

Wir basteln uns ein digitales Instrument

Carsten Lynker

September 18, 2012



1 Zusammenfassung

Dieses Dokument beschreibt den Vorgang ein digitales Instrument mit den Tool FlyWithLua zu erschaffen. Das Instrument soll dabei möglichst flexibel sein, um es später einfach in weiteren Projekten einsetzen zu können. Aus diesem Grund werden wir ein Lua-Modul `instrument.lua` erstellen, in dem das digitale Instrument als Funktion hinterlegt ist.

In diesem Beispiel wollen wir ein kombiniertes Instrument für den Ladedruck und die Propellerdrehzahl eines einmotorigen, kolbengetriebenen Kleinflugzeugs erstellen. Hier gibt es die Faustformel, dass der Ladedruck in inHg nicht ein Hundertstel der Drehzahl überschreiten soll. Dreht der Propeller also mit 2.300 U/min, so sollte man nicht mehr als 23 inHg an Ladedruck dem Motor abverlangen.

Unser Instrument soll diese Eselsbrücke visualisieren, um dem Piloten beim Flug zu helfen, die Hebel für Leistung und Drehzahl richtig zu bedienen. Aus diesem Grund werden wir beide Werte, also Ladedruck und Drehzahl, auf einem gemeinsamen Rundinstrument darstellen.

Der Ladedruck wird mit einem weißen Zeiger dargestellt, die Drehzahl mit einem blauen Zeiger. Das ist willkürlich, nutzt aber die Tatsache, dass der Drehzahlregler meistens einen blau gefärbten Griff hat. Der blaue Hebel beeinflusst den blauen Zeiger, der schwarze Hebel den weißen Zeiger.

Überschreitet der Ladedruck den Faustformelwert um mehr als 5%, so färben wir den Zeiger rot statt weiß. Überschreitet der Ladedruck den Faustformelwert um mehr als 10%, so lassen wir ihn zusätzlich blinken.

Zu allem Überfluss werden wir am Schluss das Instrument noch erweitern, um es für Flugzeuge mit zwei Propellern nutzbar zu machen.

2 Der Aufruf

Wir wollten das Instrument in einer Bibliothek oder auch Modul als Funktion hinterlegen. Damit wir aber nicht blind programmieren, beginnen wir mit einer Testdatei, die unser Instrument aufruft. Das kann dann wie folgt aussehen:

```
1 require("instrument.lua")  
2 do_every_draw("instrument.manifold_pressure_and_propeller_speed(100, 300, 25, 2500)")
```

Wir nennen die Datei `Instrumententest.lua` und erstellen sie im Unterordner `Scripts`. Nach einem Reload von Lua wird sie automatisch ausgeführt, und noch meldet sie einen Fehler, da Lua das Modul nicht finden kann.

In Zeile 1 laden wir das Modul `instrument.lua` und in Zeile 2 rufen wir dann unsere noch zu erschaffende Funktion auf. Wir haben sie, ganz im Sinne der internationalen Luftfahrt, in Englischer Sprache benannt als `manifold_pressure_and_propeller_speed()`. Weil sie aber nicht Teil unseres Scripts ist, sondern aus dem Modul `instrument.lua` stammt, müssen wir den Namen des Moduls, gefolgt von einem Punkt, voran stellen.

Die ersten beiden Parameter geben die linke untere Ecke an, mit der das Instrument auf dem Bildschirm positioniert werden soll. Dann folgen die Reichweiten der beiden Werte. Und ja, es folgen beide Reichweiten. So können wir Flugzeuge behandeln, bei denen das Faustformelverhältnis von 1:100 nicht gilt.

Damit sind wir nun fertig mit dem Testscript.

3 Das Instrument

3.1 Die Grundfläche

Wir starten nun die Entwicklung eines Moduls. Daher verlassen wir den Unterordner `Scripts` und wechseln in den Unterordner `lua52`. Hier, in der »Area 52«, legen wir alle Module ab, um sie von den Skripten zu trennen. Würden wir das nicht machen, so könnte Lua sie nicht von den Skripten unterscheiden (beide enden auf `.lua`). Es gäbe Chaos, denn Lua würde versuchen die Module als Scripte auszuführen.

