition was met
    </exit>
 </step>


  <end>                               optional; final settings & actions; see below
      <message>...</message>              optional; multiple times (random)
      <audio>...</audio>                  optional; multiple times (random)
      <set>
          ...                             optional
      </set>
      <view>
          ...                             optional
      </view>
      <nasal>
          ...                             optional
      </nasal>
  </end>
</tutorial>
```

After the tutorial has finished initialization, it goes through all <steps>.
For each it outputs the <message> or <audio>, optionally sets a <marker> and/or
a <view>, then it checks all <error>s and, if an <error><condition> is fulfilled,
outputs the respective <error><message>. If none of the <error>s occurred, then
it checks if the <exit><condition> is true, and if so, it jumps to the next
<step>. Otherwise the current <step> is endlessly repeated. Finally, after all
<step>s were processed, the <end> group is executed.

```
== ELEMENTS =========================================================


-- <nasal> -----------------------------------------------------------------

Embedded Nasal is supported on the top level, in <init> in each <step>, in a
<step>'s <error> and <exit>, and in <end>. All Nasal runs in a separate
namespace __tutorial, so it's possible to define a function in the <init>'s
Nasal block, and to use this function in other blocks without prefix. The
namespace is preloaded with some functions:


  next([n=1])       ... to switch n <step>s forward
  previous([n=1])   ... to switch n <step>s backwards

  say(what [, who="copilot [, delay=0]])
                    ... says 'what' with voice 'copilot' after 'delay' seconds


A Nasal group looks like this:

  <nasal>
      <script>
          say("Hi, I'm the pilot!", "pilot");
      </script>
      <module>__tutorial</module>           optional; preset with __tutorial
  </nasal>




-- <models> -----------------------------------------------------------------

This loads models into the scenery. It can be used to place, for example,
a helicopter landing pad at an airport where normally none is, so that the
tutorial can train landing. The layout is the following, with <path> being
relative to $FG_ROOT:
```

```
<models>
    <model>
        <path>Models/Airport/supacat_winch.ac</path>      mandatory
        <longitude-deg>-122.4950109</longitude-deg>        mandatory
        <latitude-deg>37.51403798</latitude-deg>           mandatory
        <elevation-ft>51</elevation-ft>                    mandatory
        <heading-deg>2.488888979</heading-deg>             optional (default: 0)
        <pitch-deg>0</pitch-deg>                           optional (default: 0)
        <roll-deg>0</roll-deg>                             optional (default: 0)
    </model>

    <model>
        ...                            another model
    </model>
</models>
```

The models are only removed before a new tutorial is loaded. Otherwise they
remain in the scenery for the whole FlightGear session. They aren't permanently
added.

-- <targets> --

These are simple pairs of longitude/latitude under an arbitrary name (here
"hospital" and "helipad"):

```
<targets>
    <hospital>
        <longitude-deg>-122.4950109</longitude-deg>        mandatory
        <latitude-deg>37.51403798</latitude-deg>           mandatory
    </hospital>

    <helipad>
        ...
    </helipad>
</targets>
```

The tutorial system will for each calculate how the user's aircraft is positioned relative to the respective target, and offer the information in this structure:

```
<sim>
    <tutorials>
        <targets>
            <hospital>
                <direction-deg>12.345</direction-deg>
                <heading-deg>33.333</heading-deg>
                <distance-m>12404.932</distance-m>
                <eta-min>39.2358</eta-min>
            </hospital>

            <helipad>
                ...
            </helipad>
        </targets>
    </tutorials>
</sim>
```

Where:
  <direction-deg> is an angle between the aircraft's velocity vector and the
                    azimuth to the target. 0 means that the aircraft is moving
                    right towards the target. 10 means that the target is slightly
                    to the right, -90 means that it's exactly left, and -180 or
                    179.9999 that it's right behind.

  <heading-deg>   is the absolute heading that the aircraft would currently
                    have to fly with in a straight line to reach the target

  <distance-m>   is the distance in meters

  <eta-min>   is the "Estimated Time of Arrival" given the aircraft's
                    current speed towards the target. Positive times mean that
                    the aircraft is getting nearer to the target and can arrive
                    there in this time given the current speed. It will, of course,
                    only arrive there, if <direction-deg> is zero. A negative
                    number means that the aircraft moves away, or in other words:

that in this number of minutes it will be away twice as far.

-- <presets> ---

These set the initial simulator state. All properties are optional.
The last three entries are to define the position relative to the
airport/runway or the longitude/latitude.

```
  <presets>
      <airport-id>KHAF</airport-id>
      <on-ground>1</on-ground>
      <runway>12</runway>
    <!--
      <altitude-ft>122.333</altitude-ft>
      <latitude-deg>37.555</latitude-deg>
      <longitude-deg>1000</longitude-deg>
    -->
      <heading-deg>0</heading-deg>
      <airspeed-kt>0</airspeed-kt>
      <glideslope-deg>0</glideslope-deg>
      <offset-azimuth>0</offset-azimuth>
      <offset-distance>0</offset-distance>
  </presets>
```

-- <set> --

<set> groups can be used in <init>, <step>, and <end>. They set a <property>
to a given <value> or to the value that a second <property> points to. They
can also reset values that were only temporarily changed for the duration
of the tutorial. This is desirable for properties that are saved to the
aircraft config file or to ~/.fgfs/autosave.xml.

```
<set>
    <property>/foo/bar</property>          set /foo/bar to 123
    <value>123</value>
</set>


<set>
    <property>/foo/bar</property>          set /foo/bar to value of /test
    <property>/test</property>
</set>
```

-- <view> ---

These groups can be used in <init>, <step>, and <end>. They smoothly move the
view to a new view position/direction. All parameters are optional. If, for
example, only <field-of-view> is set, then the view will only zoom in -- the
direction and position will remain the same. This feature is meant for cockpit
tutorials, where the pilot's view is directed to some switch or instrument.
view-number can be used to switch between different views, i.e. to tower-view,
copilot view etc. Default view-number is 0 (captain's view).

```
<view>
    <view-number>0</view-number>                0=captain's view, 1=copilot,...
    <heading-offset-deg>20</heading-offset-deg>  positive is left
    <pitch-offset-deg>-4</pitch-offset-deg>      positive is up
    <roll-offset-deg>0</roll-offset-deg>         positive is roll right
    <x-offset-m>0.2</x-offset-m>                 positive is move right
    <y-offset-m>0.2</y-offset-m>                 positive is move up
    <z-offset-m>0.2</z-offset-m>                 positive is move back
    <field-of-view>55</field-of-view>            default: 55; smaller zooms in
</view>                                                      bigger zooms out
```

-- <marker> ---

These are supported in `<init>`, `<step>`, and `<end>`. They show a magenta colored
circle at given position (relative to aircraft origin) in given size. See the
last section for how to conveniently find the proper coordinates.

```
  <marker>
      <x-m>1.3</x-m>                              positive is back
      <y-m>0.3</y-m>                              positive is to the right
      <z-m>0.1</z-m>                              positive is up
      <scale>1.3</scale>                          optional; default: 1
  </marker>
```

For this to work, the aircraft model needs to include the tutorial marker
model in its animation xml file:

```
  <PropertyList>
      <path>lightning-f1a.ac</path>

      <model>
          <path>Aircraft/Generic/marker.xml</path>
      </model>

      ...
  </PropertyList>
```

-- `<message>`/`<audio>` --

Groups `<step>` and `<end>` can have one or more `<message>` entries, and one or
more `<audio>` entries. If more are used of a kind, then the tutorial chooses
one at random. If `<audio>` are available, then the contents are interpreted as
file name of a *.wav sample, which is appended to the `<audio-dir>` path defined
at the `<tutorial>` top level (default: "") and played by the tutorial system.
Otherwise the `<message>` is handed over to the voice system, and synthesized
to speech by the Festival speech synthesizer (if installed). In either
case the chosen `<message>` is displayed on top of the screen. Neither `<message>`

nor <audio> are mandatory.

Because one and the same <message> string can be displayed *and* be synthesized,
which can be problematic in some cases, there is a way to specify parts for
either display *or* voice synthesizer: "{<display part>|<voice part}".
Example:

  <message>Press the {No1|number one} button!</message>

Here, "No1" would be displayed on the screen, but "number one" would be
sent to the speech synthesis system. This can also be used to add
invisible but audible exclamation marks: "Press the button{|!}"

-- <condition> --

These are explained in detail in $FG_ROOT/Docs/README.conditions. Here's just
one example:

  <condition>
      <less-than>
          <property>/foo/bar</property>
          <value>12</value>
      </less-than>
  </condition>

This condition is true when the value of /foo/bar is less than 12, and false
otherwise.

== FINDING MARKER COORDINATES ==

If an aircraft tutorial wants to use the marker, then the aircraft animation
file needs to include the marker model (see above). If this is done, then one
can use the "marker-adjust" dialog to find the respective <marker> coordinates.
Just type this into the "Help->Nasal Console" dialog:

```
tutorial.dialog()
```

Or temporarily add a key binding to the *-set.xml file:

```
<key n="96">
    <name>Backtick</name>
    <desc>Open marker adjust dialog</desc>
    <binding>
        <command>dialog-show</command>
        <dialog-name>marker-adjust</dialog-name>
    </binding>
</key>
```

The dialog allows to move a red cross around, which has the blinking marker
circle in the middle. Note that ctrl- and shift-modifiers modulate the slider
movements. Ctrl makes positioning coarser, and shift finer. The [Reset]
button moves the marker back to aircraft origin, the [Center] button centers
the sliders, and the [Dump] button dumps the marker coordinates to the
terminal, for example:

```
<marker>
    <x-m>1.1425</x-m>
    <y-m>0.1994</y-m>
    <z-m>-0.0844</z-m>
    <scale>2.0489</scale>
</marker>
```

This just needs to be copied to the tutorial XML file.

36 Wildfire

```
Cellular Automata based wildfire for FlightGear/CVS
---------------------------------------------------


Copyright (C) 2008 - 2009  Anders Gidenstam


 *  These programs are free software; you can redistribute them and/or modify
 *  them under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA


Usage
-----


A fire is started by calling wild_fire.ignite(pos) where pos is a
valid geo.Coord instance.


Example: starting fires by ctrl+shift+click:


Put this Nasal fragment somewhere where it is run at startup.
(E.g. in a <nasal><MyStuff><script>...</script></MyStuff></nasal>
block in preferences.xml.)


setlistener("/sim/signals/click", func {
  if (__kbd.shift.getBoolValue()) {
    if (__kbd.ctrl.getBoolValue()) {
      var click_pos = geo.click_position();
      wildfire.ignite(click_pos);
    }
  }
```

```
});
```

Configuration properties

These properties can be set at runtime, in preferences.xml or in any
other way supported by FlightGear.

/environment/wildfire/enabled : bool
  Enables/disables the whole WildFire module.
  On disable the current state is lost. Can be used to reset WildFire.

/environment/wildfire/share-events : bool
  Enables/disables sending and receiving of fire events over the
  multiplayer network.

/environment/wildfire/fire-on-crash : bool
  If true a fire will start if the aircraft crashes.

/environment/wildfire/report-score : bool
  Report the result of fire fighting.

/environment/wildfire/models/enabled : bool
  Enables/disables rendering of the 3d models.
  (That is, fire, smoke, soot and foam.)

/environment/wildfire/save-on-exit : bool
  If set the current log of Wildfire events is saved in
  ~/.fgfs/Wildfire/fire_log.xml .

/environment/wildfire/restore-on-startup : bool
  If set Wildfire will load and execute the events in
  ~/.fgfs/Wildfire/fire_log.xml . This recreates the fire state
  as it where when the log was saved.
  NOTE: A long event log or one that covers a long period of time will take
  a a lot of time to recreate.
  Storing and reloading of the CA state, as opposed to the event log, is not
  supported yet.

```
API
---

ignite : func (pos, source=1)
     pos    - fire location    : geo.Coord
     source - broadcast event? : {0, 1}

   Start a fire.


resolve_water_drop : func (pos, radius, volume, source=1)
     pos    - drop location    : geo.Coord
     radius - drop radius m    : double
     volume - Not used         : double
     source - broadcast event? : {0, 1}

  Extinguishes any fires in the cells within r of pos and
  makes the cells nonflammable.

resolve_retardant_drop : func (pos, radius, volume, source=1) {
     pos    - drop location    : geo.Coord
     radius - drop radius m    : double
     volume - Not used         : double
     source - broadcast event? : {0, 1}

  Identical to resolve_water_drop.

resolve_foam_drop : func (pos, radius, volume, source=1) {
     pos    - drop location    : geo.Coord
     radius - drop radius m    : double
     volume - Not used         : double
     source - broadcast?       : {0, 1}

  Extinguishes any fires in the cells within r of pos and
  makes the cells nonflammable and foamy.

load_event_log : func (filename, skip_ahead_until=-1)
     filename          - getprop("/sim/fg-home") ~ "/Wildfire/" ~ filename
     skip_ahead_until - skip from last event to this time : double (epoch)
```

```
                        fast forward from skip_ahead_until
                        to current time.
        x < last event   - fast forward all the way to current time (use 0).
                            NOTE: Can be VERY time consuming.
        -1               - skip to current time.


  Loads an event log.
  The skip_ahead_until argument can be used for synchronizing a restored
  fire state among multiple players.

save_event_log : func (filename)
     filename          - getprop("/sim/fg-home") ~ "/Wildfire/" ~ filename


  Saves an event log.

print_score = func
  Print a summary of the current wildfire state.



