# Requirements/Needs

We have the need for a system that can handle the display of Vehicle Gauge information on top of existing video feeds. We also have the need to control the display of the video feeds as well as control other aspects of the user experience and send commands to the vehicle. The requirements exist on the Omniscape VDDS as well as on PC platforms. It is a goal that 90+% of the system should be source code compatible with any platform on which it is required to run.

# Analysis

It has been identified that the end-user (Marines) will most likely request numerous on-site changes to the visual layout and even the control features of the system. This one recognition implies that we cannot depend upon precompiled solutions such as we currently use within the Omniscape or most of our PC solutions.

Analyzing several different systems for flexible gauge creation and modification, we identified that Microsoft had created an XML based gauge system for Microsoft Flight Simulator. In addition to supporting gauge definition, their system also allows for user interaction. As an example, using the mouse to click on buttons, switches, or even drag certain elements of the gauges and controls which results in feedback into the simulator to control the application. After some further investigation (which included personally inspecting the gauge definitions within a personal copy Microsoft Flight Simulator) it has been determined that the XML schema provides a highly rich description of Gauges and their interactions.

Another advantage to using the literal XML schema from Microsoft Flight Simulator is that we can utilize some of their tools for building gauges. As such expect to see numerous references to the Microsoft Flight Simulator (hereafter referred to as simply “MSFS”). We will also refer to MSFS documentation to help document our Gauge system.

From the beginning we recognize that we will need to expand the definition of the MSFS Gauge concept. First we want to be uniform in our use of XML. Currently FS only uses XML to define an individual gauge, Microsoft does not use XML to identify the gauge placement on a panel. Some investigation indicated that MSFS uses an older legacy method of identifying panel placement. Plus in MSFS the user has the need to display a single panel (for an aircraft) at a time. While they allow for multiple panel definitions the user must switch aircraft to utilize an alternate panel.

# Building Blocks

To make things easier for the gauge and panel designer we want to layout the system with some easy to understand blocks.

## Gauges

At the molecular level we present gauges. A Gauge is a single unit defined with a single XML file. A Gauge can have graphic representation (but not required). The definition of that XML for the most part has already been declared within MSFS. The Gauge XML is documented at the Microsoft website at the following URL:

<http://msdn.microsoft.com/en-us/library/cc526953.aspx>

*UPDATE:*

*It turns out that Microsoft has two different flavors of XML. The first one was introduced with MSFS 2004, the second was introduced a couple of years later. The differences are significant relative to the actual schema but not the concepts. The earlier version uses XML attributes to contain all of the information. The schema for the second uses XML values (the stuff between the open and close tags), attributes are only used to provide metadata regarding the values. Our plan is to deliver using the later version, but right now we have a hybrid implementation between the two.*

## Panels

Just as in MSFS we group a single display of gauges onto a Panel. Only one panel is displayed at a time. Unlike MSFS we define our panels through XML. Panels define the following: a group of one or more gauge-instances, key mappings (what events are generated when a user presses keys or buttons). Panels may also define a script to be executed when an update is to be performed. Panels may also react to events. Panels also identify the placement rules for Gauges. Things like a gauge should align to the right side of the display and stack with any other gauges at the bottom. It also identifies whether it should automatically adjust its position relative to non-displayed gauges. As an example: if we have a gauge that has been declared as stacking at the bottom with any other gauges that stack, should it change its position on the display if any gauges below it are not being displayed.

## Screens

A screen defines a group of panels. One screen is active on a station/display at a time. As a result the screen fully identifies what panels (and subsequently, what gauges) are available. Like Panels, Screens may identify key mappings for events. They may also react to events.

## Dictionary

The Dictionary while not explicitly defined by MSFS is strongly implied with their form of “Variables”. Most distinct and obvious in MSFS is the “A:” variables (or “Aircraft” variables). These variables identify the active running conditions of the aircraft in MSFS. The specification also identifies other variable “types” such as “E:” Environment, “G:” Gauge specific variables, “L:” local (which is sort of funny because in all reality these are global to all gauges), “P:” program preferences and so forth. Through MSFS “variables” the gauge can request certain types of actions to be performed. These are done through the “K:” variables, which are known as “Key commands” in MSFS.

Just as in MSFS, in our system the Dictionary will be the primary method that is used for Gauges to interact with the outside application.

## Expressions

MSFS defines a reverse polish notation (also known as infix notation) for describing expressions. Reverse Polish notation is where the operands are identified followed by the operators. Programs such as LISP are well known for the RPN definition.