Den Dateinamen haben wir bereits festgelegt durch den Befehl `require()` in Zeile 1 des Testscripts. Wir erzeugen also eine Datei `instrument.lua` im Unterordner `lua52`. Dabei müssen wir die Groß- und Kleinschreibung beachten!

Die Modul-Datei füttern wir nun mit diesem Code:

```

1  -----
2  --  Lua module "instrument.lua" v1.0  --
3  -----
4  module(..., package.seeall);
5  require("graphics")
6
7  function manifold_pressure_and_propeller_speed(x, y, inHg_range, rpm_range)
8      glColor3f(0, 0, 0)
9      graphics.draw_filled_circle(x + 100, y + 100, 100)
10     glColor3f(1, 1, 1)
11     graphics.draw_arc(x + 100, y + 100, -150, 150, 100)
12     draw_string(x + 90, y + 50, "inHg")
13     draw_string(x + 75, y + 30, "x100 rpm", "blue")
14 end

```

Wir können nun zum ersten Mal sinnvoll auf Reload klicken, denn nun hat das Script ein Modul, was Lua laden kann. Das Ergebnis ist dies:



Gehen wir unseren Code einmal durch. Zu Beginn finden wir drei Zeilen mit einem Kommentar, den Lua durch »-- « erkennt, also zwei Minuszeichen gefolgt von einem Leerzeichen. Kommentare können wir nach eigenen Vorlieben schreiben, Lua benötigt sie nicht.

Wichtig hingegen die nächsten beiden Zeilen. Die Zeile 4 muss genau so geschrieben werden, sonst erkennt Lua nicht, dass es sich um ein Modul handelt, und alle Funktionen durch `instrument.*` zur Verfügung gestellt werden sollen.

In Zeile 5 teilen wir Lua mit, dass wir Elemente eines anderen Moduls verwenden wollen, hier `graphics.lua`. Wichtig ist, dass wir die Dateierweiterung weglassen, sonst meckert Lua, es könne die Datei nicht finden.

Ab Zeile 7 folgt nun unsere Funktion. Wir setzen die Farbe direkt mit `glColor3f()`, einem OpenGL Befehl. Der erste Parameter ist der Rotanteil (von 0.0 bis 1.0), dann folgen Grün und Blau. Alle Werte sind 0, dadurch wird alles was nun folgt nicht farblos, sondern schwarz.

In Zeile 8 nutzen wir eine Funktion des Moduls `graphics.lua` um einen Kreis zu zeichnen. Der Kreis liegt auf dem Mittelpunkt `x + 100, y + 100` und hat den Radius 100.

Bedingt durch Zeile 7 ist er schwarz gefüllt. Wir ändern nun die Farbe in Weiß (Zeile 9) und zeichnen einen Kreisbogen (Zeile 10).

Der Kreisbogen reicht vom Winkel -150° bis $+150^\circ$, umspannt also 300° . Im Lua Code lassen wir die Einheiten weg. 0° zeigt genau nach oben. Wir bekommen also eine Öffnung nach unten hin.

Die Kreisbogenöffnung füllen wir mit Text. Da die Funktion `draw_string()` direkt von Fly-WithLua zur Verfügung gestellt wird, und nicht von dem Modul `graphics.lua`, können wir die Farbe nicht durch einen voran gestellten `glColor3f()` Befehl wählen. Wir geben sie, sofern wir nicht Weiß verlangen wollen, als zusätzlichen Parameter an.

Der Eigensinn der Funktion `draw_string()` wird uns noch öfter ärgern, wir nehmen es einfach hin.

3.2 Die Zeiger

Nun soll sich aber endlich etwas bewegen! Wir programmieren die beiden Zeiger hinzu.