/Anders
```

37 Xmlhud

```
Users' Guide to FlightGear Hud configuration
December 22 2000
Neetha Girish <neetha@ada.ernet.in>


This document describes the reconfigurable HUD of
FlightGear implemented through XML config files.
The present reconfigurable HUD code uses most of the code
of version 0.6.1 vintage and I have adapted the same to provide
a reconfigurable HUD for fgfs.
Corrections and additions are welcome.

Some History:

Older versions of FGFS had a hard coded display of HUD.
This was a less than ideal state of affairs when it came to
using different aircraft Huds. I remember, somewhere in the 0.6.1 HUD code
```

it was written that the HUD code is 'presently' hard coded but ideally should
be moved into the aircraft configuration dataset, so that when you choose
an aircraft, its HUD loads.
This implementation make that possible, all you have to do is to
create appropriate 'my_aircraft.xml' files in the HUD directory and
without re-compiling the code you could have 'your_aircraft' HUD, by choosing that
in the .fgfsrc file or as a command line option as described later. Of course,
as of now, I have only implemented those HUD instruments in .xml readable form
as was available in version 0.7.6 + few more used by ADA, Bangalore for our
aircraft carrier take-off/landing simulation studies <www.flightgear.org/projects/ADA
To use the ADA specific reticles/HUD objects, please contact me/ you can figure it ou
yourself by studying the code. All of them are relevant 'only' if you use the conform
climb/dive ladder, since they are all referenced to it.
The rewrite of Hud display code was done using pre and post release v0.7.6 code
allowing for configuration of the hud via XML.

The present Configurable Hud implements the entire functionality of
fgfs HUD (called default HUD) till this date.

Using Default/Custom Hud:

The default HUD location is $FG_ROOT/Huds/Default.
$FG_ROOT is the place on your filesystem where you installed FG
data files. Alternate huds can be specified on the command line
or set as the default in the $HOME/.fgfsrc or $FG_ROOT/preferences.xml
using a property specification. The command line format is as follows:

--prop:/sim/hud/path=Huds/Default/default.xml

The path description shown is relative to $FG_ROOT. An absolute
path may also be used for locations outside $FG_ROOT.
For the custom Hud the path will be Huds/Custom/default.xml

Hud - Implementation:

All of the hud configuration files are XML-encoded property lists.
The root element of each file is always named <PropertyList>. Tags are
always found in pairs, with the closing tag having a slash prefixing
the tag name, i.e </PropertyList>. The top level panel configuration

file is composed of a <name> and zero or more <instruments>.
Instruments are used by including a <"unique_name"> and a <path> to the
instruments configuration file.
Comments are bracketed with <!-- -->.

     Example Top Level Hud Config

```
<PropertyList>
 <name>Default Aircraft Hud</name>
 <instruments>

  <hudladder>                      <!--unique name -->
   <path>Huds/Instruments/Default/hudladder.xml</path>
  </hudladder>

  <hudcard>
   <path>Huds/Instruments/Default/hudcard.xml</path>
  </hudcard>

  <instrlabel>
   <path>Huds/Instruments/Default/instrlabel.xml</path>
  </instrlabel>

  <fgTBI>
   <path>Huds/Instruments/Default/fgtbi.xml</path>
  </fgTBI>

 </instruments>
</PropertyList>
```

The default location for instrument files is $FG_ROOT/Huds/Instruments/Default.
The location for custom instrument files is $FG_ROOT/Huds/Instruments/Custom.
The location for minimal instrument files is $FG_ROOT/Huds/Instruments/Minimal.
Alternate locations may be specified in the hud configuration, paths
must be absolute to use files outside $FG_ROOT.

About Instrument Placement:

For the sake of simplicity the FGFS HUD overlay is always 640 x 480 res.

so all x/y values for instrument placement should fall within these bounds.
Being an OpenGL program, 0,0 represents the lower left hand corner of the
screen.

Instrument Implementation:

Instruments are defined in separate configuration files.
The Instruments are basically classified into 4 types(Each of them an xml file) :
The Hud Ladder,
The Hud Card,
The Hud Label and
The Turn Bank Indicator
.... (Note that that the earlier HUD classes/objects have been retained)
Newer objects may be instantiated using the above classes, Unless a totally
new object is required).

The Default as well as the Custom directory have the same (in terms of properties)
set of configuration files (but with different values to suit the aircraft).

We have a Base class – Hud Instrument Item.
We derive two more base classes – Instrument Scale and Dual Instrument Item from this
(This implementation owes its existence to all those who wrote the HUD code for 0.6.1

The Hud Instrument Label is an instantiable class derived from Hud Instrument Item –
for displaying alphanumeric labels (altitude, velocity, Mach no and/or anything else
as long you have a call back function to pass the value using the property
'data_source').

The Hud Card is an instantiable class derived from Instrument scale – for displaying
tapes and gauges (single variable display, for displaying aoa, g's, vsi,
elevator_posn, etc.).

The Hud Ladder is an instantiable class derived from Dual Instrument Item – for
displaying pitch reference ladder or climb/dive ladder (two variable display, for
displaying two types of ladders, the pitch reference ladder or the climb/dive ladder
as defined by MIL-1787b).

The fgTBI Instrument is an instantiable class derived from Dual Instrument scale
again – for display of Bank angle and Sideslip (two variable display, for display
of TSI info, kept different from the two variable ladder object basically because

of its totally different draw member function).

Most Hud instruments may be instantiated using above. It is proposed to provide all
Hud objects as defined in MIL-STD-1797A, soon.

Here is how you position 'any' object:

```
x _____ x+width
  |          |
  |          |
y ----------- y+height
```

this defines the objects position centered about the centroid of above rectangle
in HUD overlay plane (640x480) coordinates with 0,0 at bottom-left corner.

One more, pixels per degree in the ladder class represents the compression factor of
the pitch ladder. In case of conformal HUD (climb/dive ladder) it is
<640/horizontal_fov> or <480/vertical_fov>. In case of pitch reference ladder it is
<your_no_of vertical_pixels/your_no_of_ladder_degrees>.

Example of Hud Ladder xml file.

```
<PropertyList>

 <ladders>

  <l1>
  <name>Pitch Ladder</name>        <!-- Name can be Pitch Ladder or Climb/Dive Ladder
  <x>260</x>                       <!-- x start -->
  <y>150</y>                       <!-- y start -->
  <width>120</width>               <!-- x start + width = x end -->
  <height>180</height>             <!-- y start + height = y end -->
  <compression_factor>2.68</compression_factor>   <!-- Pixels per degree -->
  <loadfn>roll</loadfn>            <!-- Name of the function to be called, here
                                       get_roll() is called provision made in Hud.cxx
  <loadfn1>pitch</loadfn1>         <!-- Name of the function to be called, here get_pitc
                                       is called -->
  <span_units>45.0</span_units>        <!-- Range of the Ladder seen at any instant
  <division_units>10.0</division_units> <!-- Divisions -->
  <screen_hole>70</screen_hole>        <!-- Hole b/w the Ladder Bars -->
```

```
    <lbl_pos>0</lbl_pos>                        <!-- Label Position to indicate pitch angle o
    <enable_frl>false</enable_frl>              <!-- To Enable Pitch Reference Symbol (used b
    <enable_target_spot>true</enable_target_spot>                    <!-- To Enable Target
    <enable_velocity_vector>false</enable_velocity_vector>          <!-- To Enable Velocit
    <enable_drift_marker>false</enable_drift_marker>                <!-- To Enable Drift M
    <enable_alpha_bracket>false</enable_alpha_bracket>              <!-- To Enable Alpha E
    <enable_energy_marker>false</enable_energy_marker>              <!-- To Enable Energy
    <enable_climb_dive_marker>false</enable_climb_dive_marker>      <!-- To Enable Climb/D
    <enable_glide_slope_marker>false</enable_glide_slope_marker>    <!-- To Enable Glide/S
    <glide_slope>0.0</glide_slope>                                  <!-- Glide slope angle
    <enable_energy_worm>false</enable_energy_worm>                  <!-- To Enable Energy
    <enable_waypoint_marker>false</enable_waypoint_marker>          <!-- To Enable Way poi
    <working>true</working>                                         <!-- use this to enabl
  </l1>

 </ladders>

</PropertyList>

Before you read this, _____ this is tick_top          |                   |
                         |                            |                   |
                         |                            |_____|
                         | this is cap_right,tick_left  cap_bottom    tick_right
                         |
                   _____| this is tick_bottom


Example of Hud Card xml file.

<PropertyList>

 <cards>

  <c1>
   <name>Gyrocompass</name>
   <x>220</x>
   <y>430</y>
   <width>200</width>
   <height>28</height>
   <loadfn>heading</loadfn>      <!-- Name of the function to be called, here get_Heac
   <options>4</options>          <!--  Read Tape Options Below or Hud.hxx file for det
```

```xml
      <maxValue>360.0</maxValue>   <!-- Maximum scale value -->
      <minValue>0.0</minValue>      <!-- Minimum Scale Value -->
      <disp_scaling>1.0</disp_scaling> <!-- Multiply by this to get numbers shown on sca
      <major_divs>5</major_divs>        <!-- major division marker units -->
      <minor_divs>1</minor_divs>        <!-- minor division marker units -->
      <modulator>360</modulator>        <!-- Its a rose, Roll Over Point -->
      <value_span>25.0</value_span>     <!-- Range Shown  -->
      <type>tape</type>                 <!-- Card type can be "tape" or "gauge" -->
      <tick_bottom>false</tick_bottom> <!-- Read Ticks and Caps below -->
      <tick_top>false</tick_top>
      <tick_right>true</tick_right>
      <tick_left>true</tick_left>
      <cap_bottom>true</cap_bottom>
      <cap_top>false</cap_top>
      <cap_right>false</cap_right>
      <cap_left>false</cap_left>
      <marker_offset>0.0</marker_offset>    <!-- Read Marker offset below -->
      <enable_pointer>true</enable_pointer> <!-- To draw a pointer -->
      <pointer_type>fixed</pointer_type>    <!-- Type of pointer, Fixed or Moving (yet t
      <working>true</working>
   </c1>

  </cards>

</PropertyList>
```

Tape Options:

```
HUDS_AUTOTICKS   =      0x0001
HUDS_VERT        =      0x0002
HUDS_HORZ        =      0x0000
HUDS_TOP         =      0x0004
HUDS_BOTTOM      =      0x0008
HUDS_LEFT        =      HUDS_TOP
HUDS_RIGHT       =      HUDS_BOTTOM
HUDS_BOTH        =      (HUDS_LEFT | HUDS_RIGHT)
HUDS_NOTICKS     =      0x0010
HUDS_ARITHTIC    =      0x0020
HUDS_DECITICS    =      0x0040
HUDS_NOTEXT      =      0x0080
```

```
HUDS_LEFT  | HUDS_VERT    = 0x0006
HUDS_RIGHT | HUDS_VERT    = 0x0010
HUDS_TOP   | HUDS_NOTEXT  = 0x0084
HUDS_BOTTOM | HUDS_NOTEXT = 0x0088
HUDS_VERT  | HUDS_LEFT  | HUDS_NOTEXT  = 0x0086
HUDS_RIGHT | HUDS_VERT  | HUDS_NOTEXT = 0x0090
```

For clarity, I repeat, Ticks and Caps :

```
1. Left Tick ->|_____|<- Right Tick
                  |
                  v
            Bottom Cap

2.     Top Cap
         |
         v
       _____
      |     |

3.     Top Tick
         |
         v
       ---
          |
          |<- Right Cap
          |
       ---
         |
         v
      Bottom Tick

4.             ---
                |
  Left Cap ->  |
                |
               ---
```

221

Marker Offset :

To Draw pointer on the scale markings. In the case of a our hud with offset 10.0
The pointer is away from the scale and points at the markings.

```
        -|                        -
        -|                        -
        <|                       -<
        -|                        -
        -|                        -
Marker offset = 0.0      Marker offset = 10.0
```

This should be useful when I implement the fixed tape/moving pointer.

Example of a Label xml file.

```
<PropertyList>

 <labels>

  <i1>
   <name>machno</name>
   <x>25</x>
   <y>130</y>
   <width>40</width>
   <height>30</height>
   <data_source>mach</data_source>      <!-- Name of the function to be called, here ge
   <label_format>%4.2f</label_format> <!-- The Label Format -->
   <pre_label_string>blank</pre_label_string>  <!-- String to be written Pre Label --
   <post_label_string>NULL</post_label_string> <!-- String to be written Post Label -
   <scale_data>1.0</scale_data>
   <options>4</options>                  <!-- Read Tape options or Hud.hxx -->
   <justification>2</justification>   <!-- Justify the label, 0=LEFT_JUSTIFY, 1=CENTE
   <blinking>0</blinking>                <!-- Yet to be implemented -->
   <working>true</working>
   <latitude>false</latitude>            <!-- True if the label is to display Latitude
   <longitude>false</longitude>          <!-- True if the label is to display Longitude
  </i1>
```

```
   </labels>

</PropertyList>


Example of a Turn Bank Indicator xml file.

<PropertyList>

 <tbis>

  <f1>
  <name>fgTBI_Instrument</name>
  <x>290</x>
  <y>45</y>
  <width>60</width>
  <height>10</height>
  <loadfn>roll</loadfn>        <!-- Name of the function to be called, get_roll() is
  <loadfn1>sideslip</loadfn1> <!-- Name of the function to be called, get_sideslip()
  <maxBankAngle>45.0</maxBankAngle> <!-- Maximum Angle of Bank -->
  <maxSlipAngle>5.0</maxSlipAngle>  <!-- Maximum Angle of Slip -->
  <gap_width>5</gap_width>          <!-- Screen Hole -->
  <working>true</working>
  </f1>

 </tbis>

</PropertyList>


I have still got to implement dials (as in MIL-STD-1787b).

