Pilot^2

Flight simulation on Palm OS

Andrew S. Downs
andrew@downs.ws

Abstract

Flight simulation on a Palm device requires attention to the purpose of such a program, followed by selective inclusion of essential features. Performance considerations encourage some calculations ahead of time. Graphics also pose a challenge due to bitmap constraints in older OS versions and lack of polygon support.

Introduction

In this paper I discuss the design and implementation of a special-purpose flight simulator for Palm OS. It does not depict all aspects of flight, but rather is meant to help aspiring pilots learn to balance the relationships between flight and engine instruments and controls.

One reasonable question is: "Why would you attempt this at all, on such a small device?" I just wanted something fun to play wherever I happened to take my Palm. There is not any interactive flight software available for Palm OS that falls under the "fun" category, although some pilot companion programs exist for flight planning and logging.

Program structure

The application goes through a very simple cycle each second: first, the flight characteristics (control data) is calculated. Then the display is updated.

This application is written in C, although C++ could easily work as well. Java? Maybe, it depends on the current state of the available Virtual Machine(s), in particular how quickly they draw to the screen.

Modeling aircraft performance

This program gives up a certain amount of realism in return for better performance on the Palm. Real aircraft are designed by a cadre of qualified engineers. Every aspect of the aircraft is thoroughly wrung-out before production begins. Since I've only flown aircraft, not designed them, complicated formulas are out of the question.

The approach I use is to create a pseudo "power curve" over a reasonable range of input values. The values are stored in a set of static arrays and can be quickly accessed rather than calculated.

Here is an example: to determine whether the aircraft is climbing, descending, or staying level, you can simply check the difference between the amount of horsepower being produced at a given throttle (power) setting and the amount required to remain level at the current airspeed:

```
excess power = available - required
```

If the excess is positive, the aircraft will climb, and if negative, the aircraft will descend. If zero, the aircraft remains level.

These values can be represented in structures:

```
struct PowerData {
      int rpm; // Power setting.
      int bhp; // Horsepower available.
} ;

struct PowerCurveData {
      int kias; // Current airpseed.
      int bhpReqd; // Horsepower req'd
                   // to remain level.
} ;
```

If the aircraft is turning (banked), less lift is produced and the amount of power required to maintain level flight increases:

```
struct BankData {
      int angle; // Angle of bank.
      int bhpReqd; // Horsepower req'd
                   // to remain level.
      float multiplier; // Used in
                   // turn rate calc.
} ;

int CalcPowerCurve( void ) {
  int retval = 0;
  int i = 0, j = 0;

  // Airpseed determines how much brake
  //   horsepower is required to stay
  //   aloft.
  for ( i = 0; i < 17; i++ ) {
    if ( gFlightData.airspeed <=
      gPowerCurveData[ i ].kias ) {
      break;
    }
  }

  // Power setting determines how much
  //   brake horsepower is available.
  for ( j = 0; j < 18; j++ ) {
    if ( gFlightData.rpm <=
      gPowerData[ j ].rpm ) {
      break;
    }
  }

  // Any bank adds to required power.
  int bank = gFlightData.bank;

  if ( bank < 0 )
    bank = -bank;

  // Return the difference between the
  //   amount available and the amount
  //   required. Use a rough estimate
  //   for the bank angle effect.
  if ( i < 17 && j < 18 )
    retval = gPowerData[ j ].bhp -
      ( gPowerCurveData[ i ].bhpReqd +
      ( bank / 2 ) );

  return retval;
}
```

Since the determination of the current performance is handled in two functions, more complicated but accurate calculations can easily be substituted at a later time.

Palm OS issues

Obviously, display size is a serious issue. This was true even for the non-HUD original version. I briefly prototyped standard analog instrument faces, but it was impossible to squeeze the necessary dials onto one screen and still have the values readable.

Processor speed is another issue. The current crop of Palm devices run in the 16-20 MHz range, which is more than adequate for the relatively few calculations that occur each second.