Um die Zeiger anzeigen zu können, benötigen wir die Werte des Ladedrucks und der Propellerdrehzahl. Beide Werte liefern uns DataRefs, die wir nun in unser Modul einbauen:

```
7 dataref("xp_MPR_in_hg0", "sim/cockpit2/engine/indicators/MPR_in_hg", "readonly", 0)
8 dataref("xp_prop_speed_rpm0", "sim/cockpit2/engine/indicators/prop_speed_rpm", "
  readonly", 0)
```

Aus diesen DataRefs heraus können wir die Werte entnehmen, doch eigentlich benötigen wir Winkel statt Werten. Wir berechnen die Winkel gemäß der als Funktionsparameter erhaltenen Grenzen. Hier das Beispiel für den Ladedruck:

```
13 local inHg
14 if xp_MPR_in_hg0 < inHg_range then
15     inHg = xp_MPR_in_hg0 / inHg_range * 300 - 150
16 else
17     inHg = 150
18 end
```

Der Wert für den Ladedruck ist erwartungsgemäß nicht negativ. Wir prüfen also nur gegen das obere Ende, da unsere Skala bei 0 beginnt. Da wir 300° umspannen, multiplizieren wir das Verhältnis zum Maximalwert unseres Anzeigebereichs mit 300° und subtrahieren 150° , um den Pfeil in die richtige Richtung zu drehen (die Skala beginnt nicht bei 0° oben, sondern bei -150° schräg unten rechts).

Nun müssen wir den Pfeil nur noch zeichnen:

```

39     glColor3f(1, 1, 1)
40     graphics.draw_angle_arrow(x + 100, y + 100, inHg, 100, 20, 3)

```

Den Pfeil für den Ladedruck zeichnen wir in weiß (Zeile 39) und nutzen eine fertige Funktion um den Pfeil zu erzeugen (Zeile 40). Die Parameter sind die Koordinaten des Mittelpunktes, der Winkel, der Radius, die Größe der Pfeilspitze und die Breite des Pfeils.

Das komplette Script sieht wie folgt aus. Ab Zeile 42 folgt noch etwas Schmuck für die Mitte des Instruments (eine runde Abdeckkappe).

```

1  ---
2  --- Lua module "instrument.lua" v1.0 ---
3  ---
4  module(..., package.seeall);
5  require("graphics")
6
7  dataref("xp_MPR_in_hg0", "sim/cockpit2/engine/indicators/MPR_in_hg", "readonly", 0)
8  dataref("xp_prop_speed_rpm0", "sim/cockpit2/engine/indicators/prop_speed_rpm", "
   readonly", 0)
9
10 function manifold_pressure_and_propeller_speed(x, y, inHg_range, rpm_range)
11     -- calculate the angles for the pointer
12     local rpm
13     local inHg
14     if xp_MPR_in_hg0 < inHg_range then
15         inHg = xp_MPR_in_hg0 / inHg_range * 300 - 150
16     else
17         inHg = 150
18     end
19     if xp_prop_speed_rpm0 < rpm_range then
20         rpm = xp_prop_speed_rpm0 / rpm_range * 300 - 150
21     else
22         rpm = 150
23     end
24
25     -- draw the instruments base
26     glColor3f(0, 0, 0)
27     graphics.draw_filled_circle(x + 100, y + 100, 100)
28     glColor3f(1, 1, 1)
29     graphics.draw_arc(x + 100, y + 100, -150, 150, 100)
30     draw_string(x + 90, y + 50, "inHg")
31     draw_string(x + 75, y + 30, "x100 rpm", "blue")
32
33     -- redefine OpenGL state after drawing strings
34     XPLMSetGraphicsState(0, 0, 0, 0, 0, 0, 0)
35
36     -- draw the pointer
37     glColor3f(0, 0, 1)
38     graphics.draw_angle_arrow(x + 100, y + 100, rpm, 100, 20, 3)
39     glColor3f(1, 1, 1)
40     graphics.draw_angle_arrow(x + 100, y + 100, inHg, 100, 20, 3)
41
42     -- make the middle of the instrument more eye-candy
43     glColor3f(0, 0, 0)
44     graphics.draw_filled_circle(x + 100, y + 100, 7.5)
45     glColor3f(1, 1, 1)
46     graphics.draw_circle(x + 100, y + 100, 7.5)
47 end