REMEMBER IF YOU NEED TO INDICATE ANY OTHER PARAMETER ON THE HUD OTHER THAN WHAT IS PF
CALLBACK FUNCTIONS (PROPERTY NAMES LISTED BELOW) YOU HAVE TO FIDDLE WITH THE CODE, AS
KNOW AND I SHALL INCLUDE THAT.

<loadfn>anzg</loadfn>                <!-- Here get_anzg() is called -->
<loadfn>heading</loadfn>             <!-- Here get_heading() is called -->
<loadfn>aoa</loadfn>                 <!-- Here get_aoa() is called -->
<loadfn>climb</loadfn>               <!-- Here get_climb() is called -->
```

```xml
<loadfn>altitude</loadfn>          <!-- Here get_altitude() is called -->
<loadfn>agl</loadfn>               <!-- Here get_agl() is called -->
<loadfn>speed</loadfn>             <!-- Here get_speed() is called -->
<loadfn>view_direction</loadfn>    <!-- Here get_view_direction() is called -->
<loadfn>aileronval</loadfn>        <!-- Here get_aileronval() is called -->
<loadfn>elevatorval</loadfn>       <!-- Here get_elevatorval() is called -->
<loadfn>rudderval</loadfn>         <!-- Here get_rudderval() is called -->
<loadfn>throttleval</loadfn>       <!-- Here get_throttleval() is called -->
<loadfn>aux16</loadfn>             <!-- Here get_aux16() is called -->
<loadfn>aux17</loadfn>             <!-- Here get_aux17() is called -->
<loadfn>aux9</loadfn>              <!-- Here get_aux9() is called -->
<loadfn>aux11</loadfn>             <!-- Here get_aux11() is called -->
<loadfn>aux12</loadfn>             <!-- Here get_aux12() is called -->
<loadfn>aux10</loadfn>             <!-- Here get_aux10() is called -->
<loadfn>aux13</loadfn>             <!-- Here get_aux13() is called -->
<loadfn>aux14</loadfn>             <!-- Here get_aux14() is called -->
<loadfn>aux15</loadfn>             <!-- Here get_aux15() is called -->
<loadfn>aux8</loadfn>              <!-- Here get_aux8() is called -->
<loadfn>ax</loadfn>                <!-- Here get_Ax() is called -->
<loadfn>mach</loadfn>              <!-- Here get_mach() is called -->
<loadfn>framerate</loadfn>         <!-- Here get_frame_rate() is called -->
<loadfn>fov</loadfn>               <!-- Here get_fov() is called -->
<loadfn>vfc_tris_culled</loadfn>   <!-- Here get_vfc_tris_culled() is called -->
<loadfn>vfc_tris_drawn</loadfn>    <!-- Here get_vfc_tris_drawn() is called -->
<loadfn>latitude</loadfn>          <!-- Here get_latitude() is called -->
<loadfn>longitude</loadfn>         <!-- Here get_longitude() is called -->
```

38   Xmlpanel

Users Guide to FlightGear panel configuration
Version 0.7.7, May 16 2001
Author: John Check <j4strngs@rockfish.net>

This document is an attempt to describe the configuration of
FlightGear flight simulator's aircraft panel display via XML. The
information was culled from the fgfs-devel@flightgear.org mailing list
and my experiences making alternate panels. Corrections and additions
are encouraged.

Some History:

Older versions of FGFS had a hard coded display of instruments. This
was a less than ideal state of affairs due to FGFS ability to use
different aircraft models. Being primarily developed on UNIX type
systems, a modular approach is taken towards the simulation. To date,
most alternatives to the default Cessna 172 aircraft are the product
of research institutions interested in the flight characteristics and
not cosmetics. The result of this was that one could fly the X-15 or
a Boeing 747 but be limited to C172 instrumentation.

A rewrite of the panel display code was done around v0.7.5 by
developer David Megginson allowing for configuration of the panel via
XML to address this limitation. Some major changes and additions were
made during the course of version 0.7.7 necessitating a rewrite and
expansion of this document.

About The Property Manager:

While not absolutely necessary in order to create aircraft panels,
some familiarity with the property manager is beneficial....
FlightGear provides a hierarchical representation of all aspects of
the state of the running simulation that is known as the property
tree. Some properties, such as velocities are read only. Others such
as the frequencies to which the navcom radios are tuned or the
position of control surfaces can be set by various means. FlightGear
can optionally provide an interface to these properties for external

225

applications such as Atlas, the moving map program, or even lowly
telnet, via a network socket. Data can even be placed on a serial port
and connected to, say a GPS receiver. Aside from its usefulness in a
flight training context, being able to manipulate the property tree on
a running copy of FG allows for switching components on the fly, a
positive boon for panel authors. To see the property tree start FG
with the following command line:

```
fgfs --props=socket,bi,5,localhost,5500,tcp
```

Then use telnet to connect to localhost on port 5500. You can browse
the tree as you would a filesystem.

XML and the Property Manager:

Panel instruments interface with the property tree to get/set values
as appropriate. Properties for which FG doesn't yet provide a value
can be created by simply making them up. Values can be adjusted using
the telnet interface allowing for creation and testing of instruments
while code to drive them is being developed.

If fact, the XML configuration system allows a user to combine
components such as flight data model, aircraft exterior model, heads
up display, and of course control panel. Furthermore, such a
preconfigured aircraft.xml can be included into a scenario with
specific flight conditions. These can be manually specified or a FG
session can be saved and/or edited and reloaded later. Options
specified in these files can be overridden on the command line. For
example:

```
--prop:/sim/panel/path=Aircraft/c172/Panels/c172-panel.xml
```

passed as an option, would override a panel specified elsewhere.
Property tree options all have the same format, specify the node and
supply it a value.

The order of precedence for options is thus:

Source   Location   Format
------   --------   ------

```
command line
.fgfsrc          Users home directory.   command line options
system.fgfsrc    $FG_ROOT                 ""        ""
preferences.xml  $FG_ROOT                 XML property list
```

Loading Panels on the fly:

When editing a panel configuration, pressing Shift +F3 will reload the
panel. If your changes don't seem to be taking effect, check the
console output. It will report the success or failure of the panel
reload*. Editing textures requires restarting FGFS so the new textures
can be loaded. Panels can be switched on the fly by setting the
/sim/panel/path property value and reloading.

Regarding Window Geometry:

For the sake of simplicity the FGFS window is always considered to be
1024x768 so all x/y values for instrument placement should relative to
these dimensions. Since FG uses OpenGL 0,0 represents the lower left
hand corner of the screen. Panels may have a virtual size larger than
1024x768. Vertical scrolling is accomplished with
Shift+F5/F6. Horizontal scrolling is via Shift+F7/F8. An offset should
be supplied to set the default visible area. It is possible to place
items to overlap the 3D viewport.

Panel Architecture:

All of the panel configuration files are XML-encoded* property lists.
The root element of each file is always named <PropertyList>. Tags are
almost always found in pairs, with the closing tag having a slash
prefixing the tag name, i.e </PropertyList>. The exception is the tag
representing an aliased property. In this case a slash is prepended to
the closing angle bracket. (see section Aliasing)

The top level panel configuration file is composed of a <name>, a
<background> texture and zero or more <instruments>.Earlier versions
required instruments to have a unique name and a path specification
pointing to the instruments configuration file.

```
[ Paths are relative to $FG_ROOT (the installed location of FGFS data files.) ]
[ Absolute paths may be used.Comments are bracketed with <!-- -->.            ]

Old style instrument call in top level panel.xml:
--------------------------------------------------
  <clock>             <!-- required "unique_name" -->
   <path>Aircraft/c172/Instruments/clock.xml</path>
   <x>110</x>         <!-- required horizontal placement -->
   <y>320</y>         <!-- required vertical placement -->
   <w>72</w>          <!-- optional width specification -->
   <h>72</h>          <!-- optional height specification -->
  </clock>


The difference between the old and new styles, while subtle, is rather
drastic.  The old and new methods are indeed incompatible. I cover the
old style only to acknowledge the incompatibility. This section will
be removed after the next official FGFS release.


New Style Example Top Level Panel Config:
-----------------------------------------
<PropertyList>
 <name>Example Panel</name>
 <background>Aircraft/c172/Panels/Textures/panel-bg.rgb</background>
 <w>1024</w>                        <!-- virtual width -->
 <h>768</h>                         <!-- virtual height -->
 <y-offset>-305</y-offset>          <!-- hides the bottom part -->
 <view-height>172</view-height>     <!-- amount of overlap between 2D panel
                                          and 3D viewport -->

 <instruments>                      <!-- from here down is where old and new
                                          styles break compatibility -->

  <instrument include="../Instruments/clock.xml">
   <name>Chronometer</name>   <!-- currently optional but strongly recommended -->
   <x>150</x>                 <!-- required horizontal placement -->
   <y>645</y>                 <!-- required vertical placement -->
   <w>72</w>                  <!-- optional width specification -->
   <h>72</h>                  <!-- optional height specification -->
  </instrument>
```

```
    </instruments>

</PropertyList>


Indexed Properties
------------------
This is a lot to do with the compatibility break so lets get it out of
the way.  The property manager now assigns incremental indices to
repeated properties with the same parent node, so that

 <PropertyList>
 <x>1</x>
 <x>2</x>
 <x>3</x>
 </PropertyList>

shows up as

 /x[0] = 1
 /x[1] = 2
 /x[2] = 3


This means that property files no longer need to make up a separate
name for each item in a list of instruments, layers, actions,
transformations, or text chunks. In fact, the new panel I/O code now
insists that every instrument have the XML element name "instrument",
every layer have the name "layer", every text chunk have the name
"chunk", every action have the name "action", and every transformation
have the name "transformation" -- this makes the XML more regular (so
that it can be created in a DTD-driven tool) and also allows us to
include other kinds of information (such as doc strings) in the lists
without causing confusion.

Inclusion:
----------
The property manager now supports file inclusion and aliasing.
Inclusion means that a node can include another property file as if it
were a part of the current file.  To clarify how inclusion works,
consider the following examples:
```

If bar.xml contains

```
<PropertyList>
<a>1</a>
<b>
<c>2</c>
</b>
</PropertyList>
```

then the declaration

```
<foo include="../bar.xml">
</foo>
```

is exactly equivalent to

```
<foo>
<a>1</a>
<b>
<c>2</c>
</b>
</foo>
```

However, it is also possible to selectively override properties in the included file. For example, if the declaration were

```
<foo include="../bar.xml">
<a>3</a>
</foo>
```

then the property manager would see

```
<foo>
<a>3</a>
<b>
<c>2</c>
</b>
</foo>
```

with the original 'a' property's value replaced with 3.

This new inclusion feature allows property files to be broken up and
reused arbitrarily -- for example, there might be separate cropping
property lists for commonly-used textures or layers, to avoid
repeating the information in each instrument file.

Aliasing

Properties can now alias other properties, similar to a symbolic link
in Unix. When the target property changes value, the new value will
show up in the aliased property as well. For example,

```
 <PropertyList>
 <foo>3</foo>
 <bar alias="/foo"/>
 </PropertyList>
```

will look the same to the application as

```
 <PropertyList>
 <foo>3</foo>
 <bar>3</bar>
 </PropertyList>
```

except that when foo changes value, bar will change too.

The combination of inclusions and aliases is very powerful, because it
allows for parameterized property files. For example, the XML file for
the NAVCOM radio can include a parameter subtree at the start, like
this:

```
 <PropertyList>
 <params>
 <comm-freq-prop>/radios/comm1/frequencies/selected</comm-freq-prop>
 <nav-freq-prop>/radios/nav1/frequencies/selected</comm-freq-prop>
 </params>
```

```
 ...

 <chunk>
 <type>number-value</type>
 <property alias="/params/nav-freq-prop"/>
 </chunk>


 ...
 </PropertyList>
```

Now, the same instrument file can be used for navcomm1 and navcomm2,
for example, simply by overriding the parameters at inclusion:

```
 <instrument include="../Instruments/navcomm.xml">
 <params>
 <comm-freq-prop>/radios/comm1/frequencies/selected</comm-freq-prop>
 <nav-freq-prop>/radios/nav1/frequencies/selected</comm-freq-prop>
 </params>
 </instrument>


 <instrument include="../Instruments/navcomm.xml">
 <params>
 <comm-freq-prop>/radios/comm2/frequencies/selected</comm-freq-prop>
 <nav-freq-prop>/radios/nav2/frequencies/selected</comm-freq-prop>
 </params>
 </instrument>
```

Instrument Architecture:

Instruments are defined in separate configuration files. An instrument
consists of a base width and height, one or more stacked layers, and
zero or more actions. Base dimensions are specified as follows:

```
<PropertyList>                  <!-- remember, all xml files start like this -->
 <name>Airspeed Indicator</name> <!-- names are good -->
 <w-base>128</w-base>           <!-- required width spec-->
 <h-base>128</h-base>           <!-- required height spec-->
  <layers>                      <!-- begins layers section -->
```

Height and width can be overriden in the top level panel.xml by

specifying <w> and <h>. Transformations are caculated against the base
size regardless of the display size. This ensures that instruments
remain calibrated

Textures:

FG uses red/green/blue/alpha .rgba files for textures. Dimensions for
texture files should be power of 2 with a maximum 8:1 aspect ratio.
The lowest common denominator for maximum texture size is 256 pixels.
This is due to the limitations of certain video accelerators, most
notably those with 3Dfx chipset such as the Voodoo2.

Instrument Layers**:

The simplest layer is a <texture>. These can be combined in <switch> layers

<texture>
A texture layer looks like this:

```
  <layer>                          <!-- creates a layer -->
   <name>face</name>
   <texture>                       <!-- defines it as a texture layer -->
    <path>Aircraft/c172/Instruments/Textures/faces-2.rgb</path>
    <x1>0</x1>                      <!-- lower boundary for texture cropping-->
    <y1>0.51</y1>                   <!-- left boundary  for texture cropping-->
    <x2>0.49</x2>                   <!-- upper boundary  for texture cropping-->
    <y2>1.0</y2>                    <!-- right boundary  for texture cropping-->
   </texture>                       <!-- closing texure tag -->
  </layer>                          <!-- closing layer tag -->
```

The texture cropping specification is represented as a decimal. There
is a table at the end of this document for converting from pixel
coordinates to percentages.

This particular layer, being a gauge face has no transformations
applied to it. Layers with that aren't static *must* include <w> and
<h> parameters to be visible.

<type> May be either text or switch..