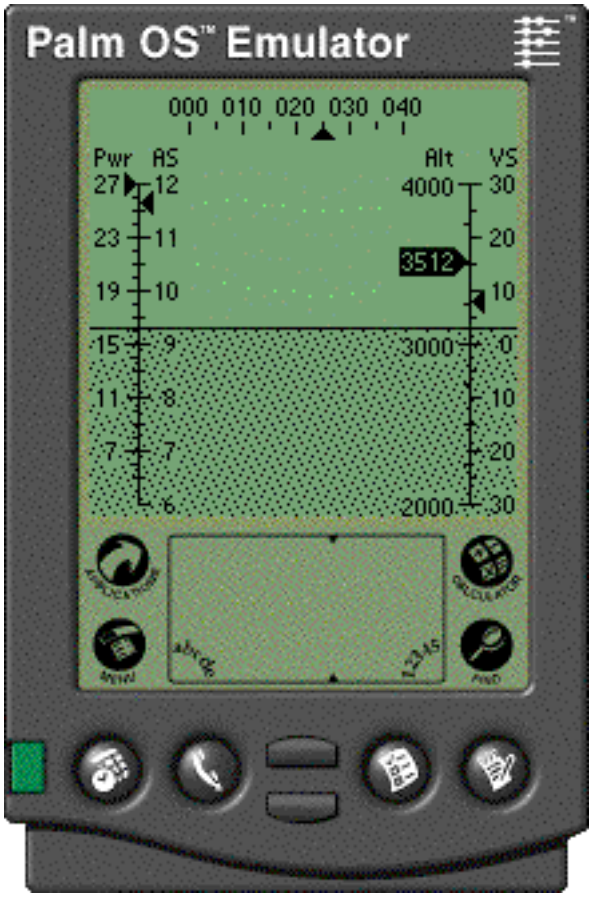

Figure 1. Integrated display view. Aircraft is in a wings-level climb.

User input

Originally this program used image buttons depicting up and down arrows to handle increasing and decreasing input values. The result was functional, but boring.

One suggestion was to use the Palm device hardware buttons to handle user input, a la the SubHunt game that ships with the Palm. This works much better than tapping an onscreen button. There are just enough buttons (ignoring the sleep button) for this scheme to work on existing Palm devices. The left application buttons (`keyBitHard1` and `keyBitHard2` in the API) control roll, the right two buttons (`keyBitHard3/4`) power or throttle, and the up-down scroll arrows (`keyBitPageUp/Down`) control pitch.

Here is an example, closely following the SubHunt sample code (included with CodeWarrior for Palm OS):

```
static void AppEventLoop( void ) {
  EventType event;
  DWord keyState;

  // Mask the game action keys:
  //   here, the four hardware buttons.
  DWord keysAllowedMask =
    keyBitHard1 | keyBitHard2 |
    keyBitHard3 | keyBitHard4 |
    keyBitPageUp | keyBitPageDown;

  do {
    EvtGetEvent( &event,
      gEventTimeout );

    if ( event.eType == keyDownEvent )
    {
      // Get current key.
      keyState = KeyCurrentState() &
        keysAllowedMask;

      // Check against our action keys.
      // Also check boundary values
      //   before processing.
      if ( ( keyState & keyBitHard4 )
        && gFlightData.rpm <
          gFlightDataLimits.rpmMax ) {

        // Do useful stuff here.
        gFlightData.rpm +=
          gRpmIncrement;
      }
    }

    // Eat the event if we've handled
    //   it.
    // Either copy the code directly
    //   from SubHunt, or insert your
    //   own "event-handled" boolean
    //   check.

    // Then, insert standard Palm event
    //  loop code here.

  } while ( event.eType !=
      appStopEvent );
}
```

Data display

The panel on this program began life as a series of labels and text fields. Values for the various instruments were updated each second and the value displayed in the appropriate text field. This was great for debugging the flight dynamics. The one luxury that I provided was a graphical "Attitude Indicator", which depicts the pitch angle (relationship between the aircraft nose and horizon) and also the degree of bank.

Most flight simulators have enough screen real estate to work with so that the instrument display sits comfortably below the "outside world" display, which usually depicts the horizon as a visual reference.

This issue in particular started me thinking about better ways to present the same information in a smaller space. The HUD (head-up display) found on modern vehicles (aircraft, automobiles, etc.) overlays important information onto the view out the front window.

The HUD representation used in this implementation is a bit crowded, and some of the graphic (analog) items could be downgraded to simple numeric fields. But one advantage of analog gauges is they allow your brain to register trends and relationships easier than if you were viewing only discrete values.

