Orbiter Payload Handler

Developer Documentation

Contents

Description

The Orbiter Payload Handler (OPH) is designed to be a resource to developers wishing to implement payload functionality for their vessels. OPH will load, store, attach, alter, and release payloads through a series of methods that can be called from the parent vessel, freeing the developer from worrying about the specifics of payload management. The vessel developer does not have to worry about dialogs as OPH is distributed with an Orbiter Module that implements a dialog. The dialog can be redistributed to the end-user.

How to use

To use OPH, simply point your compiler's “additional include directories” to the inc folder of this package, your compiler's “additional library directories” to this package's lib directory and type “OPHandler.lib” in your compiler's “additional dependencies” property. Then have your source code #include “OPHandler.h.” Currently, only Visual Studio 2012 will link with the library.

Method Descriptions

OPHandler(void)

Default constructor. Constructs a base OPHandler class, sets the payload origination position to 0,0,0 and the payload mass to 0.0kg.
NOTE: If this constructor is used, a call to setParent MUST be performed later, before the OPMS is used.

OPHandler(VESSEL3* v)

Constructor which takes a pointer to the instance of the vessel using the payload management system, otherwise, performance is the same as above.

OPHandler(VESSEL3* v, VECTOR3 pos)

Constructor which takes a pointer to the instance of the vessel using the payload management system, as well as a VECTOR3 which defines the origin point for payloads. This has the effect of offsetting the default payload position from the center of the parent vessel.

void setParent(VESSEL3* v)

Gives the OPH a pointer to the vessel using the OPH. This method must be called if the default constructor is used.

void setPos(VECTOR3* v)

This method will set an offset position where to add payloads when adding from the scenario file where no position is individually specified for each payload.

void parseScn(char* line)

This method takes a string and will look for a payload definition line of the following format:
PAYLOAD <string><int> <float> <float> <float> <float> <float> <float> <float> <float> <float> <float>
The string is the payload name, the in is for determining whether to add forces to the parent, the first three floats are the position of the payload, the second three are the jettison direction, the final three are the direction the payload faces independent of the jettison direction.
If PAYLOAD is included with a scenario with no other variables or an invalid name, then an error will be reported in Orbiter.log. At minimum, a PAYLOAD tag needs the name of a payload to attach.

void savePayloads(FILEHANDLE scn)

This method should be called in the vessel's clbkSaveState method and will create a PAYLOAD entry in the scenario file for each payload still attached to the parent.

void postCreation(bool s)

This method should be called from the vessel's clbkPostCreation(void) function. Currently, the boolean parameter is not used (and may be eliminated). This method sets up the payloads and is where attachment point definition and payload attachment is performed. If there is no vessel that matches the name from a PAYLOAD tag, then that payload definition will be discarded and a log entry made in Orbiter.log.

preStep(int n)

This method should be called by the parent vessel in its own clbkPreStep(...) function. n is a flag value, if it's a 1, then both the masses and forces of payloads will be updated. If it's a 2, then only forces will be updated. If it's a 3, then only masses will be updated.

int handleGeneric(int msgid, int prm, void* context)

This method should be called by the parent vessel in its own clbkGeneric(...) function. This method is designed to handle communication between vessels, and between the payload dialog box and the parent vessel. Simply have your vessel's clbkGeneric(...) return this method. If your vessel has its own

communications messages too, this should be called and returned after the last check.

bool transferPayload(OBJHANDLE o, char* c)

This method will transfer vessel c to the vessel defined by OBJHANDLE o. This method will return true if it was successful, or false if not. The vessel o should be a vessel that implements this OPH class, however transfer to a (valid) vessel that does not OPH should not cause a crash.

bool transferIn(payloadinfo *p)

This method is to be called by a receiving vessel in the handleGeneric(...) method if it accepts payloads transferred to it. This method will return true if it was successfully transferred in and attached to the new parent vessel, and false if it was not. Currently, this method simply positions the payload at the same relative position to the receiving vessel.

void addSecEdPgs(HWND hEditor, OBJHANDLE hVessel, HINSTANCE hDLL)

This method should be called from your vessel's secInit() method. This adds the payload handling dialog button to your vessel.
THIS METHOD IS OBSOLETE SHOULD NEVER BE CALLED BY ANYTHING EVER

bool jettisonPayload(char* n)

This method jettisons the vessel specified at the velocity unique to each payload.

bool jettisonPayload(char* n, double v)

This method jettisons the payload with the name of n at a velocity of v meters per second in the jettison direction. Returns false if unsuccessful.

bool jettisonPayload(char* n, bool v)

This method selects whether to jettison a vessel with its own velocity, or at 0 velocity relative to the parent.

bool jettisionPayload(int n, double v)

This method jettisons the payload at the slot position n at a velocity of v meters per second in the jettison direction. Returns false is unsuccessful.

void listPayloads(char** c, int n)

c should be an array of char*'s large enough to hold n payloads. If the number of payloads is less than n, then only n payloads will be placed in the array. If the number of payloads is greater than n, then the remaining array positions will simply not be written to.

bool addPayload(OBJHANDLE p, VECTOR3 v, VECTOR3 r,

VECTOR3 j)

This method is used when the developer wants to add a vessel payload that already exists in-sim. p should be a valid OBJHANDLE of a vessel, v is the position relative to the parent vessel that the payload should be added at. r is a VECTOR3 of the angle around the axes which the payload points in radians. j is a VECTOR3 of the angle around the axes which the jettision direction is, in radians.

bool addPayload(payloadinfo* p)

This method adds a payload directly from a payloadinfo struct. Primarily used within the handleGeneric function to handle transferring payloads between vessels.