```
<type>switch</type>
A switch layer is composed of two or more nested layers and will
display one of the nested layers based on a boolean property. For a
simple example of a switch see
$FG_ROOT/Aircraft/c172/Instruments/brake.xml.

  <layer>
   <name>Brake light</name>
   <type>switch</type>                          <!-- define layer as a switch -->
   <property>/controls/brakes</property>  <!-- tie it to a property -->
    <layer1>                                    <!-- layer for true state -->
     <name>on</name>                            <!-- label to make life easy -->
     <texture>                                  <!-- layer1 of switch is a texture layer --
     <path>Aircraft/c172/Instruments/Textures/brake.rgb</path>
     <x1>0.25</x1>
     <y1>0.0</y1>
     <x2>0.5</x2>
     <y2>0.095</y2>
     </texture>
     <w>64</w>                                  <!-- required width - layer isn't static -->
     <h>24</h>                                  <!-- required height - layer isn't static --
    </layer1>                                   <!-- close layer1 of switch -->
    <layer2>                                    <!-- layer for false state -->
     <name>off</name>
     <texture>
     <path>Aircraft/c172/Instruments/Textures/brake.rgb</path>
     <x1>0.0</x1>
     <y1>0.0</y1>
     <x2>0.25</x2>
     <y2>0.095</y2>
     </texture>
     <w>64</w>
     <h>24</h>
    </layer2>
  </layer>
```

Switches can have more than 2 states. This requires nesting one switch
inside another. One could make, for example, a 3 color LED by nesting
switch layers.

```
<type>text</type>
A text layer may be static, as in a label, generated from a property
or a combination of both.  This example is a switch that contains both
static and dynamic text:

 <layer1>                             <!-- switch layer -->
  <name>display</name>
  <type>text</type>                   <!-- type == text -->
  <point-size>12</point-size>    <!-- font size -->
  <color>                             <!-- specify rgb values to color text -->
   <red>1.0</red>
   <green>0.5</green>
   <blue>0.0</blue>
  </color>                            <!-- close color section -->
  <chunks>                            <!-- sections of text are referred to as chunks --
   <chunk>                            <!-- first chunk of text -->
    <type>number-value</type>       <!-- value defines it as dynamic -->
    <property>/radios/nav1/dme/distance</property>    <!-- ties it to a property -->
    <scale>0.00053995680</scale>  <!-- convert between statute and nautical miles? --
    <format>%5.1f</format>        <!-- define format -->
   </chunk>
  </chunks>
 </layer1>
 <layer2>                             <!-- switch layer -->
  <name>display</name>
  <type>text</type>                   <!-- type == text -->
  <point-size>10</point-size>    <!-- font size -->
  <color>                             <!-- specify rgb values to color text -->
   <red>1.0</red>
   <green>0.5</green>
   <blue>0.0</blue>
  </color>                            <!-- close color section -->
  <chunks>                            <!-- sections of text are referred to as chunks --
   <chunk>                            <!-- first chunk of text -->
    <type>literal</type>         <!-- static text -->
    <text>---.--</text>          <!-- fixed value -->
   </chunk>
  </chunks>
 </layer2>
```

Transformations:

A transformation is a rotation, an x-shift, or a
y-shift. Transformations can be static or they can be based on
properties. Static rotations are useful for flipping textures
horizontally or vertically. Transformations based on properties are
useful for driving instrument needles. I.E. rotate the number of
degrees equal to the airspeed. X and y shifts are relative to the
center of the instrument. Each specified transformation type takes an
<offset>. Offsets are relative to the center of the instrument. A
shift without an offset has no effect. For example, let's say we have
a texure that is a circle. If we use this texture in two layers, one
defined as having a size of 128x128 and the second layer is defined as
64x64 and neither is supplied a shift and offset the net result
appears as 2 concentric circles.

About Transformations and Needle Placement:

When describing placement of instrument needles, a transformation
offset must be applied to shift the needles fulcrum or else the needle
will rotate around it's middle. The offset will be of <type> x-shift
or y-shift depending on the orientation of the needle section in the
cropped texture.

This example comes from the altimeter.xml

```
  <layer>
   <name>long needle (hundreds)</name>  <!-- the altimeter has more than one needle
   <texture>
    <path>Aircraft/c172/Instruments/Textures/misc-1.rgb</path>
    <x1>0.8</x1>
    <y1>0.78125</y1>
    <x2>0.8375</x2>
    <y2>1.0</y2>
   </texture>
   <w>8</w>
   <h>56</h>
```

```
   <transformations>                          <!-- begin defining transformations -->
    <transformation>                          <!-- start definition of
                                                  transformation that drives the needle --
     <type>rotation</type>
     <property>/steam/altitude</property>  <!-- bind it to a property -->
     <max>100000.0</max>                      <!-- upper limit of instrument -->
     <scale>0.36</scale>                      <!-- once around == 1000 ft -->
    </transformation>                         <!-- close this transformation -->
    <transformation>                          <!-- shift the fulcrum of the needle -->
     <type>y-shift</type>                     <!-- y-shift relative to needle -->
     <offset>24.0</offset>                    <!-- amount of shift -->
    </transformation>
   </transformations>
  </layer>
```

This needles has its origin in the center of the instrument. If the
needles fulcrum was towards the edge of the instrument, the
transformations to place the pivot point must precede those which
drive the needle,

Interpolation

Non linear transformations are now possible via the use of
interpolation tables.

```
 <transformation>
 ...
 <interpolation>
 <entry>
 <ind>0.0</ind>                <!-- raw value -->
 <dep>0.0</dep>                <!-- displayed value -->
 </entry>
 <entry>
 <ind>10.0</ind>
 <dep>100.0</dep>
 </entry>
 <entry>
 <ind>20.0</ind>
 <dep>-5.0</dep>
 </entry>
```

```
<entry>
<ind>30.0</ind>
<dep>1000.0</dep>
</entry>
</interpolation>
</transformation>
```

Of course, interpolation tables are useful for non-linear stuff, as in
the above example, but I kind-of like the idea of using them for
pretty much everything, including non-trivial linear movement -- many
instrument markings aren't evenly spaced, and the interpolation tables
are much nicer than the older min/max/scale/offset stuff and should
allow for a more realistic panel without adding a full equation parser
to the property manager.

If you want to try this out, look at the airspeed.xml file in the base
package, and uncomment the interpolation table in it for a very funky,
non-linear and totally unreliable airspeed indicator.

Actions:

An action is a hotspot on an instrument where something will happen
when the user clicks the left or center mouse button. Actions are
always tied to properties: they can toggle a boolean property, adjust
the value of a numeric property, or swap the values of two properties.
The x/y placement for actions specifies the origin of the lower left
corner. In the following example the first action sets up a hotspot
32 pixels wide and 16 pixels high. It lower left corner is placed 96
pixels (relative to the defined base size of the instrument) to the
right of the center of the instrument. It is also 32 pixels below the
centerline of the instrument. The actual knob texture over which the
action is superimposed is 32x32. Omitted here is a second action,
bound to the same property, with a positive increment value. This
second action is placed to cover the other half of the knob. The
result is that clicking on the left half of the knob texture decreases
the value and clicking the right half increases the value. Also
omitted here is a second pair of actions with the same coordinates but
a larger increment value. This second pair is bound to a different
mouse button. The net result is that we have both fine and coarse

adjustments in the same hotspot, each bound to a different mouse
button.

These examples come from the radio stack:

```
<actions>                                     <!-- open the actions section -->
  <action>                                    <!- first action -->
   <name>small nav frequency decrease</name>
   <type>adjust</type>
   <button>0</button>                         <!-- bind it to a mouse button -->
   <x>96</x>                                  <!-- placement relative to instrument center -
   <y>-32</y>
   <w>16</w>                                  <!-- size of hotspot -->
   <h>32</h>
   <property>/radios/nav1/frequencies/standby</property>    <!-- bind to a property -
   <increment>-0.05</increment>              <!-- amount of adjustment per mouse click -->
   <min>108.0</min>                          <!-- lower range -->
   <max>117.95</max>                         <!-- upper range -->
   <wrap>1</wrap>                            <!-- value wraps around when it hits bounds --
  </action>
  <action>
   <name>swap nav frequencies</name>
   <type>swap</type>                         <!-- define type of action -->
   <button>0</button>
   <x>48</x>
   <y>-32</y>
   <w>32</w>
   <h>32</h>
   <property1>/radios/nav1/frequencies/selected</property1>   <!-- properties to
   <property2>/radios/nav1/frequencies/standby</property2>       toggle between --
  </action>
  <action>
   <name>ident volume on/off</name>
   <type>adjust</type>
   <button>1</button>
   <x>40</x>
   <y>-24</y>
   <w>16</w>
   <h>16</h>
   <property>/radios/nav1/ident</property>  <!-- Morse code ID of nav beacons -->
   <increment>1.0</increment>               <!-- the increment equals the max value
```

```
                                                    so this toggles on/off -->
    <min>0</min>
    <max>1</max>
    <wrap>1</wrap>                          <!-- a shortcut to avoid having separate
                                                 actions for on/off -->

  </action>
</actions>
```

More About Textures:

As previously stated, the usual size instrument texture files in FGFS
are 256x256 pixels, red/green/blue/alpha format. However the mechanism
for specifying texture cropping coordinates is decimal in nature. When
calling a section of a texture file the 0,0 lower left convention is
used. There is a pair of x/y coordinates defining which section of
the texture to use.

The following table can be used to calculate texture cropping
specifications.

```
# of divisions | width in pixels | decimal specification
per axis
        1   =    256 pixels             1
        2   =    128 pixels,            0.5
        4   =    64 pixels,             0.25
        8   =    32 pixels,             0.125
       16   =    16 pixels,             0.0625
       32   =    8 pixels,              0.03125
       64   =    4 pixels,              0.015625
      128   =    2 pixels,              0.0078125
```

A common procedure for generating gauge faces is to use a vector
graphics package such as xfig, exporting the result as a postscript
file. 3D modeling tools may also be used and I prefer them for pretty
items such as levers, switches, bezels and so forth. Ideally, the
size of the item in the final render should be of proportions that fit
into the recommended pixel widths. The resulting files can be
imported into a graphics manipulation package such as GIMP, et al for
final processing.

How do I get my panels/instruments into the base package?

Cash bribes always help ;) Seriously though, there are two main
considerations. Firstly, original artwork is a major plus since you
as the creator can dictate the terms of distribution. All Artwork must
have a license compatible with the GPL. Artwork of unverifiable
origin is not acceptable. Secondly, texture sizes must meet the
lowest common denominator of 256e2 pixels. Artwork from third parties
may be acceptable if it meets these criteria.

* If there are *any* XML parsing errors, the panel will fail to load,
   so it's worth downloading a parser like Expat (http://www.jclark.com/xml/)
   for checking your XML. FlightGear will print the location of errors, but
   the messages are a little cryptic right now.

** NOTE: There is one built-in layer -- for the mag compass ribbon --
   and all other layers are defined in the XML files. In the future,
   there may also be built-in layers for special things like a
   weather-radar display or a GPS (though the GPS could be handled with
   text properties).

39   Xmlparticles

Document started 27/01/2008 by Tiago Gusmão
Updated 02/02/2008 to reflect syntax changes
Updated 03/02/2008 to add trails (connected particles)

This is a short specification/tutorial to define particle systems in
FlightGear using XML

Meaningless example (what i had accumulated due to tests):

```
  <particlesystem>
    <name>fuel</name>
<!--     <texture>particle.rgb</texture> -->
    <emissive>false</emissive>
    <lighting>true</lighting>
```

```xml
<offsets>
  <x-m>35</x-m>
  <y-m>-0.3</y-m>
  <z-m>0</z-m>
  <!--<pitch-deg>90</pitch-deg>-->
</offsets>

<!--<condition>
  <and>
    <equals>
      <property>engines/engine/smoking</property>
      <value>true</value>
    </equals>
    <less-than>
      <property>position/altitude-agl-ft</property>
      <value>12000</value>
    </less-than>
  </and>
</condition>-->

<attach>world</attach>

<placer>
  <type>point</type>
</placer>

<shooter>
  <theta-min-deg>84</theta-min-deg>
  <theta-max-deg>86</theta-max-deg>
  <phi-min-deg>-1.5</phi-min-deg>
  <phi-max-deg>1.5</phi-max-deg>
  <speed>
    <value>10</value>
    <spread>2.5</spread>
  </speed>
  <rotation-speed>
    <x-min-deg-sec>0</x-min-deg-sec>
    <y-min-deg-sec>0</y-min-deg-sec>
    <z-min-deg-sec>0</z-min-deg-sec>
    <x-max-deg-sec>0</x-max-deg-sec>
```