RPN is typically used with a stack based expression interpreter. What typically happens is that each operand is pushed on a stack, then when the operator is processed it pops from the stack as many operands as it requires; performs its operation and then pushes the result back onto the stack. While we will model our implementation after a stack model, the stack will only be used to parse the expression. Our runtime will be hooked together so we do not have to “interpret” the expression.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operator | Operation | Arguments | Example | Result |
| + | Addition | 2 | 3 5 + | 8 |
| - | Subtraction | 2 | (G:George) 90 - | Subtract 90 from the gauge variable George |
| / | Division | 2 |  |  |
| \* | Multiplication | 2 |  |  |
| % | Modulo | 2 |  |  |
| /-/  neg | Negate | 1 | 5 /-/ | -5 |
| ++ | Increment | 1 |  |  |
| -- | Decrement | 1 |  |  |
|  |  |  |  |  |
| &  &amp;  BITAND | Bitwise And | 2 | 5 3 & | 1 |
| |  BITOR | Bitwise Or | 2 | 5 3 | | 7 |
| ^  BITXOR | Bitwise Xor | 2 | 5 3 ^ | 6 |
| ~  BITNOT | Bitwise Not | 1 | 5 ~ | -6 |
|  |  |  |  |  |
| ==  EQ | True if Equal | 2 |  |  |
| !=  NE | True if Not Equal | 2 |  |  |
| >  &gt;  GT | True if Greater | 2 |  |  |
| <  &lt;  LT | True if Less than | 2 |  |  |
| >=  &gt;=  &ge;  GE | True if Greater than or Equal | 2 |  |  |
| <=  &lt;=  &le;  LE | True if Less than or Equal | 2 |  |  |
| ? | Third operand determines whether the first (True) or second (False) is selected | 3 | true false condition ? |  |
| !  NOT | Invert test | 1 |  |  |
| &&  AND | True if both operands are true | 2 |  |  |
| ||  OR | True if at least one of the operands is true | 2 |  |  |
| rng  range | Returns true if the third operand lies between the values of first and second operands | 3 |  |  |
| case | Case statement | variable | 50 40 30 20 10 5 (G:George) case | The “5” indicates that there are five values, which are selected depending on the evaluation of (G:George). If the evaluation is equal to or greater than 0 but less than 1, then result is 10. If the evaluation is equal to or greater than 1, but less than 2, the result is 20, and so on. |
| Find | Match the argument to a list and return the index | variable | ‘alpha’ ‘beta’ ‘gama’ 3 (G:George) find | The “3” indicates the number of arguments to be searched. Careful because the index is in the reverse order: matching ‘gama’ will return a zero. |
|  |  |  |  |  |
| abs | Absolute value | 1 |  |  |
| max | Maximum of operands | 2 |  |  |
| min | Minimum of operands | 2 |  |  |
| flr  floor | Nearest integer number which is less than or equal to the operand | 1 |  |  |
| ceil | Nearest integer number which is greater than or equal to the operand | 1 |  |  |
| near | Nearest integer number (rounds) | 1 |  |  |
| dgrd | Converts degrees to radians | 1 |  |  |
| rddg | Converts radians to degrees | 1 |  |  |
| dnor | Normalizes the operand to 360 degrees | 1 |  |  |
| rnor | Normalizes the operand to radians for a circle (2\*pi) | 1 |  |  |
| sqr square | Operand multiplied by itself | 1 |  |  |
|  |  |  |  |  |
| ind indirect  @ | Uses the operand to evaluate a variable name | 1 | “A:RPM” ind | Results in retrieving from the Dictionary the variable (A:RPM) |
| Indasn indirectassign  =@ | Uses the second operand to evaluate a variable name to which it assigns the first operand | 2 | 25 “U:George” =@ | Results in assigning to variable (=U:George) the value of 25 |
| format | Format the input operand according to a printf style format string. The format string MUST be a literal value. | 2 | (G:George) “%d” format | Results in the contents of (G:George) being evaluated as an integer and formatted accordingly. |
| scat  strcat | String concatenation | 2 | “abc” “def” strcat | “abcdef” |
|  |  |  |  |  |
| rgb | Make a color value based on red, green & blue | 3 | 255 128 64 rgb 1.0 0.5 0.25 rgb |  |
| rgba | Make a color value based on red, green, blue and alpha | 4 |  |  |
| color | Make a color value based on either a quoted name or a numeric color value | 1 | ‘purple’ color 0x00FFCC33 color |  |
| colora | Make a color value based on either a quoted name or a numeric color value in addition to a translucency alpha value | 2 | ‘purple’ 127 colora  ‘purple’ 0.5 colora |  |
|  |  |  |  |  |
| unitcnv | Convert input operand to requested units | 2 | (A:Vehicle Speed) ‘k/h’ unitcnv |  |
| unitname | Get the character string representing the unit name of the input operand | 1 | (A:Vehicle Speed) unitname | Results in “mile per hour” |
| unitplural | Get the character string representing the plural unit name of the input operand | 1 | (A:Vehicle Speed) unitplural | Results in “miles per hour” |
| unitabbrev | Get the character string representing the abbreviated unit name of the input operand | 1 | (A:Vehicle Speed) unitabbrev | Results in “MPH” |
|  |  |  |  |  |
| both | Process both operands but only return the second | 2 |  |  |
|  |  |  |  |  |
| height | Height in pixels of the current gauge | 0 |  |  |
| width | Width in pixels of the current gauge | 0 |  |  |
| top | Pixel position of the gauge top | 0 |  |  |
| left | Pixel position of the left side of the gauge | 0 |  |  |
|  |  |  |  |  |
| true |  | 0 |  |  |
| false |  | 0 |  |  |
| pi |  | 0 |  |  |
| pi2 | 2 times pi | 0 |  |  |