```

Es wurde ja schon erwähnt, dass uns der Befehl `draw_string()` ärgern wird. Um dies zu verhindern, sind Zeile 33 und 34 da.

```

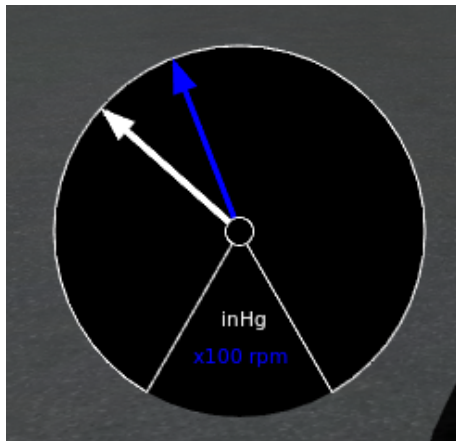
33  — redefine OpenGL state after drawing strings
34  XPLMSetGraphicsState(0, 0, 0, 0, 0, 0, 0)

```

Da die Lua-Funktion `draw_string()` die C-Funktion `XPLMDrawString()` aus dem Plugin SDK aufruft, muss sie damit leben, dass der Zustand von OpenGL verändert wird.

Mit `XPLMSetGraphicsState()` setzen wir den Zustand zurück wie wir ihn benötigen. Wir müssen dies nicht so genau verstehen, es reicht zu wissen, dass man diese Zeile 34 nach jeder Benutzung von `draw_string()` in seinen Code einfügt.

FlyWithLua macht es nicht automatisch, da nicht bekannt ist, welchen OpenGL Zustand genau der Script Autor wünscht. Mit dieser Zeile 34, also alle Parameter 0, haben wir z. B. keine Möglichkeit mit Transparenz zu arbeiten.



3.3 Skalenstriche

Das Instrument sieht ja schon ganz nett aus, und funktioniert anscheinend auch, jedoch kann man mit der Stellung der Pfeile noch wenig anfangen. Gut, man kann beurteilen, ob der weiße Pfeil den blauen Pfeil »illegal überholt«, aber man möchte schon gerne die Werte ablesen können.

Die Bibliothek (das Modul) `graphics.lua` hat hierfür eine passende Funktion parat. Wir können mit `graphics.draw_tick_mark(x, y, angle, radius, length, width)` die Skalenstriche ohne nennenswerten Aufwand erzeugen.

Mittelpunkt und Radius sind uns vorgegeben, Länge und Breite der Skalenstriche unterlegen dem Geschmack, der Winkel muss von uns berechnet werden. Wir erzeugen uns eine lokale Variable `tick_angle`, um diese in einer `for` Schleife zu berechnen.

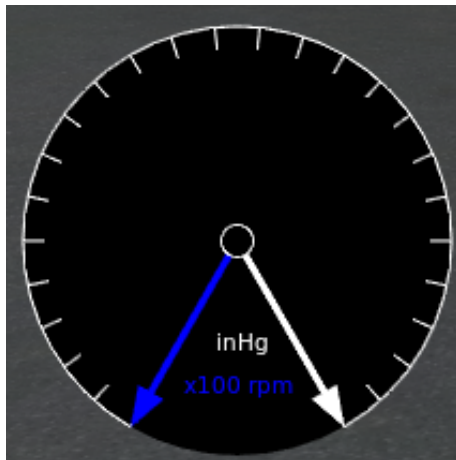
Der neue Code ist:

```

36  — draw tick marks
37  local tick_angle
38  for tick = 100, rpm_range, 100 do
39      tick_angle = tick / rpm_range * 300 - 150
40      graphics.draw_tick_mark(x + 100, y + 100, tick_angle, 100, 10, 1)
41  end

```

Die Berechnung kennen wir bereits von den Pfeilen. Das spannende ist, dass wir uns automatisch den übergebenen Maximalwerten anpassen. Allerdings sieht das Ergebnis noch recht simpel aus.