```
        <y-max-deg-sec>0</y-max-deg-sec>
        <z-max-deg-sec>0</z-max-deg-sec>
    </rotation-speed>
</shooter>

<counter>
  <particles-per-sec>
    <value>1</value>
    <spread>0</spread>
  </particles-per-sec>
</counter>

<align>billboard</align>

<particle>
  <start>
    <color>
      <red>
        <value>0.9</value>
      </red>
      <green>
        <value>0.09</value>
      </green>
      <blue>
        <value>0.09</value>
      </blue>
      <alpha>
        <value>1.0</value>
      </alpha>
    </color>
    <size>
      <value>0.25</value>
    </size>
  </start>

  <end>
    <color>
      <red>
        <value>1</value>
      </red>
```

```xml
          <green>
            <value>0.1</value>
          </green>
          <blue>
            <value>0.1</value>
          </blue>
          <alpha>
            <value>0.0</value>
          </alpha>
        </color>
        <size>
          <value>4</value>
        </size>
      </end>

      <life-sec>
        <value>10</value>
      </life-sec>

      <mass-kg>0.25</mass-kg>
      <radius-m>0.1</radius-m>
    </particle>

    <program>
      <fluid>air</fluid>
      <gravity type="bool">true</gravity>
      <wind type="bool">true</wind>
    </program>
</particlesystem>
```

Stick this inside any model XML like it was an animation and it should
work (notice the condition requires wheel on the ground)

Specification:

Note:
<VALUEORPROP/> means you can either specify a property with factor and
offset (result = (prop*factor)+offset) in the usual way

```
<particlesystem>  = the base tag
  <type>string</type> = can be "normal" or "trail", normal is the usual quad
                        particles, trail is a string of connected line shapes
                        by default.
  <offsets>  = this places the source of the particles (the emitter) in relation
               to the perhaps already offset model (see model-howto.html for details)
    <x-m>float</x-m>
    <y-m>float</y-m>
    <z-m>float</z-m>
    <pitch-deg>float</pitch-deg>
    <roll-deg>float</roll-deg>
    <heading-deg>float</heading-deg>
  </offsets>
  <condition> = a typical condition that if not true stops particles from being
    ....         emitted (PPS=0)
  </condition>
  <name>string</name> = the name of the particle system (so it can be referenced
                        by normal animations)
  <attach>string</attach> = can be "world" or "local". "world means the particles
                            aren't "physically linked" to the model (necessary for
                            use outside moving models), "local" means the opposite
                            (can be used for static objects or inside moving objects)
  <texture>string</texture> = the texture path relative to the XML file location
  <emissive>bool</emissive> = self-explanatory
  <lighting>bool</lighting> = yet to be tested, but seems obvious
  <align>string</align> = can be "billboard" or "fixed"
  <placer> = where particles are born
    <type>string</type> = can be "sector" (inside a circle), "segments"(user-defined
                          segments) and "point" (default)
    *<radius-min-m>float</radius-min-m> = only for sector, inner radius at which
                                          particles appear
    *<radius-max-m>float</radius-max-m> = only for sector, outer radius at which
                                          particles appear
    *<phi-min-deg>float</phi-min-deg> = only for sector, starting angle of the
                                        slide at which particles appear
    *<phi-max-deg>float</phi-max-deg> = only for sector, ending angle of the slide
                                        at which particles appear
    <segments> = only for segments, encloses sequential points that form segments
      <vertex> = specifies one point, put as many as you want
        <x-m>float</x-m>
```

```
        <y-m>float</y-m>
        <z-m>float</z-m>
      </vertex>
       ....
      <vertex>
       ...
      </vertex>
    </segments>
</placer>
<shooter> = the shooter defines the initial velocity vector for your particles
  *<theta-min-deg>float</theta-min-deg> = horizontal angle limits of the particle c
  *<theta-max-deg>float</theta-max-deg>
  *<phi-min-deg>float</phi-min-deg> = vertical angle limits of the particle cone
  *<phi-max-deg>float</phi-max-deg>    for an illustration of theta/phi see
  http://www.cs.clemson.edu/~malloy/courses/3dgames-2007/tutor/web/particles/partic
  <speed-mps> = the scalar velocity (meter per second)
   <VALUEORPROP/> = see note
   *<spread> = the "tolerance" in each direction so values are in the range
                 [value-spread, value+spread]
  </speed-mps>
  <rotation-speed> = the range of initial rotational speed of the particles
    *<x-min-deg-sec>float</x-min-deg-sec>
    *<y-min-deg-sec>float</y-min-deg-sec>
    *<z-min-deg-sec>float</z-min-deg-sec>
    *<x-max-deg-sec>float</x-max-deg-sec>
    *<y-max-deg-sec>float</y-max-deg-sec>
    *<z-max-deg-sec>float</z-max-deg-sec>
  </rotation-speed>
</shooter>
<counter>
  <particles-per-sec>
    <VALUEORPROP/> = see note
    *<spread> = the "tolerance" in each direction so values are in the range
                 [value-spread, value+spread]
  </particles-per-sec>
</counter>
<particle> = defines the particle properties
  <start>
    <color> = initial color (at time of emission)
      <red><VALUEORPROP/></red> = color component in normalized value [0,1]
```

```
      <green><VALUEORPROP/></green>
      <blue><VALUEORPROP/></blue>
      <alpha><VALUEORPROP/></alpha>
    </color>
    <size> = as above, but for size
      <VALUEORPROP/>
    </size>
  </start>
  <end>
    <color> = final color (at the end of the particle life)
      <red><VALUEORPROP/></red>
      <green><VALUEORPROP/></green>
      <blue><VALUEORPROP/></blue>
      <alpha><VALUEORPROP/></alpha>
    </color>
    <size>
      <VALUEORPROP/>
    </size>
  </end>
  *<life-sec> = the time the particles will be alive, in seconds
    <VALUEORPROP/>
  *</life-sec>
  *<radius-m>float</radius-m> = each particles is physically treated as a sphere
                                with this radius
  *<mass-kg>float</mass-kg> = mass in KG
</particle>
<program> = defines external forces acting upon a particle
  <fluid>string<fluid> = can be "air" or "water"
  <gravity>bool</gravity> = can be "true" or "false". uses standard gravity
  <wind>bool</wind> = can be "true" or "false". uses user position wind (not the
                      model position, but shouldn't be noticeable, you want to
                      disabled it when using local attach)
</program>
</particles>

Remarks:
* Don't forget you can use existing animations with particles, so if you want to
  direct or translate the emitter, just use translate, rotate, spin and so on
  (other animations might have interesting effects too I guess)
```

* Particle XML should be compatible with plib, as the tags will be ignored (you might get some warning if you attach them to animations though)

* Try not to use a lot of particles in a way that fills the screen, that will demand lots of fill rate and hurt FPS

* If you don't use any properties nor conditions, your particle system doesn't need to use a callback a so it's slightly better on the CPU (mostly useful for static mo

* If your particle lifetime is too big you might run out of particles temporarily (still being investigated)

* Use mass and size(radius) to adjust the reaction to gravity and wind (mass/size = density)

* Although at the moment severe graphical bugs can be seen in the trails, they are usable.

* Consider your options correctly! You should consider giving them no initial velocity and most important, no spread, otherwise particles will race and the trail will fold. Start simple (no velocities and forces) and work your way up.

40   Xmlsound

Users Guide to FlightGear sound configuration
Version 0.9.8, October 30, 2005
Author: Erik Hofman <erik at ehofman dot com>

This document is an attempt to describe the configuration of
FlightGear flight simulator's aircraft sound in XML.

Sound Architecture:

All of the sound configuration files are XML-encoded* property lists.
The root element of each file is always named <PropertyList>. Tags are
almost always found in pairs, with the closing tag having a slash
prefixing the tag name, i.e </PropertyList>. The exception is the tag
representing an aliased property. In this case a slash is prepended to
the closing angle bracket. (see section Aliasing)

The top level sound configuration file is composed of a <fx>, a
<name>, a <path> sound file and zero or more <volume> and/or <pitch>
definitions.

[Paths are relative to $FG_ROOT (the root of the installed base package .)]
[Absolute paths may be used. Comments are bracketed with <!-- -->.   ]

A limited sound configuration file would look something like this:

```
<PropertyList>
 <fx>
  <engine>
   <name>engine</name>
   <path>Sounds/wasp.wav</path>
   <mode>looped</mode>
   <condition>
    <property>/engines/engine/running</property>
   </condition>
   <volume>
    <property>/engines/engine/mp-osi</property>
    <factor>0.005</factor>
    <min>0.15</min>
    <max>0.5</max>
    <offset>0.15</offset>
   </volume>
   <pitch>
    <property>/engines/engine/rpm</property>
    <factor>0.0012</factor>
    <min>0.3</min>
    <max>5.0</max>
    <offset>0.3</offset>
   </pitch>
  </engine>
 </fx>
</PropertyList>
```

This would define an engine sound event handler for a piston engine driven
aeroplane. The sound representing the engine is located in $FG_ROOT/Sounds
and is named wasp.wav. The event is started when the property

/engines/engine/running becomes non zero.

When that happens, the sound will be played looped (see <mode>) until the property returns zero again. As you can see the volume is mp-osi dependent, and the pitch of the sound depends on the engine rpm.

Configuration description:

<fx>
    Named FX subtree living under /sim/sound

 < ... >
    This is the event separator. The text inside the brackets
    can be anything. Bit it is advised to give it a meaningful name
    like: crank, engine, rumble, gear, squeal, flap, wind or stall

    The value can be defined multiple times, thus anything which is
    related may have the same name (grouping them together).

  <name>
    This defines the name of the event. This name is used internally
    and, although it can me defined multiple times in the same file,
    should normally have an unique value.

    Multiple definitions of the same name will allow multiple sections
    to interfere in the starting and stopping of the sample.

    This method can't be used to control the pitch or volume of the
    sample, but instead multiple volume or pitch section should be
    included inside the same event.

    The types "raise" and "fall" will stop the playback of the sample
    regardless of any other event. This means that when the type "raise"
    is supplied, sample playback will stop when the event turns false.
    Using the type "fall" will stop playback when the event turns true.

    IMPORTANT:
    If the trigger is used for anything else but stopping the sound
     at a certain event, all sections with the same name *should* have

exactly the same sections for everything but property and type.

In the case of just stopping the sample at a certain event, the
sections for path, volume and pitch may be omitted.

<path>
This defined th path to the sound file. The path is relative to the
FlightGear root directory but could be specified absolute.

<condition>
Define a condition that triggers the event.
For a complete description of the FlightGear conditions,
please read docs-mini/README.conditions

An event should define either a condition or a property.

<property>
Define which property triggers the event, and refers to a node
in the FlightGear property tree. Action is taken when the property
is non zero.

A more sophisticated mechanism to trigger the event is described
in <condition>

<mode>
This defines how the sample should be played:

once: the sample is played once.
                this is the default.

looped: the sample plays continuously,
                until the event turns false.

in-transit: the sample plays continuously,
                while the property is changing its value.

<type>
This defines the type os this sample:

fx: this is the default type and doesn't need to be defined.

```
    avionics:        sounds set to this time don't have a position and
                     orientation but are treated as if it's mounted to
                     the aircraft panel. it's up to the user to define
                     if it can always be heard or only when in cockpit
                     view.
```

```
<volume> / <pitch>
    Volume or Pitch definition. Currently there may be up to 5
    volume and up to 5 pitch definitions defined within one sound
    event. Normally all offset values are added together and the
    results after property calculations will be multiplied.
    A special condition occurs when the value of factor is negative,
    in which case the offset doesn't get added to the other offset values
    but instead will be used in the multiplication section.
```

```
  <property>
    Defines which property supplies the value for the calculation.
    Either a <property> or an <internal> should be defined.
    The value is treated as a floating point number.
```

```
  <internal>
    Defines which internal variable should be used for the calculation.
    The value is treated as a floating point number.
    The following internals are available at this time:
```

```
    dt_play:         the number of seconds since the sound started playing.
```

```
    dt_stop:         the number of seconds after the sound has stopped.
```

```
  <delay-sec>
    Delay after which the sound starts playing. This is useful to let
    a property start two sounds at the same time, where the second is
    delayed until the first stopped playing.
```

```
  <type>
    Defines the function that should be used upon the property
    before it is used for calculating the net result:
```

```
    lin:             linear handling of the property value.
```

this is the default.

    ln: convert the property value to a natural logarithmic
                        value before scaling it. Anything below 1 will return
                        zero.

    log: convert the property value to a true logarithmic
                        value before scaling it. Anything below 1 will return
                        zero.

    inv: inverse linear handling (1/x).

    abs: absolute handling of the value (always positive).

    sqrt: calculate the square root of the absolute value
                        before scaling it.

<factor>
    Defines the multiplication factor for the property value.
    A special condition is when scale is defined as a negative
    value. In this case the result of |<scale>| * <property) will be
    subtracted from <default>

<offset>
    The initial value for this sound. This value is also used as an
    offset value for calculating the end result.

<min>
    Minimum allowed value.
    This is useful if sounds start to sound funny. Anything lower
    will be truncated to this value.

<max>
    Maximum allowed value.
    This is useful if sounds gets to loud. Anything higher will be
    truncated to this value.

<position>
    Specify the position of the sounds source relative to the
    aircraft center. The coordinate system used is a left hand

```
    coordinate system where +Y = left, -Y = right, -Z = down, +Z =
    up, -X = forward, +X = aft.  Distances are in meters.
    The volume calculation due to distance and orientation of the
    sounds source ONLY work on mono samples!

  <x>
    X dimension offset

  <y>
    Y dimension offset

  <z>
    Z dimension offset


<orientation>
    Specify the orientation of the sounds source.