# Building Objects

The XML in our Gauges Interface System defines a set of objects (C++ variety) as well as the relationships between them. We will not have run-time interpretation of any XML. *That last statement is not completely true – the “active vehicle” parameters are communicated to the system via XML. But within the running gauges/panels/screen we will not be interpreting XML after the system is loaded*.

Because we are using MSFS as our model for the XML we also get the benefit of using (stealing) what is implied for their objects. What we have is two basic types of information described in the XML: Properties (please don’t confuse these with XML attributes) and Property Sets. In simple terms Properties are XML tags that define a value or expression and Property sets are the XML tags that contain either properties or other tags that contain properties. So an Easy way to think about it is that Property Sets equate to Objects (and in most cases they do). And Properties equate to member variables in the Objects. And when you have XML for a property set that is containing other property sets then you have implied a relationship between objects.

I want to provide an example, but before we cite that example, I want to be clear that my purpose is not to go into detail describing the XML (MSFS has already done a good job of that), but to identify what are properties and what items are property sets.

<?xml version=”1.0”?>

<SimBase.Document Type="AceXML" version="1,0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="Gauges2.xsd">

<SimGauge.Gauge id=”somename”>

<Size>200,200</Size>

<Element>

<FloatPosition>100,100</FloatPosition>

<Circle>

<Axis>100,100</Axis>

<Radius>100</Radius>

<FillColor>silver</FillColor>

<LineWidth>2</LineWidth>

<LineColor>black</LineColor>

</Circle>

<Element>

<Rotation>

<Expression>

<Script>(A:RPM) 6000 / 360 \* dnor</Script>

</Expression>

</Rotation>

<Line>

<LineWidth>5</LineWidth>

<LineColor>red</LineColor>

<Point2>0,0</Point2>

<Point2>95,0</Point2>

</Line>

<Element>

</Element>

</SimGauge.Gauge>

</SimBase.Document>

In the previous example an easy way to tell which tags represent properties and which represent property-sets is that properties contain data (text or numbers). Those tags that contain other tags are property-sets. Most property sets equate to objects.

So to cite the actual objects from this example: a “Gauge” that owns an “Element”, that element owns a “Circle” and another “Element”, and that element owns a “Rotation” object and a “Line”. The remaining tags are properties of the objects in which they are nested.

# RPN Expressions

MSFS implements a stack-based RPN Expression system. Instead of implementing a run-time stack we should create an execution tree. If you are familiar with language parsing; you should be familiar with expression trees. A simple explanation of an expression tree is node tree where each node in the tree is a component of the expression. A simple example is the addition operator. The addition operator would be a single node with two children; the left-land operand and the right-hand operand. In code form the addition operator’s GetValue function would look something like:

float Addition::GetValue()

{

float fLeft = m\_pLeft->GetValue();

float fRight = m\_pRight->GetValue();

return fLeft + fRight;

}

The preceding example is an over-simplification of the solution, but it does show you how simple many of the operations can be. By hooking the nodes together in various combinations we can achieve most any expression. And best of all, the run-time performance is exceptional.