For example, I may make a mental note that the power and airpseed markers maintain a particular constant offset from each other once the proper pitch angle for a full-power climb is attained. This "managing of relationships" allows an experienced pilot to quickly determine whether the aircraft is in a stable or unstable state (which is particularly important when you venture into clouds and cannot see outside!)

Many of the instruments work over a fixed, relatively small range of values. These scales are drawn in the display and the current value pointer overlaid at the appropriate location. The power setting (RPM) and Vertical Speed fall into this category.

Altitude, however, requires more detail because it is a critical component of most flight maneuvers, including straight-and-level flight. I implemented a "moving tape" display that brackets the current altitude between two boundary values (which are separated by 2000 feet). To provide more detail, the current altitude is drawn as text in an inverted rectangle, and a pointer to the tape value is appended to the side of this inverted rect.

In the current implementation, the minimum altitude is 0 and the maximum is 14,400. This was an arbitrary decision, particularly the minimum, which does not allow for below sea level values.

Figure 2 illustrates the altitude display. The column labeled "Alt" has, at the top and bottom of the column, the current bracketing values, and in the center is an intermediate reference value. The current altitude is in the black block near the bottom of the screen.

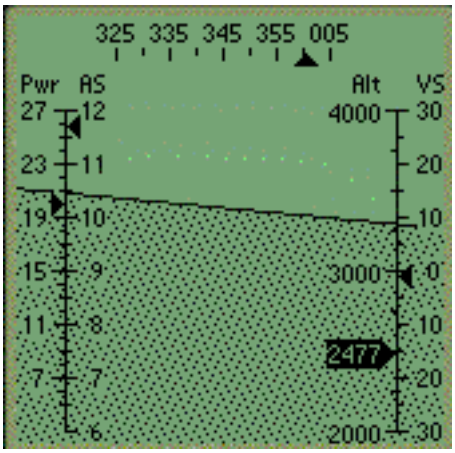

Figure 2. Current altitude is 2,477 feet. Aircraft is slowly descending in a left bank.

Here is how of some of the altitude display is handled. There are three scale values displayed at any time. The corresponding variables are named `gAltTop`, `gAltCenter`, and `gAltBottom`. When the current altitude gets within 100 feet of either the top or bottom scale value, the scale adjusts by 100 feet in the appropriate direction. The position of the current altitude readout then adjusts in relation to the top of the tape. This all happens before the display is drawn.

The variable `gFlightData` refers to a structure containing the current aircraft data.

```
if ( gAltTop - gFlightData.altitude <
     100 && gFlightData.verticalSpeed
     > 0 )
{
  gAltTop += 100;
  gAltCenter += 100;
  gAltBottom += 100;
}
else if ( gFlightData.altitude -
     gAltBottom < 100 &&
     gFlightData.verticalSpeed < 0 )
{
  gAltTop -= 100;
  gAltCenter -= 100;
  gAltBottom -= 100;
```

```
}

// Top of display rect is offset from
//   top of tape.

top = 30;

top += ( ( gAltTop -
  gFlightData.altitude ) / 1000 )
  * 60;

// Adjust display rectangle values to
// account for width of drawn string.

// Fill rectangle, then draw chars.
RctSetRectangle( &r, left, top,
  width, height );

WinInvertRectangle( &r, 0 );

StrPrintF( altBuf, "%d",
  ( int )gFlightData.altitude );

WinDrawInvertedChars( altBuf,
  StrLen( altBuf ), left + 1, top );

// Fetch and draw pointer bitmap.
bitmapH = DmGet1Resource( 'Tbmp',
  RightArrowBitmap );
bitmapP = ( BitmapPtr )
  MemHandleLock( bitmapH );

WinDrawBitmap( bitmapP, left - 5,
  top - 4 );

MemHandleUnlock( bitmapH );
DmReleaseResource( bitmapH );
```

I used the sliding scale approach for the heading as well (at the top of the screen), since that range of values is 0-360 degrees and accuracy is important to a pilot. The rollover of the min and max values uses a smaller threshold than the altitude adjustment (5 vs. 100), but the concept is the same.

