Aircraft Definition File (TMD-File)

The single text-file with the file extension `.tmd` defines an Aerofly FS 2 aircraft's physics characteristics, visuals/animations and sound definition. This file is located in the aircraft folder together with other files like the aircraft description file TMC.

> “THE” tmd-file of an aircraft is the file that has the file extension `.tmd` and the exact same file name as the folder it is in, e.g. 'aircraftname.tmd'

Other TMD files are the

- controls.tmd - click-spot definition
- aircraft_presets.tmd - preset state for takeoff, landing, clean

Purpose

The TMD-file contains the main definition of an aircraft. It describes the aircraft as a collection of objects that can have different attributes and connections to each other in an object-oriented approach. Among the objects that can be defined in a tmd file are engines, propellers, wings, airfoils, wheels, actuators, logic circuits, rigidbodies, joints, sound sources, instruments, autopilot, navigation receivers and many general purpose objects like integrals and mathematical functions.

Sections

The TMD file consist of three sections:

- DynamicObjects
- GraphicObjects
- SoundObjects

Here is an example for how the three sections are defined in the tmd text file.

```
<[file][][]
    <[modelmanager][][]
        <[pointer_list_tmuniverse][DynamicObjects][]

            // objects are added here

        >
        <[pointer_list_tmgraphics][GraphicObjects][]
        >
        <[pointer_list_tmsound][SoundObjects][]
        >
    >
>
```

DynamicObjects

The dynamics section defines all objects that create the dynamics and systems simulation for an aircraft. Physical values, such as the location of the aircraft, it's speed and orientation as well as the deflection of it's control surfaces, the rotation speed of engines or propellers, electrical loads, fuel consumption, system states and numerous other values are calculated numerically by Aerofly FS 2's physics engine.

The core of each aircraft in the Aerofly flight simulator is a rigidbody simulation. Each body has its own mass, position, velocity, orientation and angular velocity. Forces like gravity, aerodynamic forces, engine or propeller thrust and torque, collisions with the ground are computed and applied to the body. Joints connect different bodies and apply forces and torques to the bodies. In the tmd code the rigidbody system definition is usually positioned at the very beginning of the file. Further down in the dynamics section the aerodynamic properties (aerowings, airfoils) of the aircraft are defined, followed by the primary flight control deflections and more advanced aircraft systems like electrical, hydraulic or fuel systems. Simulated instruments like the attitude indicators, airspeed indicators and altimeters are towards the end tmd dynamics section.

The objects defined in the graphics and sound section use the output generated by the dynamics simulation to render the 3D objects and animations. Therefore it is necessary to hand over the required values to the graphics and sound using an output object. This includes the position of all control surfaces and primary flight controls as well as values that are visible on any display in the cockpit. An example can be found below.

Here is a list of pages that describe the systems within the dynamics section of the TMD file further:

- Rigid-Body-System
- Collision
- Aerodynamics
- Actuators
- Propulsion
- Instrumentation
- Inputs (buttons, switches and knobs, flight controls, digital inputs)
- Math and Logic Circuits
- Event Based Programming
- Electrical Systems
- Fuel Systems
- Hydraulic Systems
- Autopilot and Flight Management

GraphicObjects

The second section of the tmd file is the graphics section. This section defines the rendering of all cockpit displays and each visible part of the 3D model and its animation.

There are a number of different animation methods available. To name a few, there are static animations that just move all parts through the simulated world together with the fuselage's position (rigidbodygraphics), there are objects that rotate around a certain axis (hingedbodygraphics or rotation) like the elevator, aileron and rudder and there are objects that can be shifted along an axis like the throttle in the C172 (movinggraphics or translations).

Each object of the 3D model (geometry) that should be rendered to the screen must be placed in one and only one 'GeometryList'. For the beginning, all geometries can be added to a single rigidbodygraphics object's 'GeometryList' as shown in the example code below. Every object in the 3D model that is this list will be moved together with the 'Fuselage' rigidbody and will then be rendered by the graphics card if it is in the view of the camera.

```
            <[rigidbodygraphics][Fuselage][]
                <[uint32][PositionID][Fuselage.R]>
                <[uint32][OrientationID][Fuselage.Q]>
                <[string8][GeometryList][ Fuselage MainPanel Antenna ]>
            >
```

Later on in development, the individual parts will be moved from this 'GeometryList' to the 'GeometryList' of a new hingedbodygraphics, movinggraphics or animated rigidbodygraphics object.