# Variances

Yes, we are basing our Gauge Interface System on MSFS, but how are we going to need to extend or be different from MSFS.

## Expressions

Primarily for time we will not be implementing the entire set of operators that MSFS has defined. Most notably we will not implement the “if/else” operator. We will need to make use of the question “?” operator instead.

## Dictionary Pages:

In MSFS the variable dictionary is divided into logical pages. The page is identified by specifying a single letter prefix followed by a colon before the actual name of the variable. The following is a listing of the pages that we should support and their meanings.

* A: -- Active Engineering Data. These include things like: engine speed (RPM), vehicle speed, engine coolant temperature, etc.. This data is refreshed every 100 milliseconds (which is configurable). The page is considered to be readonly for the Gauges.
* B: -- Baseline meta-data for the “A:” page. Entries include things like the high/low range of the variable, if the variable has yellow or red ranges those are identified as well. Also any labels are included. The naming convention should be some like: base-variable name followed by a period followed by the name of the meta-data. For instance if we had a Variable named “George” (I know, that is not very realistic, but work with me here), “George” has a numeric range of 0 to 300 and any value above 250 is considered a “Red” condition. The meta-data names would look something like: B:George.Low, B:George.High, B:George.Green.Low, B:George.Green.High, B:George.Red.Low, B:George.Red.High. The data in the “B:” page is created and populated during system startup and does not change thereafter.
* C: -- Command requests to the running system. The running system identifies the set of legal commands. Please be aware that any command request that is initiated should be treated as asynchronous. This page is treated as write-only by the Gauges.
* D: -- Data/Response. The running system identifies the state of various parts of the system.
* E: -- Events – Any expression as the ability to create named events. The event can have a single data item. Any object that responds to events may then read the data associated with the event.
* G: -- Gauge local variables. Gauges may create variables that are persistent between calls. We will extend the concept of the MSFS’ “G:” variables in that a Gauge may create as many G-Variables as it likes and we will support any of the Variant types.
* K: -- Constants that are typically used for communicating with the Command page (C:) or responses in the Data/Response page (D:). Once this page has been populated on startup then it should be treated as read-only.
* M: -- Mouse/Touch Screen data. This page contains details about the current or most recent mouse activity. Variables are: (M:X), (M:Y), (M:Event). M:Event may have values of “Down”, “Up”, or “Drag”.
* N: -- Notifications/Alerts – when the Vehicle Health system needs to identify an alert, that information is recorded in the “N:” page and an event is generated.
* P: -- Program preferences – This page is actually loaded from the configuration xml file.
* S: -- System variables – these include things like the current system time, milli-time, etc.
* U: -- Universal variables – include any variable that any Gauge, Panel, or Screen chooses to create. These are similar to “G:” variables except that all components in the system that have scripts may access them.

## Extended Objects

MSFS provides a very rich set of objects within their schema for Gauges. However, in some cases we will need to extend their definition or meaning.

### Select/Case

There are two primary differences between the MSFS definition and ours. The first is that we allow a <CaseElse> tag that provides a default case to handle the non-matching situation. The second is that the value on the <Case> for MSFS is always a constant (using the <ExpressionResult> tag), we will extend that so that the <Expression> tag may be used for the <Case> instead of only the <ExpressionResult>. This primarily done so we could access the dictionary variables and use them as constants.

### Property Expressions

Many properties that are considered to be constants in MSFS may be defined as expressions in the Gauge System.

## New Constructs

The following identifies those constructs (tags and objects) that are completely unique to the DRS Gauge implementation.

### Choose/When/Otherwise

The Choose/When construct is modeled after a conditional (if/then/else) construct that XSLT uses. This is used very much like an if/then/else construct. Each <When> owns an expression (“<Expression>”) that is evaluated; if the value is true then the contents of the <When> are used.

You create a “Choose” block by entering the <Choose></Choose> tags, between those tags you create one or more <When></When> blocks.

### ForLoop

The <ForLoop> tag is used to actually process a set of tags and their expressions repetitively. The <ForLoop> most closely resembles C’s “for” loop construct.

for ( initialize; loop-test; increment )

{

Instructions to loop across

}

In similar fashion the <ForLoop> tag supports an <InitialExpression> to setup any initial state or variables, a <TestExpression> tag to be evaluated to determine if we should continue processing, and an <IncrementExpresison> to set us up for the next loop.