    The zero vector is default, indicating that a Source is not directional.
    Specifying a non-zero vector will make the Source directional in
    the X,Y,Z direction

  <x>
    X dimension

  <y>
    Y dimension

  <z>
    Z dimension

  <inner-angle>
    The inner edge of the audio cone in degrees (0.0 - 180.0).
    Any sound withing that angle will be played at the current gain.

  <outer-angle>
    The outer edge of the audio cone in degrees (0.0 - 180.0).
    Any sound beyond the outer cone will be played at "outer-gain" volume.

  <outer-gain>
```

The gain at the outer edge of the cone.

    <reference-dist>
        Set a reference distance of sound in meters. This is the
        distance where the volume is at its maximum.
        Volume is clamped to this maximum for any distance below.
        Volume is attenuated for any distance above.
        Attenuation depends on reference and maximum distance. See
        OpenAL specification on "AL_INVERSE_DISTANCE_CLAMPED" mode
        for details on exact computation.

    <max-dist>
        Set the maximum audible distance for the sound in meters.
        Sound is cut-off above this distance.

Creating a configuration file:

To make things easy, there is a default value for most entries to allow a
sane configuration when a certain entry is omitted.

Default values are:

type: lin
factor: 1.0
offset: 0.0 for volume, 1.0 for pitch
min: 0.0
max: 0.0 (don't check)

Calculations are made the following way (for both pitch and volume):

```
    value = 0;
    offs = 0;

    for (n = 0;  n < max; n++) {
        if (factor < 0)
```

```
        {
            value += offset[n] - abs(factor[n]) * function(property[n]);
        }
        else
        {
            value += factor[n] * function(property[n]);
            offs += offset[n];
        }
    }

    volume = offs + value;
```

where function can be one of: lin, ln, log, inv, abs or sqrt

41   Xmlsyntax

XML IN FIFTEEN MINUTES OR LESS

Written by David Megginson, david@megginson.com
Last modified: $Date$

This document is in the Public Domain and comes with NO WARRANTY!

1. Introduction

FlightGear uses XML for much of its configuration. This document
provides a minimal introduction to XML syntax, concentrating only on
the parts necessary for writing and understanding FlightGear
configuration files. For a full description, read the XML
Recommendation at

  http://www.w3.org/TR/

This document describes general XML syntax. Most of the XML
configuration files in FlightGear use a special format called
"Property Lists" -- a separate document will describe the specific
features of the property-list format.

2. Elements and Attributes

An XML document is a tree structure with a single root, much like a
file system or a recursive, nested list structure (for LISP fans).
Every node in the tree is called an _element_: the start and end of
every element is marked by a _tag_: the _start tag_ appears at the
beginning of the element, and the _end tag_ appears at the end.

Here is an example of a start tag:

  <foo>

Here is an example of an end tag:

  </foo>

Here is an example of an element:

  <foo>Hello, world!</foo>

The element in this example contains only data element, so it is a
leaf node in the tree. Elements may also contain other elements, as
in this example:

  <bar>
   <foo>Hello, world!</foo>
   <foo>Goodbye, world!</foo>
  </bar>

This time, the 'bar' element is a branch that contains other, nested
elements, while the 'foo' elements are leaf elements that contain only
data. Here's the tree in ASCII art (make sure you're not using a
proportional font):

  bar +-- foo -- "Hello, world!"
      |
      +-- foo -- "Goodbye, world!"

257

There is always one single element at the top level: it is called the
root element. Elements may never overlap, so something like this is
always wrong (try to draw it as a tree diagram, and you'll understand
why):

```
<a><b></a></b>
```

Every element may have variables, called _attributes_, attached to
it. The attribute consists of a simple name=value pair in the start
tag:

```
<foo type="greeting">Hello, world!</foo>
```

Attribute values must be quoted with '"' or "'" (unlike in HTML), and
no two attributes may have the same name.

There are rules governing what can be used as an element or attribute
name. The first character of a name must be an alphabetic character
or '_'; subsequent characters may be '_', '-', '.', an alphabetic
character, or a numeric character. Note especially that names may not
begin with a number.

3. Data

Some characters in XML documents have special meanings, and must
always be escaped when used literally:

```
<  &lt;
&  &amp;
```

Other characters have special meanings only in certain contexts, but
it still doesn't hurt to escape them:

```
>  &gt;
'  &apos;
"  &quot;
```

Here is how you would escape "x < 3 && y > 6" in XML data:

    x < 3 && y > 6

Most control characters are forbidden in XML documents: only tab, newline, and carriage return are allowed (that means no ^L, for example). Any other character can be included in an XML document as a character reference, by using its Unicode value; for example, the following represents the French word "cafe" with an accent on the final 'e':

    café

By default, 8-bit XML documents use UTF-8, **NOT** ISO 8859-1 (Latin 1), so it's safest always to use character references for characters above position 127 (i.e. for non-ASCII).

Whitespace always counts in XML documents, though some specific applications (like property lists) have rules for ignoring it in some contexts.

4. Comments

You can add a comment anywhere in an XML document except inside a tag or declaration using the following syntax:

    <!-- comment -->

The comment text must not contain "--", so be careful about using dashes.

5. XML Declaration

Every XML document may begin with an XML declaration, starting with "<?xml" and ending with "?>". Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

The XML declaration must always give the XML version, and it may also
specify the encoding (and other information, not discussed here).
UTF-8 is the default encoding for 8-bit documents; you could also try

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

to get ISO Latin 1, but some XML parsers might not support that
(FlightGear's does, for what it's worth).

6. Other Stuff

There are other kinds of things allowed in XML documents. You don't
need to use them for FlightGear, but in case anyone leaves one lying
around, it would be useful to be able to recognize it.

XML documents may contain different kinds of declarations starting
with "<!" and ending with ">":

```
<!DOCTYPE html SYSTEM "html.dtd">

<!ELEMENT foo (#PCDATA)>

<!ENTITY myname "John Smith">
```

and so on. They may also contain processing instructions, which look
a bit like the XML declaration:

```
<?foo processing instruction?>
```

Finally, they may contain references to _entities_, like the ones used
for escaping special characters, but with different names (we're
trying to avoid these in FlightGear):

```
&chapter1;

&myname;
```

Enjoy.

42   Yasim

Coordinate system notes: All positions specified are in meters (which
is weird, since all other units in the file are English). The X axis
points forward, Y is left, and Z is up. Take your right hand, and
hold it like a gun. Your first and second fingers are the X and Y
axes, and your upwards-pointing thumb is the Z. This is slightly
different from the coordinate system used by JSBSim. Sorry. The
origin can be placed anywhere, so long as you are consistent. I use
the nose of the aircraft.

```
XML Elements
------------


airplane: The top-level element for the file.  It contains only one
          attribute:
          mass: The empty (no fuel) weight, in pounds.

approach: The approach parameters for the aircraft.  The solver will
          generate an aircraft that matches these settings.  The element
          can (and should) contain <control> elements indicating pilot
          input settings, such as flaps and throttle, for the
          approach.
          speed: The approach airspeed, in knots TAS.
          aoa:   The approach angle of attack, in degrees
          fuel:  Fraction (0-1) of fuel in the tanks.  Default is 0.2.

cruise:   The cruise speed and altitude for the solver to match.  As
          above, this should contain <control> elements indicating
          aircraft configuration.  Especially, make sure the engines
          are generating enough thrust at cruise!
          speed: The cruise speed, in knots TAS.
          alt:   The cruise altitude, in feet MSL.
          fuel:  Fraction (0-1) of fuel in the tanks.  Default is 0.2.
```

```
cockpit:   The location of the cockpit (pilot eyepoint).
           x,y,z: eyepoint location (see coordinates note)

fuselage:  This defines a tubelike structure.  It will be given an even
           mass and aerodynamic force distribution by the solver.  You
           can have as many as you like, in any orientation you please.
           ax,ay,az: One end of the tube (typically the front)
           bx,by,bz: The other ("back") end.
           width:    The width of the tube, in meters.
           taper:    The approximate radius at the "tips" of the fuselage
                     expressed as a fraction (0-1) of the width value.
           midpoint: The location of the widest part of the fuselage,
                     expressed as a fraction of the distance between A and B.
           idrag:    Multiplier for the "induced drag" generated by this
                     object. Default is one. With idrag=0 the fuselage
                     generates only drag.
           cx,cy,cz: Factors for the generated drag in the fuselages "local
                     coordinate system" with x pointing from end to front,
                     z perpendicular to x with y=0 in the aircraft coordinate
                     system. E.g. for a fuselage of a height of 2 times the
                     width you can define cy=2 and (due to the doubled front
                     surface) cx=2.

wing:      This defines the main wing of the aircraft.  You can have
           only one (but see below about using vstab objects for extra
           lifting surfaces).  The wing should have a <stall> subelement to
           indicate stall behavior, control surface subelements (flap0,
           flap1, spoiler, slat) to indicate what and where the control
           surfaces are, and <control> subelements to map user input
           properties to the control surfaces.
           x,y,z:    The "base" of the wing, specified as the location of
                     the mid-chord (not leading edge, trailing edge, or
                     aerodynamic center) point at the root of the LEFT
                     (!)  wing.
           length:   The length from the base of the wing to the midchord
                     point at the tip.  Note that this is not the same
                     thing as span.
           chord:    The chord of the wing at its base, along the X axis
                     (not normal to the leading edge, as it is
                     sometimes defined).
```

incidence: The incidence angle at the wing root, in degrees. Zero is level with the fuselage (as in an aerobatic plane), positive means that the leading edge is higher than the trailing edge (as in a trainer).

twist: The difference between the incidence angle at the wing root and the incidence angle at the wing tip. Typically, this is a negative number so that the wing tips have a lower angle of attack and stall after the wing root (washout).

taper: The taper fraction, expressed as the tip chord divided by the root chord. A taper of one is a hershey bar wing, and zero would be a wing ending at a point. Defaults to one.

sweep: The sweep angle of the wing, in degrees. Zero is no sweep, positive angles are swept back. Defaults to zero.

dihedral: The dihedral angle of the wing. Positive angles are upward dihedral. Defaults to zero.

idrag: Multiplier for the "induced drag" generated by this surface. In general, low aspect wings will generate less induced drag per-AoA than high aspect (glider) wings. This value isn't constrained well by the solution process, and may require tuning to get throttle settings correct in high AoA (approach) situations.

camber: The lift produced by the wing at zero angle of attack, expressed as a fraction of the maximum lift produced at the stall AoA.

hstab: These defines the horizontal stabilizer of the aircraft. Internally, it is just a wing object and therefore works the same in XML. You are allowed only one hstab object; the solver needs to know which wing's incidence to play with to get the aircraft trimmed correctly.

vstab: A "vertical" stabilizer. Like hstab, this is just another wing, with a few special properties. The surface is not "mirrored" as are wing and hstab objects. If you define a left wing only, you'll only get a left wing. The default

dihedral, if unspecified, is 90 degrees instead of zero. But all parameters are equally settable, so there's no requirement that this object be "vertical" at all. You can use it for anything you like, such as extra wings for biplanes. Most importantly, these surfaces are not involved with the solver computation, so you can have none, or as many as you like.

mstab: A mirrored horizontal stabilizer. Exactly the same as wing, but not involved with the solver computation, so you can have none, or as many as you like.

stall: A subelement of a wing (or hstab/vstab/mstab) that specifies the stall behavior.
          aoa: The stall angle (maximum lift) in degrees. Note that this is relative to the wing, not the fuselage (since the wing may have a non-zero incidence angle).
          width: The "width" of the stall, in degrees. A high value indicates a gentle stall. Low values are viscious for a non-twisted wing, but are acceptable for a twisted one (since the whole wing will not stall at the same time).
          peak: The height of the lift peak, relative to the post-stall secondary lift peak at 45 degrees. Defaults to 1.5. This one is deep voodoo, and probably doesn't need to change much. Bug me for an explanation if you're curious.

flap0, flap1, slat, spoiler:
          These are subelements of wing/hstab/vstab objects, and specify the location and effectiveness of the control surfaces.
          start: The position along the wing where the control surface begins. Zero is the root, one is the tip.
          end: The position where the surface ends, as above.
          lift: The lift multiplier for a flap or slat at full extension. One is a no-op, a typical aileron might be 1.2 or so, a giant jetliner flap 2.0, and a spoiler 0.0. For spoilers, the interpretation is a little different -- they spoil only "prestall" lift. Lift due purely to "flat plate" effects isn't

affected. For typical wings that stall at low AoA's
essentially all lift is pre-stall and you don't have
to care. Jet fighters tend not to have wing
spoilers, for exactly this reason. This value is
not applicable to slats, which affect stall AoA
only.

drag: The drag multiplier, as above. Typically should be
higher than the lift multiplier for flaps.

aoa: Applicable only to slats. This indicates the
angle by which the stall AoA is translated by the
slat extension.

thruster: A very simple "thrust only" engine object. Useful for
things like thrust vectoring nozzles. All it does is map
its THROTTLE input axis to its output thrust rating. Does
not consume fuel, etc...

thrust: Maximum thrust in pounds

x,y,z: The point on the airframe where thrust will be
applied.

vx,vy,vy: The direction of the thrust in airframe
coordinates. The vector will be normalized
automatically, so any non-zero vector will work
fine.

jet: A turbojet/fan engine. It accepts a <control> subelement to map a
property to its throttle setting, and an <actionpt> subelement
to place the action point of the thrust at a different
position than the mass of the engine.

x,y,z: The location of the engine, as a point mass.
If no actionpt is specified, this will also
be the point of application of thrust.

mass: The mass of the engine, in pounds.

thrust: The maximum sea-level thrust, in pounds.

afterburner: Maximum total thrust with afterburner/reheat,
in pounds [defaults to "no additional
thrust"].

rotate: Vector angle of the thrust in degrees about the
Y axis [0].

n1-idle: Idling rotor speed [55].

n1-max: Maximum rotor speed [102].

265