Wir fügen noch ein paar dickere Striche hinzu und zeichnen in blauer Farbe noch die Werte für die Drehzahl.

Mit verändertem Aufruf, damit wir die blauen Striche sehen können, sieht das nun so aus:

```

2 do_every_draw("instrument.manifold_pressure_and_propeller_speed(100, 300, 25, 2150)")

```




```

36  — draw tick marks
37  local tick_angle
38  — small inHg ticks
39  glColor3f(1, 1, 1)
40  for tick = 1, inHg_range, 1 do
41      tick_angle = tick / inHg_range * 300 - 150
42      graphics.draw_tick_mark(x + 100, y + 100, tick_angle, 100, 10, 1)
43  end
44  — big inHg ticks
45  for tick = 5, inHg_range, 5 do
46      tick_angle = tick / inHg_range * 300 - 150
47      graphics.draw_tick_mark(x + 100, y + 100, tick_angle, 100, 20, 3)
48  end
49  — small rpm ticks
50  glColor3f(0, 0, 1)
51  for tick = 100, rpm_range, 100 do
52      tick_angle = tick / rpm_range * 300 - 150
53      graphics.draw_tick_mark(x + 100, y + 100, tick_angle, 100, 10, 1)
54  end
55  — big rpm ticks
56  for tick = 500, rpm_range, 500 do
57      tick_angle = tick / rpm_range * 300 - 150
58      graphics.draw_tick_mark(x + 100, y + 100, tick_angle, 100, 20, 3)
59  end

```

3.4 Beschriftung der Skala

Die Beschriftung erfolgt wieder mit der (eigenwilligen) Funktion `draw_string`. Das »Problem« ist die richtige Positionierung der Werte. Dazu nehmen wir uns die Hilfsfunktion `move_angle()` aus dem `graphics.lua` Modul zu Hilfe. Sie gibt uns Koordinaten zurück, wenn wir ihr als Parameter den Mittelpunkt, einen Winkel und einen Radius übergeben.

Das Ergebnis ist nun:



```

61  local x1
62  local y1
63
64  — draw numbers to the big inHg tick marks
65  for tick = 5, inHg_range, 5 do
66      tick_angle = tick / inHg_range * 300 - 150
67      x1, y1 = graphics.move_angle(x + 100, y + 100, tick_angle, 65)
68      x1 = x1 - 5
69      y1 = y1 - 5
70      draw_string(x1, y1, tick, "white")
71  end

```

Wir subtrahieren noch 5 Pixel in jede Richtung, um die Nummer einigermaßen mittig unter den Skalenstrichen zu platzieren. Für die Beschriftung der Drehzahl gehen wir genauso vor. Das komplette Script ist nun:

```

1  — — — — —
2  — — Lua module "instrument.lua" v1.0 — — —
3  — — — — —
4  module(..., package.seeall);
5  require("graphics")
6
7  dataref("xp_MPR_in_hg0", "sim/cockpit2/engine/indicators/MPR_in_hg", "readonly", 0)
8  dataref("xp_prop_speed_rpm0", "sim/cockpit2/engine/indicators/prop_speed_rpm", "
   readonly", 0)
9
10 function manifold_pressure_and_propeller_speed(x, y, inHg_range, rpm_range)
11     — calculate the angles for the pointer
12     local rpm
13     local inHg
14     if xp_MPR_in_hg0 < inHg_range then
15         inHg = xp_MPR_in_hg0 / inHg_range * 300 - 150
16     else
17         inHg = 150
18     end
19     if xp_prop_speed_rpm0 < rpm_range then
20         rpm = xp_prop_speed_rpm0 / rpm_range * 300 - 150
21     else
22         rpm = 150
23     end
24
25     — draw the instruments base
26     glColor3f(0, 0, 0)
27     graphics.draw_filled_circle(x + 100, y + 100, 100)
28     glColor3f(1, 1, 1)
29     graphics.draw_arc(x + 100, y + 100, -150, 150, 100)
30     draw_string(x + 90, y + 50, "inHg")
31     draw_string(x + 75, y + 30, "x100 rpm", "blue")
32
33     — redefine OpenGL state after drawing strings
34     XPLMSetGraphicsState(0, 0, 0, 0, 0, 0, 0)
35
36     — draw tick marks
37     local tick_angle
38     — small inHg ticks
39     glColor3f(1, 1, 1)
40     for tick = 1, inHg_range, 1 do
41         tick_angle = tick / inHg_range * 300 - 150
42         graphics.draw_tick_mark(x + 100, y + 100, tick_angle, 100, 10, 1)
43     end