VECTOR3 getPayloadPosition(char* c)

This method gets the position of a payload.

VECTOR3 getPayloadRotation(char* c)

This method gets the rotation of a payload about each axis in radians.

VECTOR3 getPayloadJettrot(char* c)

This method gets the jettison rotation of a payload about each axis in radians.

void getPayloadPositions(char **c, VECTOR3 v[], VECTOR3 r[], VECTOR3 j[], unsigned int n)

This method will give the positions, and rotations of each vessel attached. C, v, r, and j should be arrays large enough to hold n values.

double getPayloadMass(char* c)

This method returns the mass of the payload named c.
Returns -1 if c was not found.

double getTotaPayloadMass(void)

This method returns the total mass of the attached payloads.

int getPayloadCount(void)

This method returns the number of payloads currently attached.

int getPayloadJettvel(char* c)

This method returns the jettison velocity of the payload.

void changePayloadCoords(char* c,VECTOR3 pos,VECTOR3 r,

VECTOR3 j)

This method is used to change the location, jettision angle, or angle of a payload.

void setJettVel(char* n, double v)

This method is used to set the jettision velocity of an individual payload.

void setAddForce(char* n,int a)

Directs that the vessel of name n shall have its aerodynamic and thruster forces added to the parent vehicle if the value of a is not zero.

void setAddForceGlobal(int a)

Will direct that all vessels shall have their aerodynamic and thruster forces added to the parent vehicle if the value of a is not zero.

bool isPayload(char* n)

Returs whether the vessel n is a payload.

OBJHANDLE getOBJHANDLE(char* n)

This method returns the OBJHANDLE of the payload n. If n is not found in the payloads, NULL is returned.

Use Examples

The following examples show how to use this system.

```
//Creates a private instance of an OPMS class (currently manager)
Class MyVessel: public VESSEL3
{
	public:
		MyVessel(OBJHANDLE hVessel, int flightmodel);
		~MyVessel();
		…
	private:
		OPHandler m;
};
```

```
//Passes each line to the Manager class for processing
void MyVessel::clbkLoadStateEx(FILEHANDLE scn, void *status)
{
	char *line;
```

```
        while(oapiReadScenario_nextline(scn,line))
        {
                if(!_strnicmp(line, "CUSTOMPARAM",11))
                {
                        …
                }
                else
                {
                        m.parseScn(line);
                        ParseScenarioLineEx(line,status);
                }
        }
}
```

```
//saveState function
void MyVessel::clbkSaveState(FILEHANDLE scn)
{
        VESSEL3::clbkSaveState(scn);
        m.savePayloads(scn);
}
```

```
//Getting payload positions
char** payloads = new char*[m.getPayloadCount()];
VECTOR3* v = new VECTOR3[m.getPayloadCount()];
VECTOR3* r = new VECTOR3[m.getPayloadCount()];
VECTOR3* j = new VECTOR3[m.getPayloadCount()];
m.getPayloadPositions(payloads,v,r,jm.getPayloadCount());
```

```
//clbkGeneric
int Prometheus::clbkGeneric(int msgid, int prm, void* context)
{
       switch(msgid)
       {
          case somemsg:
               //do stuff
               break;
          ..
          default:
               return m.handleGeneric(msgid,prm,context);
               break;
       }
}
```

Advanced Use Ideas

One of the advantages of using a class to manage payloads is that multiple payload storage systems could be defined. Perhaps the developer can designate certain kinds of payloads for fuel, then others for oxygen, then maybe another type for an electrical system. With a single payload "container" it would be hard to designate individual payloads without a lot of cross-checking for types when they

need to be used. But creating separate payload storage containers for each types means it's a simple matter to use and update the payloads in each.

This system can also be used to simulate launch escape systems. The transfer methods were written with this in mind.

Credits

This system would not have been possible without the help of some very important people:

Izack of Orbiter-Forum for coming up with the name, Orbiter Payload Handler for this system.

Douglas Beachy (dbeachy1) of Orbiter-Forum for providing untold amounts of help with C++, libraries and various programming topics.

Face of Orbiter-Forum for various help throughout the years and in particular for providing a solution for passing data via clbkGeneric.

Woo482 of Orbiter-Forum for testing OPH at various stages.

Yuri Kulchitsky (Kulch) of the Orbiter community for creating the original Payload Manager which is still a great system, and the inspiration for the creation of OPH.

Interceptor of Orbiter-Forum for doing some end-user testing of OPH's dialog.

The entire Orbiter Spaceflight Simulator community for providing help along the way throughout the years.

Most importantly, Dr. Martin Schweiger for creating the wonderful Orbiter Spaceflight Simulator for which this system is designed to operate with.