```
            n2-idle:         Idling compressor speed [73].
            n2-max:          Maximum compressor speed [103].
            tsfc:            Thrust-specific fuel consumption [0.8].
                             This should be considerably lower for modern
                             turbofans.
            egt:             Exhaust gas temperature at takeoff [1050].
            epr:             Engine pressure ratio at takeoff [3.0].
            exhaust-speed:   The maximum exhaust speed in knots [~1555].
            spool-time:      Time, in seconds, for the engine to respond to
                             90% of a commanded power setting.

propeller:  A propeller.  This element requires an engine subtag.
            Currently <piston-engine> and <turbine-engine> are
            supported.
            x,y,z:           The position of the mass (!) of the
                             engine/propeller combination.  If the point
                             of force application is different (and it
                             will be) it should be set with an <actionpt>
                             subelement.
            mass:            The mass of the engine/propeller, in pounds.
            moment:          The moment, in kg-meters^2.  This has to be
                             hand calculated and guessed at for now.  A
                             more automated system will be forthcoming.
                             Use a negative moment value for
                             counter-rotating ("European" -- CCW as seen
                             from behind the prop) propellers.
                             A good guess for this value is the radius of
                             the prop (in meters) squared times the mass
                             (kg) divided by three; that is the moment of
                             a plain "stick" bolted to the prop shaft.
            radius:          The radius, in meters, or the propeller.
            cruise-speed:    The max efficiency cruise speed of the
                             propeller.  Generally not the same as the
                             aircraft's cruise speed.
            cruise-rpm:      The RPM of the propeller at max-eff. cruise.
            cruise-power:    The power sunk by the prop at cruise, in horsepower.
            cruise-alt:      The reference cruise altitude in feet.
            takeoff-power:   The takeoff power required by the propeller...
            takeoff-rpm:     ...at the given takeoff RPM.
            min-rpm:         The minimum operational RPM for a constant speed
```

```
                      propeller.  This is the speed to which the
                      prop governor will seek when the blue lever
                      is at minimum.  The coarse-stop attribute
                      limits how far the governor can go into trying
                      to reach this RPM.
        max-rpm:      The maximum operational RPM for a constant speed
                      propeller.  See above.  The fine-stop attribute
                      limits how far the governor can go in trying
                      to reach this RPM.
        fine-stop:    The minimum pitch of the propeller (high RPM) as a
                      ratio of ideal cruise pitch.  This is set to 0.25
                      by default -- a higher value will result in a
                      lower RPM at low power settings (e.g. idle, taxi,
                      and approach).
        coarse-stop:  The maximum pitch of the propeller (low RPM) as
                      a ratio of ideal cruise pitch.  This is set to
                      4.0 by default -- a lower value may result in a
                      higher RPM at high power settings.
        gear-ratio:   The factor by which the engine RPM is multiplied
                      to produce the propeller RPM.  Optional (defaults
                      to 1.0).
        contra:       When set (contra="1"), this indicates that the
                      propeller is a contra-rotating pair.  It
                      will not contribute to the aircraft's net
                      gyroscopic moment, nor will it produce
                      asymmetric torque on the aircraft body.
                      Asymmetric slipstream effects, when
                      implemented, will also be zero when this is
                      set.

piston-engine: A piston engine definition.  This must be a subelement
               of an enclosing <propeller> tag.
               eng-power:    Maximum BHP of the engine at sea level.
               eng-rpm:      The engine RPM at which eng-power is developed
               displacement: The engine displacement in cubic inches.
               compression:  The engine compression ratio.
               turbo-mul:    The turbo/super-charger pressure multiplier.
                             Static pressure will be multiplied by this
                             value to get the manifold pressure.
               wastegate-mp: The maximum manifold pressure.  Beyond
```

267

this, the gate will release to keep the
                          MP below this number. (inHG). This value
                          can be changed at runtime using the
                          WASTEGATE control axis, which is a
                          multiplier in the range [0:1].
              turbo-lag: Time lag, in seconds, for 90% of a power change
                          to be reflected in the turbocharger boost
                          pressure.

turbine-engine: A turbine engine definition. This must be a subelement
                of an enclosing <propeller> tag.
                eng-power: Maximum BHP of the engine at a suitable
                            cruise altitude.
                eng-rpm: The engine RPM at which eng-power is
                            developed. Note that this is "shaft" RPM
                            as seen by the propeller. Don't use a
                            gear-ratio on the enclosing propeller, or
                            else you'll get confused. :)
                alt: The altitude at which eng-power is developed.
                            This should be high enough to be lower (!)
                            than the flat-rating power.
                flat-rating:The maximum allowed power developed by
                            the engine. Most turboprops are flat
                            rated below a certain altitude and
                            temperature range to prevent engine
                            damage.
                min-n2:   N2 (percent) turbine speed at zero throttle.
                max-n2:   N2 (percent) turbine speed at max throttle.
                bsfc: Specific fuel consumption, in lbs/hr per
                            horsepower.

actionpt: Defines an "action point" for an enclosing jet or propeller
          element. This is the location where the force from the thruster
          will be applied.
          x,y,z: The location of force application.

gear: Defines a landing gear. Accepts <control> subelements to map
          properties to steering and braking. Can also be used to simulate
          floats. Although the coefficients are still called ..fric, it

```
is calculated in fluids as a drag (proportional to the square
of the speed). In fluids gears are not considered to detect
crashes (as on ground).
x,y,z:  The location of the fully-extended gear tip.
compression:  The distance in meters along the "up" axis that
              the gear will compress.
initial-load: The initial load of the spring in multiples of
              compression. Defaults to 0. (With this parameter
              a lower spring-constants will be used for the
              gear-> can reduce numerical problems (jitter))
              Note: the spring-constant is varied from 0%
              compression to 20% compression to get continuous
              behavior around 0 compression. (could be physically
              explained by wheel deformation)
upx/upy/upz:  The direction of compression, defaults to
              vertical (0,0,1) if unspecified.  These are
              used only for a direction -- the vector need
              not be normalized, as the length is specified
              by "compression".
sfric:        Static (non-skidding) coefficient of
              friction.  Defaults to 0.8.
dfric:        Dynamic friction.  Defaults to 0.7.
spring:       A dimensionless multiplier for the automatically
              generated spring constant.  Increase to make
              the gear stiffer, decrease to make it
              squishier.
damp:         A dimensionless multiplier for the automatically
              generated damping coefficient.  Decrease to
              make the gear "bouncier", increase to make it
              "slower".  Beware of increasing this too far:
              very high damping forces can make the numerics
              unstable.  If you can't make the gear stop
              bouncing with this number, try increasing the
              compression length instead.
on-water:     if this is set to "0" the gear will be ignored if
              on water. Defaults to "0"
on-solid:     if this set to "0" the gear will be ignored if
              not on water. Defaults to "1"
speed-planing:
spring-factor-not-planing:
```

At zero speed the spring factor is multiplied by
spring-factor-not-planing. Above speed-planing this
factor is equal to 1. The idea is, to use this for
floats simulating the transition from swimming to
planing. speed-planing defaults to 0,
spring-factor-not-planing defaults to 1.

reduce-friction-by-extension: at full extension the friction is
reduced by this relative value. 0.7 means 30% friction
at full extension. If you specify a value greater
than one, the friction will be zero before reaching
full extension. Defaults to "0"

ignored-by-solver: with the on-water/on-solid tags you can have more
than one set of gears in one aircraft, If the solver
(who automatically generates the spring constants)
would take all gears into account, the result would be
wrong. E. G. set this tag to "1" for all gears, which
are not active on runways. Defaults to "0". You can
not exclude all gears in the solving process.

launchbar: Defines a catapult launchbar or strop. The default acceleration
provided by the catapult is 25m/s^2. This can be
modified by the use of the control axis LACCEL.

    x,y,z: The location of the mount point of the launch bar or
strop on the aircraft.

    length: The length of the launch bar from mount point to tip

    down-angle: The max angle below the horizontal the
launchbar can achieve.

    up-angle: The max angle above the horizontal the launchbar
can achieve.

    holdback-{x,y,z}: The location of the holdback mount point
on the aircraft.

    holdback-length: The length of the holdback from mount
point to tip. Note: holdback up-angle and
down-angle are the same as those defined
for the launchbar and are not specified in
the configuration.

tank: A fuel tank. Tanks in the aircraft are identified
numerically (starting from zero), in the order they are
defined in the file. If the left tank is first, "tank[0]"

```
          will be the left tank.
          x,y,z:   The location of the tank.
          capacity: The maximum contents of the tank, in pounds.  Not
                    gallons -- YASim supports fuels of varying
                    densities.
          jet:     A boolean.  If present, this causes the fuel
                    density to be treated as Jet-A.  Otherwise,
                    gasoline density is used.  A more elaborate
                    density setting (in pounds per gallon, for
                    example) would be easy to implement.  Bug me.

ballast:  This is a mechanism for modifying the mass distribution of
          the aircraft.  A ballast setting specifies that a particular
          amount of the empty weight of the aircraft must be placed at
          a given location.  The remaining non-ballast weight will be
          distributed "intelligently" across the fuselage and wing
          objects.  Note again: this does NOT change the empty weight
          of the aircraft.
          x,y,z: The location of the ballast.
          mass:  How much mass, in pounds, to put there.  Note that
                 this value can be negative.  I find that I often need
                 to "lighten" the tail of the aircraft.

weight:   This is an added weight, something not part of the empty
          weight of the aircraft, like passengers, cargo, or external
          stores.  The actual value of the mass is not specified here,
          instead, a mapping to a property is used.  This allows
          external code, such as the panel, to control the weight
          (loading a given cargo configuration from preference files,
          dropping bombs at runtime, etc...)
          x,y,z:     The location of the weight.
          mass-prop: The name of the fgfs property containing the
                     mass, in pounds, of this weight.
          size:      The aerodynamic "size", in meters, of the
                     object.  This is important for external stores,
                     which will cause drag.  For reasonably
                     aerodynamic stuff like bombs, the size should be
                     roughly the width of the object.  For other
                     stuff, you're on your own.  The default is zero,
                     which results in no aerodynamic force (internal
```

cargo).

    solve-weight:
            Subtag of approach and cruise parameters. Used to specify a
            non-zero setting for a <weight> tag during solution. The
            default is to assume all weights are zero at the given
            performance numbers.
            idx: Index of the weight in the file (starting with zero).
            weight: Weight setting in pounds.

    control-input:
            This element manages a mapping from fgfs properties (user
            input) to settable values on the aircraft's objects. Note
            that the value to be set MUST (!) be valid on the given
            object type. This is not checked for by the parser, and
            will cause a runtime crash if you try it. Wing's don't have
            throttle controls, etc... Note that multiple axes may be
            set on the same value. They are summed before setting.

            axis: The name of the double-valued fgfs property "axis" to
                    use as input, such as "/controls/flight/aileron".
            control: Which control axis to set on the objects. It can have
                     the following values:
                THROTTLE - The throttle on a jet or propeller.
                MIXTURE - The mixture on a propeller.
                REHEAT - The afterburner on a jet
                PROP - The propeller advance
                BRAKE - The brake on a gear.
                STEER - The steering angle on a gear.
                INCIDENCE - The incidence angle of a wing.
                FLAP0 - The flap0 deflection of a wing.
                FLAP1 - The flap1 deflection of a wing.
                FLAP[0/1]EFFECTIVENESS - a multiplier for flap lift, but not drag
                                        (useful for blown flaps)
                SLAT - The slat extension of a wing.
                SPOILER - The spoiler extension for a wing.
                CYCLICAIL - The "aileron" cyclic input of a rotor
                CYCLICELE - The "elevator" cyclic input of a rotor
                COLLECTIVE - The collective input of a rotor

                              272

```
                   ROTORENGINEON - If not equal zero the rotor is rotating
                   WINCHRELSPEED - The relative winch speed
           LACCEL - The acceleration provided by the catapult.
                   {... and many more, see FGFDM.cpp ...}
           invert: Negate the value of the property before setting on
                   the object.
           split:  Applicable to wing control surfaces.  Sets the
                   normal value on the left wing, and a negated value
                   on the right wing.
           square: Squares the value before setting.  Useful for
                   controls like steering that need a wide range, yet
                   lots of sensitivity in the center.  Obviously only
                   applicable to values that have a range of [-1:1] or
                   [0:1].
           src0/src1/dst0/dst1:
                   If present, these defined a linear mapping from the
                   source to the output value.  Input values in the
                   range src0-src1 are mapped linearly to dst0-dst1,
                   with clamping for input values that lie outside the
                   range.

control-output:
           This can be used to pass the value of a YASim control axis
           (after all mapping and summing is applied) back to the
           property tree.

           control: Name of the control axis.  See above.
           prop:    Property node to receive the value.
           side:    Optional, for split controls.  Either "right" or "left"
           min/max: Clamping applied to output value.

control-speed:
           Some controls (most notably flaps and hydraulics) have
           maximum slew rates and cannot respond instantly to pilot
           input.  This can be implemented with a control-speed tag,
           which defines a "transition time" required to slew through
           the full input range.  Note that this tag is
           semi-deprecated, complicated control input filtering can be
           done much more robustly from a Nasal script.
```

```
         control: Name of the control axis. See above.
         transition-time: Time in seconds to slew through input range.