```

```

44 | — big inHg ticks
45 | for tick = 5, inHg_range, 5 do
46 |     tick_angle = tick / inHg_range * 300 - 150
47 |     graphics.draw_tick_mark(x + 100, y + 100, tick_angle, 100, 20, 3)
48 | end
49 | — small rpm ticks
50 | glColor3f(0, 0, 1)
51 | for tick = 100, rpm_range, 100 do
52 |     tick_angle = tick / rpm_range * 300 - 150
53 |     graphics.draw_tick_mark(x + 100, y + 100, tick_angle, 100, 10, 1)
54 | end
55 | — big rpm ticks
56 | for tick = 500, rpm_range, 500 do
57 |     tick_angle = tick / rpm_range * 300 - 150
58 |     graphics.draw_tick_mark(x + 100, y + 100, tick_angle, 100, 20, 3)
59 | end
60 |
61 | local x1
62 | local y1
63 |
64 | — draw numbers to the big inHg tick marks
65 | for tick = 5, inHg_range, 5 do
66 |     tick_angle = tick / inHg_range * 300 - 150
67 |     x1, y1 = graphics.move_angle(x + 100, y + 100, tick_angle, 65)
68 |     x1 = x1 - 5
69 |     y1 = y1 - 5
70 |     draw_string(x1, y1, tick, "white")
71 | end
72 |
73 | — draw numbers to the big rpm tick marks
74 | for tick = 5, rpm_range / 100, 5 do
75 |     tick_angle = tick / rpm_range * 30000 - 150
76 |     x1, y1 = graphics.move_angle(x + 100, y + 100, tick_angle, 65)
77 |     x1 = x1 - 5
78 |     y1 = y1 - 5
79 |     draw_string(x1, y1, tick, "blue")
80 | end
81 |
82 | — redefine OpenGL state after drawing strings
83 | XPLMSetGraphicsState(0, 0, 0, 0, 0, 0, 0, 0)
84 |
85 | — draw the pointer
86 | glColor3f(0, 0, 1)
87 | graphics.draw_angle_arrow(x + 100, y + 100, rpm, 100, 20, 3)
88 | glColor3f(1, 1, 1)
89 | graphics.draw_angle_arrow(x + 100, y + 100, inHg, 100, 20, 3)
90 |
91 | — make the middle of the instrument more eye-candy
92 | glColor3f(0, 0, 0)
93 | graphics.draw_filled_circle(x + 100, y + 100, 7.5)
94 | glColor3f(1, 1, 1)
95 | graphics.draw_circle(x + 100, y + 100, 7.5)
96 | end

```

Bei der Beschriftung der Drehzahl-Skala lassen wir die Werte von 5 an in 5'er Schritten laufen, um das Instrument nicht mit Nullen zuzupflastern. So wie wir es von echten Instrumenten auch gewohnt sind. Allerdings erfordert es etwas andere Mathematik in Zeile 74 und 75. Statt $*300*100$ haben wir zu $*30000$ zusammengefasst, Das entspricht:

```
tick_angle = tick / (rpm_range/100) * 300 - 150
```

3.5 Einfärben des Zeigers

Wir wollten ja, dass der weiße Zeiger (er »liegt oben«) sich rot färbt, wenn er den blauen Zeiger um mehr als 5% der Anzeigereichweite »überholt«.