The rendering engine itself can't directly access the user's control inputs or simulate the position of slow moving actuators; this is what the physics engine does. Every animation, therefore, needs to get their inputs from the dynamics section where all the physical movements or delays can be simulated. To send a value from the dynamics section to the graphics section the dynamics section needs to define an 'output' object as described above and the graphics section needs to have a graphics_input object whose InputID matches the 'Name' property of the output followed by the text '.Output'.

Collection of the pages relevant to the GraphicObjects section:

- Inputs (animation of buttons, switches and knobs)
- Rigidbodygraphics, transformations and bendingbodygraphics
- Interieur lighting and external lights
- Numerical Displays, LCDs, flight displays, HUDs
- Human character

SoundObjects

The third and last one is the sound section. The sound section also reads in values from the numerical simulation in the dynamics section and then uses a mix of interpolations to determine the pitch and volume of a particular sound file.

Similar to the graphics section a soundinput object will read the value of a dynamics output object and provide its value for the entire sound section.

Most relevant pages

- Button, switch and knob sounds triggered in DynamicObjects
- Sound input, mixing, mapping, fading, etc.

Object unique names

Each object in a section must have a unique name by which it can be referenced. Using a name for more than one object in a section can have undesired effects.

Naming Conventions

We use naming rules and give similar objects the same names across different airplanes:

- The name of an object uses the characters A-Z, a-z, 0-9 only and the first character of the name is always capitalized.
- Object names must not contain any spaces. Separate words by capitalizing the first letter of each word (UpperCamelCase style).
- Use underscores only if necessary.
- Hyphens, colons and commas outside of comments (//) must not be used.
- To assure cross-platform compatibility, the usage of tabulator indentation is deprecated. Use spaces for indentation instead. It is recommended to set the text editors settings to replace tabs with spaces automatically.
- Abbreviations are avoided, the full name is always preferred.

Object structure

The example code below creates an object of the class 'control_input' with the name 'ThrottleLever' in the dynamics section when the simulator loads the aircraft. During runtime, this object will filter out messages with the name 'Controls.Throttle1' which are typically sent by the control device assigned to the left throttle input in the 'Controls Settings' user interface. The control_input object will store the value of the latest message in memory for later use by other objects.

A message with a custom message name can also be generated by adding click spot to the 'controls.tmd' file.

```
<[file][][]
    <[modelmanager][][]
        <[pointer_list_tmuniverse][DynamicObjects][]

            // throttle
            <[input_lever][ThrottleInput][]
                <[string8][Input][Controls.Throttle1]>
            >

        >
    >
>
```

Object communication within a section

Most objects in the tmd file need to communicate in some form. For example, a jet_engine object needs to know the throttle lever position so that it can calculate the thrust increase or decrease over time and apply a variable strength force to the 'Fuselage' to accelerate it. In the tmd code snipped below, the developer defined that the engine should use the output of the ThrottleLever object as its ThrottleControl input. It would also be possible to use the autothrottle output as an example or any

other imaginable output from an instrument reading or an output of a logic circuit if that is desired.

```
            // example of the JetEngine getting the output of the
ThrottleLever object
            // throttle
            <[input_lever][ThrottleInput][]
                <[string8][Input][Controls.Throttle1]>
            >
            <[jet_engine][JetEngine][]
                <[string8][Body][Fuselage]>
                <[string8][ThrottleControl][ThrottleInput.Output]>
                <[float64][MaximumThrust][10000.0]>
            >
```

These manually programmed connections allow enormous flexibility, easy debugging and incredibly simple and fast object code. Each object can be as 'stupid' as can be, it just uses its inputs, does something with it and sets its output value or internal state if it has one. Then, if another object comes by and wants to know the output value, it just grabs it and calculates its outputs with that. This way very long chains, logic trees or even iterations and control loops can be created that actually perform different logic depending on the aircraft's state or the user's inputs. It's important to note that the order in which the objects appear in the tmd code has no effect on the simulation.

Object communication across sections

Objects defined in different sections of the tmd cannot communicate directly. The only possible connections are from the dynamics section to the graphics section and from the dynamics section to the sound section.

Almost all values used outside of the dynamics section will have to go through an 'output' object and need to be read in by any section that needs that value. The reason for that is the core design of the Aerofly FS 2 that strictly separates the physical simulation from the graphical representation and sound generation. Designing the graphics and sound sections as 'read-only' functions of physical values has quite a strong performance advantage and allows for a more flexible software design. The different processes of calculating the physics and calculating the representation of graphics model can be done on different CPU cores or even partly on the graphics card to avoid major bottlenecks.