Bitmaps

Palm OS provides improved bitmap and graphics support, including color, in OS 3.5 and later. However, to retain compatibility with older OS versions and devices, I used 3.0-compatible calls and monochrome graphics.

One problem was the overlay of multiple bitmaps on the screen. Since bitmaps can only be rectangular in shape, having separate terrain and instrument bitmaps did not work. The instrument display consists primarily of small sections of hash marks and numbers spread out over the surface of the screen. The terrain is (in the initial version of the app) simply a big rectangle containing a pattern. (The sky in the display consists of empty pixels.)

To solve the problem, I elected to use only the terrain bitmap, clipping it as necessary, and to draw the instrument panel dynamically. Some items on the panel are bitmaps themselves.

Also note that there is no polygon support in Palm OS at this time. This was a problem when drawing the pointers for the various instruments, such as the altitude display block. It would be nice to define a polygon and fill/draw within that area. Instead I filled a rectangle, then painted a bitmap (for the triangular pointer) at the correct location beside the rectangle. The bitmap was an irregular shape, and the transparency setting applied via the Constructor resource editor did not seem to work, so I juggled the order in which drawing occurred in order to not overwrite useful data with empty background pixels.

The following code fragment illustrates the creation and filling of the offscreen terrain window at application launch time.

```
static void MainFormInit(FormPtr frmP)
{
  VoidHand bitmapH = 0;
  BitmapPtr bitmapP = 0;
  WinHandle tempWin = 0;

  // Load terrain background.
  bitmapH = DmGet1Resource( 'Tbmp',
    TerrainBitmap );

  bitmapP = ( BitmapPtr )MemHandleLock(
```

```
    bitmapH );

  if ( bitmapP != NULL ) {
    gOffscreenTerrainWindow =
      WinCreateOffscreenWindow( 160,
        160, genericFormat, &err );

    if ( gOffscreenTerrainWindow !=
      NULL ) {
      tempWin = WinSetDrawWindow(
        gOffscreenTerrainWindow );

      WinDrawBitmap( bitmapP, 0, 0 );

      MemHandleUnlock( bitmapH );
      DmReleaseResource( bitmapH );
    }
  }

  if ( tempWin )
    WinSetDrawWindow( tempWin );
}
```

When the aircraft is banked, we have a problem. The terrain can no longer remain "flat"; it must appear at an angle to give the illusion of turning. Since there are no polygons available to hold a triangle to overlay above the terrain, I went with a brute force approach:

1. I drew pattern-filled bitmaps for each possible angle of bank using CodeWarrior Constructor. These bitmaps are only tall enough to hold a triangle drawn with an appropriate hypotenuse in relation to the horizon. The width of each bitmap fills the display (160 pixels). The bitmaps are compiled into the application.

2. During the display updating at runtime, the angle of bank determines which horizon bitmap is retrieved.

3. Determine the appropriate y-coordinate at which to draw the horizon based on the aircraft pitch angle and bitmap height.

4. Draw the terrain bitmap in the lower portion of the display.

5. Draw the horizon bitmap "above" the terrain.

6. Draw the instrument scales, pointers and values.

7. Transfer the offscreen image to the onscreen window.

Figure 3 illustrates the overlay process.