*It is important to note that any registers referenced by a ForLoop have the scope of the loop itself. You may not refer to any register that is created outside of the scope of the ForLoop.*

### Line

Surprising as it may seem but MSFS does not support a simple line. I guess technically the Polyline could be used as a line as long as you restrict the input to two points.

# Screen

Simply owns lists of panels and supports the ability to identify the “active” panel. Screen may also own our Dictionary

<Screen>

<PanelInstance Name=”some name” selected=”default” />

<PanelInstance Name=”another panels name” />

<PanelInstance Name=”etc.” />

</Screen>

Please note that the names of the panel-instance does NOT include the XML name suffix.

# Panel

Panels own sub-panels, controls and gauges; they also probably own Button-Map and Box-Map objects.

<Panel id=”somename”>

<GaugeInstance Name=”name” X=”999” Y=”999” />

<ControlInstance Name=”name2” X=”888” Y=”888” />

<PanelDiv Name=”name-of-sub” Display=”show/hidden”>

<Position X=”999” Y=”999” Float=”left/right” Align=”left/center/right” />

<GaugeInstance Name=”xxxxxx” />

<ControlInstance Name=”yyyyy” />

</PanelDiv>

</Panel>

# PanelDiv (div)

# Control

# Gauge

Element

Rotate

Shift

# Graphic

## Arc

Axis = center of arc  
Radius  
LineWidth = width in pixels  
LineColor =  
LineColorScript  
StartAngle (degrees)  
EndAngle (degrees)  
StartAngleScript (optional)  
EndAngleScript (optional)

## Circle

Axis  
Radius  
FillColor  
FillColorScript  
LineWidth  
LineColor  
LineColorScript

## Ellipse

Axis  
Height  
Width  
FillColor  
FillColorScript  
LineWidth  
LineColor  
LineColorScript

## ClipRect

FloatPosition

Size

## HorizontalLine

Axis  
Width  
LineWidth  
LineColor  
LineColorScript

## Pie

Axis

Radius

LineWidth

FillColor

FillColorScript

LineColor

LineColorScript

StartAngle

EndAngle

StartAngleScript

EndAngleScript

## Polygon

Axis

Point (list of points)

FillColor

FillColorScript

LineWidth

LineColor

LineColorScript

## Point

FloatPosition

## Polyline

Axis

Point (list of Points)

LineWidth

LineColor

LineColorScript

## Rectangle

Axis

LineWidth

Width

WidthScript

Height

HeightScript

LineColor

LineColorScript

FillColor

FillColorScript

Transparency

## VerticalLine

Axis

Height

LineWidth

LineColor

LineColorScript

# Dictionary

# Button-Map

# Events

Event-Queue

# Platform Wrappers

### OmniScape

Draw

XML

File

Commands

### Windows

Draw

XML

File

Commands

# Event Processing

This is just some thoughts and not a design document.

Consider that we have basically two flavors of possible stimulus from a “user”; one is by pressing bezel buttons, and the second is by clicking on a sensitive area on the screen (mouse or touch).

Now consider the idea that we really don’t want the Gauges/Controls to have to know what buttons are being pressed. If we introduce the idea of having a “stimulus identifier” (just a simple string), then we can allow a gauge/control to “export” those identifiers with which it is interested. Then if we say that our “Button Map” creates a relationship between physical keys/buttons and the stimulus-id. Something similar to:

<Stimulate key=”F10” id=”Stim-George”/>

To make things easier, I suggest that the “key” attribute allows for “key” names or characters. Such as “A” representing the press of an upper-case letter-A, the function keys would be specified as “F1”, “F2”, “F3” and so on, the arrow keys would be “UP”, “DOWN”, “LEFT”, “RIGHT”, and character codes would be something like “%20” or “%xFA”.

Then within a gauge/control we would have section entitled <Stimulus> which would look something like:

<Stimulus>

<Action for=”Stim-George”>

<Command>Some command identifier</Command>  
<Value>some expression that will result in assigning

a value to a variable</Value>

</Action>

Then the mouse processing section would look something like:

<Mouse>

<Area Left=”68” Right=”86” Top=”71” Bottom=”88”>

<Click>Assign some value or generate a Stimulus-id</Click>

</Area>

</Mouse>

The interesting thing about one item that I implied here is the ability for a mouse input to generate stimulus. This would allow a mouse-click (touch screen) to emulate the operation of pressing a bezel button. The other nice thing about this approach is that stimulus can cause more than one gauge to do things.

Stimulus deals with bubble processing similar to event-bubbling (but not exactly the same).

## Initialization

Now that we have identified some of the base concepts