control-setting:
         This tag is used to define a particular setting for a
         control axis inside the <cruise> or <approach> tags, where
         obviously property input is not available.  It can be used,
         for example, to inform the solver that the approach
         performance values assume full flaps, etc...

         axis:  Name of the control input (i.e. a property name)
         value: Value of the control axis.

hitch:   A hitch, can be used for winch-start (in gliders) or aerotow (in
         gliders and motor aircrafts) or for external cargo with helicopter.
         You can do aerotow over the net via multiplayer (see j3 and bocian
         as an example).

         name:  the name of the hitch. must be aerotow if you want to do
                aerotow via multiplayer. You will find many properties
                at /sim/hitches/name. Most of them are directly tied to
                the internal variables, you can modify them as you like.
                You can add a listener to the property "broken", e. g. for
                playing a sound.
         x,y,z: The position of the hitch
         force-is-calculated-by-other: if you want to simulate aerotowing
                over the internet, set this value to "1" in the motor
                aircraft. Don't specify or set this to zero in gliders.
                In a LAN the time lag might be small enough to set it on
                both aircrafts to "0". It's intended, that this is done
                automatically in the future.

tow: The tow used for aerotow or winch. This must be a subelement
                of an enclosing <hitch> tag.
         length: upstretched length in m
         weight-per-meter: in kg/m
         elastic-constant: lower values give higher elasticity
         break-force: in N
         mp-auto-connect-period: the every x seconds a towed multiplayer
                aircraft is searched. If found, this tow is connected
```

274

automatically, parameters are copied from the other
aircraft. Should be set only in the motor aircraft, not
in the glider

winch: The tow used for aerotow or winch. This must be a subelement
of an enclosing <hitch> tag.
max-tow-length:
min-tow-length:
initial-tow-length: all are in m. The initial tow length also
defines the length/search radius used for the mp-autoconnect
feature
max-winch-speed: in m/s
power: in kW
max-force: in N

rotor: A rotor. Used for simulating helicopters. You can have one, two
or even more.
There is a drawing of a rotor in the Doc-directory
(README.yasim.rotor.png) Please find the measures from this drawing
for several parameters in square brackets [].
If you specify a rotor, you do not need to specify a wing or hstab,
the settings for approach and cruise will be ignored then. You have
to specify the solver results manually. See below.
The rotor generates downwash acting on all stabs, surfaces and
fuselages. For all fuselages in the rotor downwash you should
specify idrag="0" to get realistic results.

name: The name of the rotor.
(some data is stored at /rotors/name/)
The rpm, cone angle, yaw angle and roll angle are stored
for the complete rotor. For every blade the position
angle, the flap angle and the incidence angle are stored.
All angles are in degree, positive values always mean "up".
This is not completely tested, but seem to work at least
for rotors rotating counterclockwise.
A value stall gives the fraction of the rotor in stall
(weighted by the fraction the have on lift and drag
without stall). Use this for modifying the rotor-sound.
x,y,z: The position of the rotor center

nx,ny,nz: The normal of the rotor (pointing upwards, will be
        normalized by the computer)
fx,fy,fz: A Vector pointing forward, if not perpendicular to the
        normal it will be corrected by the computer
diameter: The diameter in meter [D]
numblades: The number of blades
weightperblade: The weight per blade in pounds
relbladecenter: The relative center of gravity of the blade. Maybe
        not 100% correct interpreted; use 0.5 for the start and
        change in small steps [b/R]
chord: The chord of the blade its base, along the X axis
          (not normal to the leading edge, as it is
          sometimes defined). [c]
twist: The difference between the incidence angle at the
          blade root and the incidence angle at the wing
          tip. Typically, this is a negative number so
          that the rotor tips have a lower angle of attack.
taper: The taper fraction, expressed as the tip chord
          divided by the root chord. A taper of one is a
          bar blade, and zero would be a blade ending
          at a point. Defaults to one. [d/c]
rel-len-where-incidence-is-measured: If the blade is twisted,
          you need a point where to measure the incidence angle.
          Zero means at the base, 1 means at the tip. Typically
          it should be something near 0.7
rel-len-blade-start: Typically the blade is not mounted in the
        center of the rotor [a/R]
rpm: rounds per minute.
phi0:   initial position of this rotor
ccw: determines if the rotor rotates clockwise (="0") or
          counterclockwise (="1"), (if you look on the top of the
          normal, so the bo105 has counterclockwise rotor).
          "true" and "false" are not any longer supported to
          increase my lifespan. ;-)
maxcollective: The maximum of the collective incidence in degree
mincollective: The minimum of the collective incidence in degree
maxcyclicele: The maximum of the cyclic incidence in degree for
        the elevator like function
mincyclicele: The minimum of the cyclic incidence in degree for
        the elevator like function

maxcyclicail: The maximum of the cyclic incidence in degree for
         the aileron like function
mincyclicail: The minimum of the cyclic incidence in degree for
         the aileron like function
airfoil-incidence-no-lift: non symmetric airfoils produces lift
         with no incidence. This is is the incidence, where the
         airfoil is producing no lift. Zero for symmetrical airfoils
         (default)
incidence-stall-zero-speed:
incidence-stall-half-sonic-speed: the stall incidence is a function
         of the speed. I found some measured data, where this is
         linear over a wide range of speed. Of course the linear
         region ends at higher speeds than zero, but just
         extrapolate the linear behavior to zero.
lift-factor-stall: In stall airfoils produce less lift. Without
         stall the c-lift of the profile is assumed to be
         sin(incidence-"airfoil-incidence-no-lift")*liftcoef;
         And in stall:
         sin(2*(incidence-"airfoil-incidence-no-lift"))*liftcoef*...
         ..."lift-factor-stall";
         Therefore this factor is not the quotient between lift
         with and without stall. Use 0.28 if you have no idea.
drag-factor-stall: The drag of an airfoil in stall is larger than
         without stall.
         Without stall c-drag is assumed to be
         abs(sin(incidence-"airfoil-incidence-no-lift"))...
         ..*dragcoef1+dragcoef0);
         With stall this is multiplied by drag-factor
stall-change-over: For incidence<"incidence-stall" there is no stall.
         For incidence>("incidence-stall"+"stall-change-over") there
         is stall. In the range between this incidences it is
         interpolated linear.

pitch-a:
pitch-b: collective incidence angles, If you start flightgear
         with --log-level=info, flightgear reports lift and needed
         power for theses incidence angles
forceatpitch-a:
poweratpitch-b:
poweratpitch-0: old tokens, not supported any longer, the result are

not exactly the expected lift and power values. Will be
removed in one of the next updates.directly.Use "real"
coefficients instead (see below) and adjust the lift with
rotor-correction-factor.

The airfoil of the rotor is described as follows:
The way is to define the lift and drag coefficients directly.
Without stall the c-lift of the profile is assumed to be
        sin(incidence-"airfoil-incidence-no-lift")*liftcoef;
And in stall:
        sin(2*(incidence-"airfoil-incidence-no-lift"))*liftcoef*...
        ..."lift-factor-stall";
Without stall c-drag is assumed to be
        abs(sin(incidence-"airfoil-incidence-no-lift"))...
        ..*dragcoef1+dragcoef0);
See above, how the coefficients are defined with stall.
The parameters:
airfoil-lift-coefficient: liftcoef
airfoil-drag-coefficient0: dragcoef0
airfoil-drag-coefficient1: dragcoef1
        To find the right values: see README.yasim.rotor.ods
        (Open Office file) or README.yasim.rotor.xls (Excel
        file). With theses files you can generate graphs of the
        airfoil coefficients and adjust the parameters to match
        real airfoils. For many airfoils you find data published
        in the internet. Parameters for the airfoils NACA 23012
        (main rotor of bo105) and NACA 0012 (tail rotor of bo105?)
        are included.

rotor-correction-factor:
        If you calculate the lift of a heli rotor or even of a
        propeller, you get a value larger than the real measured
        one. (Due to vortex effects.) This is considered in the
        simulation, but with a old theory by Prantl, which is known
        to give still too large. This is corrected by this token,
        default: 1
flapmin: Minimum flapping angle. (Should normally never reached)
flapmax: Maximum flapping angle. (Should normally never reached)
flap0:   Flapping angle at no rotation, i.e. -5
dynamic: this changes the reactions speed of the rotor to an input.

normally 1 (Maybe there are rotors with a little faster reaction, than use a value a little greater than one. A value greater than one will result in a more inert, system. Maybe it's useful for simulating the rotor of the Bell UH1

rellenflaphinge: The relative length from the center of the rotor to the flapping hinge. Can be taken from pictures of the helicopter (i.e. 0 for Bell206, about 0.05 for most rotors) For rotors without flapping hinge (where the blade are twisted instead, i.e. Bo 105, Lynx) use a mean value, maybe 0.2. This value has a extreme result in the behavior of the rotor [F/r]

sharedflaphinge: determines, if the rotor has one central flapping hinge (="1") for the blades (like the Bell206 or the tail rotor of the Bo 105), default is "0".

delta3: Some rotors have a delta3 effect, which results in a decreasing of the incidence when the rotor is flapping. A value of 0 (as most helicopters have) means no change in incidence, a value of 1 result in a decreases of one degree per one degree flapping.
So delta3 is the proportional factor between flapping and decrease of incidence. I.e. the tail rotor of a Bo105 has a delta3 of 1.
In some publications delta3 is described by an angle. The value in YASim is the atan of this angle

delta: A factor for the damping constant for the flapping. 1 means a analytical result, which is only a approximation. Has a very strong result in the reaction of the rotor system on control inputs.
If you know the flapping angle for a given cyclic input you can adjust this by changing this value. Or if you now the maximum roll rate or ...

translift-maxfactor: Helicopters have "translational lift", which is due to turbulence. In forward flying the rotor gets less turbulence air and produces more lift. The factor is the quotient between lift at high airspeeds to the lift at hover (with same pitch).

translift-ve: the speed, where the translational lift reaches 1/e of the maximum value. In m/s.

ground-effect-constant: Near to the ground the rotor produces more

```
                    torque than in higher altitudes. The ground effect is
                    calculated as
                    factor = 1+diameter/altitude*"ground-effect-constant"
number-of-parts:
number-of-segments: The rotor is simulated in "number-of-parts"
                    different directions.
                    In every direction the rotor is simulated at
                    number-of-segments points. If the value is to small, the
                    rotor will react unrealistic. If it is to high, cpu-power
                    will be wasted. I now use a value of 8 for
                    "number-of-parts" and 8 for number-of-segments for the main
                    rotor and 4 for "number-of-parts" and 5 for
                    "number-of-segments" for the tail rotor.
                    "number-of-parts" must be a multiple of 4 (if not, it
                    is corrected)
cyclic-factor: The response of a rotor to cyclic input is hard to
                    calculate (its a damped oscillator in resonance, some
                    parameters have very large impact to the cyclic response)
                    With this parameter (default 1) you can adjust the
                    simulator to the real helo.
downwashfactor: A factor for the downwash of the rotor, default 1.
balance: The balance of the rotor. 1.0: the rotor is 100% balanced,
                    0.0: half of the blades are missing. Use a value near one
                    (0.98 ... 0.999) to add some vibration.
tiltcenterx:
tiltcentery:
tiltcenterz: The center for the tilting of the complete rotorhead/
                        mast. Can be used for simulating of the Osprey or small
                        autogyros.
mintiltyaw:
mintiltpitch:
mintiltroll:
maxtiltyaw:
maxtiltpitch:
maxtiltroll: The limits (in degree) for tilting the rotor head

All rotor can have <control> subelements for the cyclic
(CYCLICELE, CYCLICAIL) and collective (COLLECTIVE) input.
and can have <control> subelements for the tilting the whole rotor
head around the y-axis (TILTPITCH), the x-axis (TILTROLL) and the
```

z-axis (TILTYAW). ROTORBALANCE is a factor for the balance.

rotorgear: If you are using one or more rotors you have to define a
rotorgear. It connects all the rotors and adds a simple engine.
In future it will be possible, to add a YASim-engine.
max-power-engine: the maximum power of the engine, in kW.
engine-prop-factor: the engine is working as a pd-regulator. This
is the width of the regulation-band, or, in other words,
the inverse of the proportional-factor of the regulator.
If you set it to 0.02, than up to 98% of the rotor-rpm
the engine will produce maximum torque. At 100% of
the engine will produce no torque. It is planned to use
YASim-engines instead of this simple engine.
engine-accel-limit: The d-factor of the engine is defined as the
maximum acceleration rate of the engine in %/s,
default is 5%/s.
max-power-rotor-brake: the maximum power of the rotor brake, in kW
at normal rpm (most? real rotor brakes would be overheated
if used at normal rpm, but this is not simulated now)
rotorgear-friction: the power loss due to friction in kW at normal
RPM
yasimdragfactor:
yasimliftfactor: the solver is not working with rotor-aircraft.
Therefore you have to specify the results yourself.
10 for drag and 140 for lift seem to be good starting
values. Although the solve is not invoked for aircraft
with at least one rotor, you need to specify the cruise
and the approach settings. The approach speed is needed to
calculate the gear springs. Use a speed of approx. 50knots.
They do not need to match any real value.

The rotorgear needs a <control> subelement for the engine
(ROTORGEARENGINEON) and can have further <control> subelements:
ROTORBRAKE: rotor brake
ROTORRELTARGET: the target rpm of the engine relative to
the "normal" value for the governor. Default is
1.
ROTORENGINEMAXRELTORQUE: the maximum torque of the engine
relative to the torque defined by the engine-
power. Default is 1. By setting the rel-target

to a large number you get control over the
engine by this control.
Alternatively you can use these two values for
individual start-up sequences (see the s58)