Die Berechnung kann hier relativ einfach erfolgen, 5% von 300° sind 15°. Wir testen einfach gegen den Winkel, nicht gegen die Werte des Ladedrucks und der Drehzahl. Wie man einen Pfeil färbt wissen wir bereits. Relevant sind die Zeilen 89 bis 91.

```

85  — draw the pointer
86  glColor3f(0, 0, 1)
87  graphics.draw_angle_arrow(x + 100, y + 100, rpm, 100, 20, 3)
88  glColor3f(1, 1, 1)
89  if inHg > rpm + 30 then
90      glColor3f(1, 0, 0)
91  end
92  graphics.draw_angle_arrow(x + 100, y + 100, inHg, 100, 20, 3)

```

Wir stellen den Aufruf wieder zurück und sehen uns das Ergebnis erschreckt an. Die blauen Werte überlagern die weißen, wir wollen es jedoch umgekehrt. Nach einer Umstrukturierung des Scripts ist die Welt wieder in Ordnung. Das folgende Bild zeigt das Instrument bei ausgeschaltetem Motor (Luftdruck um die 30 inHg).

```

2 do_every_draw("instrument.manifold_pressure_and_propeller_speed(100, 300, 25, 2150)")

```



Im Bild auch zu sehen ist das Plugin [DataRefEditor](#), ein äußerst nützlicher Helfer.

Das Blinken des Zeigers erreichen wir, indem wir mit `os.clock()` die Sekunden seit dem Start von Lua abfragen und diesen Fließkommawert mit der Sinusfunktion zu einem hübschen Schwingen des Rotwertes der RGB-Farbe nutzen.

Das schaut dann so aus:

```
85  — draw the pointer
86  glColor3f(0, 0, 1)
87  graphics.draw_angle_arrow(x + 100, y + 100, rpm, 100, 20, 3)
88  glColor3f(1, 1, 1)
89  if inHg > rpm + 15 then
90      glColor3f(1, 0, 0)
91  end
92  if inHg > rpm + 30 then
93      glColor3f(math.abs(math.sin(os.clock()*5)), 0, 0)
94  end
95  graphics.draw_angle_arrow(x + 100, y + 100, inHg, 100, 20, 3)
```

3.6 Feintuning am Aufrufer

Nachdem wir es geschafft haben, können wir den Aufrufer noch etwas verbessern. Wir wollen das Instrument immer oben rechts anzeigen, aber nur, wenn wir uns außerhalb des Cockpits befinden.

Dazu nutzen wir ein weiteres DataRef.

```
1  require("instrument")
2
3  dataref("xp_view_is_external", "sim/graphics/view/view_is_external")
4
5  function show_a_little_instrument()
6      if xp_view_is_external > 0 then
7          instrument.manifold_pressure_and_propeller_speed(SCREEN_WIDTH - 210,
8              SCREEN_HIGHT - 220, 25, 2500)
9      end
10  end
11  do_every_draw("show_a_little_instrument()")
```

Damit es nicht ganz so am Rand »klebt« gönnen wir dem Instrument noch 20 Pixel Anstand zum oberen Rand und 10 Pixel zur Seite.

