Camera MFD API

## Introduction

The API works by the MFD sending a pointer to itself to the vessel’s clbkGeneric method, along with the MFD index. The vessel receives the pointer and castes it to the API interface, then it can control the MFD. Since clbkGeneric is used, the vessel class must be VESSEL3 or higher.

When the MFD is destroyed, it sends a null pointer with the same index to clbkGeneric, thus the vessel should no longer use this pointer. The camera control is accomplished by the CameraData struct.

## Implementation

To use the API, include the API file in the vessel header file, implement clbkGeneric if not already implemented, and declare 2 arrays. One to store the MFD instances, and the other to store the CameraData struct to control the camera. The MFD instances array size should be set according to the number of MFDs you wish to control (in this example the 2 main MFDs), while the CameraData structs array should be set to how many cameras you wish to create and control (4 cameras in this case, 2 for each MFD).

#include <CameraMFD\_API.h>

class Vessel : public VESSEL4

{

public:

int clbkGeneric(int msgid, int prm, void \*context);

private:

CameraMFD \*cameraMFD[2];

CameraMFD::CameraData cameraData[4];

};

In the vessel class constructor, set both MFD instances to nullptr. This is done so you can check if the instance is initialized before calling the API.

Vessel::Vessel(OBJHANDLE hVessel, int flightmodel) : VESSEL4(hVessel, flightmodel)

{

cameraMFD[0] = cameraMFD[1] = nullptr;

}

In clbkGeneric, handle the Camera MFD messages. You must return CAMERA\_MFD when you receive the instance and control it.

int Vessel::clbkGeneric(int msgid, int prm, void \*context)

{

// Check if the message is from Camera MFD, defined in the API file

if (msgid == CAMERA\_MFD)

{

// If the MFD instance is created

if (context)

{

// Check for MFD index for the MFD you want to control (0 here)

if (prm == 0)

{

// Cast the pointer to the API interface

cameraMFD[0] = static\_cast<CameraMFD\*>(context);

// Set the camera data for the first MFD

cameraData[0] = { "Camera 1", { 0,-1,-1.3 }, 0, 90, 0, 40, { true, true, true, true, true } };

cameraData[1] = { "Camera 2", { 0,-1,-1.3 }, 0, -90, 0, 40, { true, true, true, true, true } };

// If no data were found in the scenario

if (cameraMFD[0]->CameraDataExist())

cameraMFD[0]->SetCameraData(1, cameraData[1]);

else

cameraMFD[0]->AddCamera(1, cameraData[1]);

// The first camera will always exist whether there are data or not

cameraMFD[0]->SetCameraData(0, cameraData[0]);

return CAMERA\_MFD;

}

else if (prm == 1)

{

cameraMFD[1] = static\_cast<CameraMFD\*>(context);

cameraData[2] = { "Camera 3", { 0,-1,-1.3 }, 0, 0, 0, 40, { true, true, true, true, true } };

cameraData[3] = { "Camera 4", { 0,-1,-1.3 }, 0, 180, 0, 40, { true, true, true, true, true } };

if (cameraMFD[1]->CameraDataExist())

cameraMFD[1]->SetCameraData(1, cameraData[3]);

else

cameraMFD[1]->AddCamera(1, cameraData[3]);

cameraMFD[1]->SetCameraData(0, cameraData[2]);

return CAMERA\_MFD;

}

}

You must check if CameraDataExist returns false before adding new cameras as shown above, otherwise expect undesirable camera behavior.

// If the MFD instance is destroyed

else

{

// Set the MFD instance to nullptr

if (prm == 0)

cameraMFD[0] = nullptr;

else if (prm == 1)

cameraMFD[1] = nullptr;

}

}

return 0;

}

If you want to control the MFD in other places, you must check if the MFD instance is initialized (i.e. not a null pointer).

## User Control

If the user is allowed to control the position, pitch, yaw, rotation, and/or FOV of the camera, the user inputs are separated from the vessel inputs. Imagine that the vessel set the pitch angle to 30 degrees and the user increased it to 35 degrees. If the vessel sets the pitch angle to 35 degrees, the user 5 degrees will be added (so the total is 40 degrees). This applies to other parameters as well.

Regardless of the user control settings, the user can’t change the camera label or add/delete cameras, only the vessel can.