One way to imagine the communication between the sections is that each value from the dynamics section is sent through a wire to either the graphics processing unit or the sound processing devices.

Example dynamics to graphics section

In the example below the position of the throttle lever input is put out by the dynamics section and then in the graphics section the value is read in. We usually give the 'graphics_input' object the same name as the 'output' from the dynamics section to avoid confusion.

```
<[file][][]
    <[modelmanager][][]
        <[pointer_list_tmuniverse][DynamicObjects][]
```

```
            // this object makes it possible to call
'ThrottleInputPosition.Output'
            // in a graphics_input (or soundinput) object as shown below
            <[output][ThrottleInputPosition][]
                <[string8][Input][ThrottleInput.Output]>
            >


        >
        <[pointer_list_tmgraphics][GraphicObjects][]

            // this object reads the output of the dynamics section in
            // and provides that value for other objects in the graphics
section
            <[graphics_input][ThrottleInputPosition][]
                <[string8][InputID][ThrottleInputPosition.Output]>
            >

        >
    >
>
```

Now, to animate the 3D model of the throttle lever, a 'hingedbodygraphics' object is added with its GeometryList containing the name of the geometry in the 3D model. It uses the ThrottleLeverPosition for its rotation angle input. The 'Axis' and 'Pivot' parameters can be measured in the 3D software or can be found in the exporter's log file 'tm.log' in the intermediate folder.

```
            <[graphics_input][ThrottleLeverPosition][]
                <[string8][InputID][ThrottleLeverPosition.Output]>
            >
            <[hingedbodygraphics][ThrottleLeverGraphics][]
                <[uint32][PositionID]   [Fuselage.R]>
                <[uint32][OrientationID][Fuselage.Q]>
                <[string8][GeometryList][ ThrottleLever ]>
                <[string8][InputAngle][ThrottleLeverPosition.Output]>
                <[tmvector3d][Axis] [ 0.0 1.0 0.0 ]>
                <[tmvector3d][Pivot][ 5.123 0.0 -0.321 ]>
                <[float64][AngleMax][0.5]>
            >
```

Summary

1. The 'control_input' object in the dynamics section reads the position of the user's control input device and makes this value available in the entire dynamics section as 'ThrottleLever.Output'.
2. An 'output' object with the name 'ThrottleLeverPosition' is declared in the dynamics section which makes its input value also available for 'graphics_input' and 'soundinput' objects in the graphics and sound section respectively.
3. In the graphics section, a 'graphics_input' receives the values sent by the dynamics. Now the value of the throttle lever angle is available as 'ThrottleLeverPosition.Output' for all objects in the graphics section.

4. The 'hingedbodygraphics' object rotates all geometries in its GeometryList depending on the 'ThrottleLeverPosition.Output' value.

Debugging

The most important resource for troubleshooting is the simulator's log file **tm.log** in the 'Documents/Aerofly FS 2' folder. This file logs all events during the Aerofly FS 2 Flight Simulator execution. It contains warnings and errors that might be generated when the new aircraft is loaded.

Common issues during aircraft development are:

- **Syntax** accidentally changed... pay particular attention to the braces and brackets, for each opening bracket/brace there has to be a closing one, too. Some text editors offer a counting function, which can help you trace down the position of the missing character.
- **Spelling mistakes**, Aerofly is case sensitive and does not allow spaces.
- **Same identifying name** used more than once per section
- **Infinite loops** created by using the output of the same object or by chaining object to a loop without any object in the chain that can provide a delay (like a servoclassic or servolinear).
- **Referenced object** does not exist (e.g. trying to call .Output on an object that is not defined in the scope of the same section)
- **3D object part used multiple times** in a section
- **Joint** (connection) force and dampening constants to high (will show black screen or aircraft explodes)
- **Output/Input** not defined or names not matching (link across sections broken)
- **Control message names** not matched (link from controls to dynamics broken)
- **Control message qualifier** 'toggle' not set for a toggle switch

To debug a longer chain of connected objects one can simply overwrite the input property of an object by replacing the usual xyz.Output text with '1.0', '0.0' or any other constant value. Note that inputs with an 'ID' in their name don't directly read the value but use a pointer object instead. It is possible to write a constant value into the InputID parameter of an object.

For debugging it is very useful to assign a key to the 'Reload aircraft' in the simulator control settings (available only if the 'developer' option is set to true in the 'main.mcf'). Using this function only the aircraft but no scenery is reloaded.