4 Anhang

Zum Schluss noch das komplette Script:

```

1  ---
2  --- Lua module "instrument.lua" v1.0 ---
3  ---
4  module(..., package.seeall);
5  require("graphics")
6
7  dataref("xp_MPR_in_hg0", "sim/cockpit2/engine/indicators/MPR_in_hg", "readonly", 0)
8  dataref("xp_prop_speed_rpm0", "sim/cockpit2/engine/indicators/prop_speed_rpm", "
    readonly", 0)
9
10 function manifold_pressure_and_propeller_speed(x, y, inHg_range, rpm_range)
11     -- calculate the angles for the pointer
12     local rpm
13     local inHg
14     if xp_MPR_in_hg0 < inHg_range then
15         inHg = xp_MPR_in_hg0 / inHg_range * 300 - 150
16     else
17         inHg = 150
18     end
19     if xp_prop_speed_rpm0 < rpm_range then
20         rpm = xp_prop_speed_rpm0 / rpm_range * 300 - 150
21     else
22         rpm = 150
23     end
24
25     -- draw the instruments base
26     glColor3f(0, 0, 0)
27     graphics.draw_filled_circle(x + 100, y + 100, 100)
28     glColor3f(1, 1, 1)
29     graphics.draw_arc(x + 100, y + 100, -150, 150, 100)
30     draw_string(x + 90, y + 50, "inHg")
31     draw_string(x + 75, y + 30, "x100 rpm", "blue")
32
33     -- redefine OpenGL state after drawing strings
34     XPLMSetGraphicsState(0, 0, 0, 0, 0, 0, 0)
35
36     -- draw tick marks
37     local tick_angle
38     -- small rpm ticks
39     glColor3f(0, 0, 1)
40     for tick = 100, rpm_range, 100 do
41         tick_angle = tick / rpm_range * 300 - 150
42         graphics.draw_tick_mark(x + 100, y + 100, tick_angle, 100, 10, 1)
43     end
44     -- big rpm ticks
45     for tick = 500, rpm_range, 500 do
46         tick_angle = tick / rpm_range * 300 - 150
47         graphics.draw_tick_mark(x + 100, y + 100, tick_angle, 100, 20, 3)
48     end
49     -- small inHg ticks
50     glColor3f(1, 1, 1)
51     for tick = 1, inHg_range, 1 do
52         tick_angle = tick / inHg_range * 300 - 150
53         graphics.draw_tick_mark(x + 100, y + 100, tick_angle, 100, 10, 1)
54     end
55     -- big inHg ticks
56     for tick = 5, inHg_range, 5 do

```

```

57     tick_angle = tick / inHg_range * 300 - 150
58     graphics.draw_tick_mark(x + 100, y + 100, tick_angle, 100, 20, 3)
59 end
60
61 local x1
62 local y1
63
64 — draw numbers to the big rpm tick marks
65 for tick = 5, rpm_range / 100, 5 do
66     tick_angle = tick / rpm_range * 30000 - 150
67     x1, y1 = graphics.move_angle(x + 100, y + 100, tick_angle, 65)
68     x1 = x1 - 5
69     y1 = y1 - 5
70     draw_string(x1, y1, tick, "blue")
71 end
72
73 — draw numbers to the big inHg tick marks
74 for tick = 5, inHg_range, 5 do
75     tick_angle = tick / inHg_range * 300 - 150
76     x1, y1 = graphics.move_angle(x + 100, y + 100, tick_angle, 65)
77     x1 = x1 - 5
78     y1 = y1 - 5
79     draw_string(x1, y1, tick, "white")
80 end
81
82 — redefine OpenGL state after drawing strings
83 XPLMSetGraphicsState(0, 0, 0, 0, 0, 0, 0)
84
85 — draw the pointer
86 glColor3f(0, 0, 1)
87 graphics.draw_angle_arrow(x + 100, y + 100, rpm, 100, 20, 3)
88 glColor3f(1, 1, 1)
89 if inHg > rpm + 15 then
90     glColor3f(1, 0, 0)
91 end
92 if inHg > rpm + 30 then
93     glColor3f(math.abs(math.sin(os.clock()*5)), 0, 0)
94 end
95 graphics.draw_angle_arrow(x + 100, y + 100, inHg, 100, 20, 3)
96
97 — make the middle of the instrument more eye-candy
98 glColor3f(0, 0, 0)
99 graphics.draw_filled_circle(x + 100, y + 100, 7.5)
100 glColor3f(1, 1, 1)
101 graphics.draw_circle(x + 100, y + 100, 7.5)
102 end

```