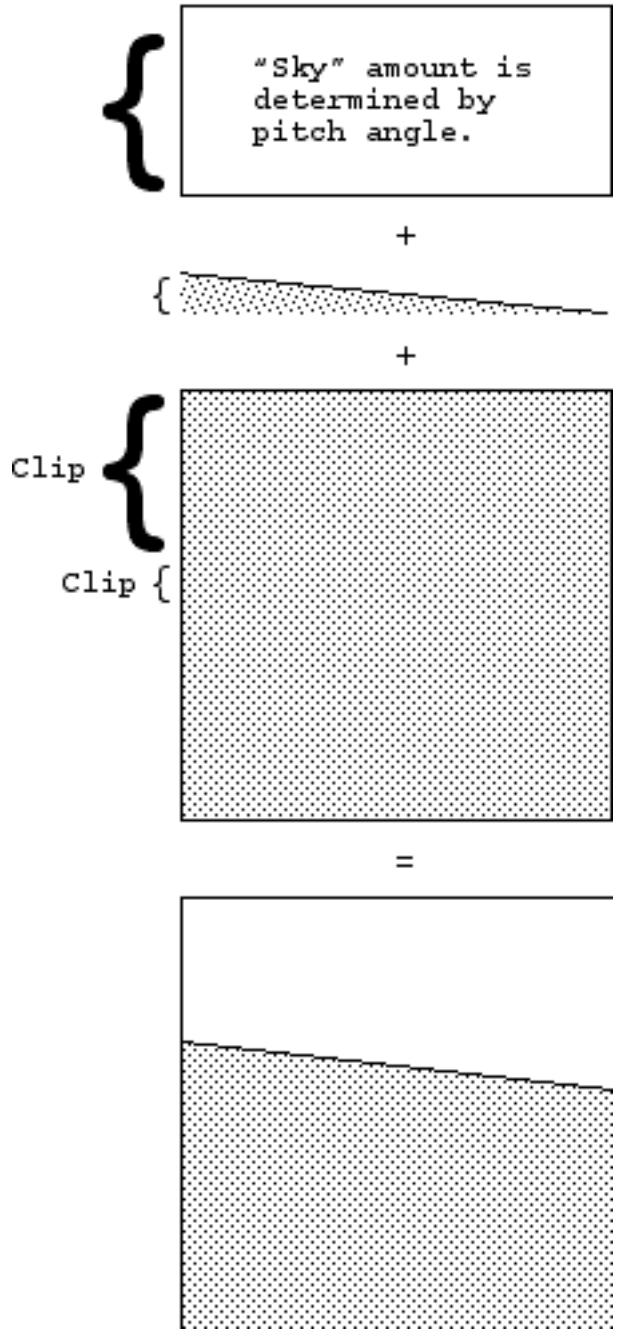

Figure 3. Bitmap overlays.

During a display update, all drawing is first performed to an offscreen window. Once the terrain and panel have been drawn offscreen, that window is transferred onscreen. The initial version does not use clipping rectangles to reduce the amount of drawn data during the display phase, since with a one-second update cycle there appears to be plenty of time for all calculations and drawing to occur. I will run some tests to determine whether this is blind luck or not. As more features get added, it is important to know whether or not they can be comfortably handled during the existing update cycle.

Sound

The program supports a simple engine sound. The sound frequency changes when the engine power setting passes through threshold values at 2000 and 2500 RPM.

A prefs dialog within the app allows the user to turn this item on or off, even if the system and game sound prefs (as defined in the Prefs app in Launcher) are disabled.

In the app's main event handler:

```
if ( gUserPrefs.soundOn ) {
   SndCommandType s;

   // Sound setup.
   s.cmd = sndCmdFrqOn; // Sound on.
   s.param2 = 2000; // Duration in ms.

   if ( gFlightData.rpm < 2000 ) {
     s.param1 = 20; // Frequency.
     s.param3 = 1; // Amplitude.
   }
   else if ( gFlightData.rpm < 2500 )
   {
     s.param1 = 30;
     s.param3 = 2;
   }
   else {
     s.param1 = 40;
     s.param3 = 3;
   }

   // Now play the sound.
   SndDoCmd( NULL, &s, 0 );
}
```

When the user exits, remember to turn off the sound:

```
SndCommandType s;
s.cmd = sndCmdQuiet; // Sound off.
s.param1 = 0;
s.param2 = 0;
s.param3 = 0;
SndDoCmd( NULL, &s, 0 );
```

Conclusion

In spite of the challenges imposed by the platform, the display of information in a HUD is possible, and is certainly more user friendly than a text-based approach. The same techniques used on other platforms for drawing and animation work on Palm OS, although you may have to do more of the work yourself.

Bibliography

[Cessna80] Cessna Aircraft Company. Information Manual, 1980 Model 172N.

[Horne00] Horne, Thomas A. "Future Flight: Cockpit Cinerama". AOPA Pilot. September 2000, Volume 43, Number 9.

[Kershner90] Kershner, William K. The Instrument Flight Manual. Iowa State University Press. Ames, IA. 1990.

[McCornack95] McCornack, Jamie et al. Tricks of the Mac Game Programming Gurus. Hayden Books, Indianapolis, IN. 1995.

[Sollman94] Sollman, Henry and Sherwood Harris. Mastering Instrument Flying. TAB Books, New York, NY. 1994.