TP008: Overlays - 2D Functionality

Introduction

For many real-life applications, a simple 3D view is usually not sufficient. Some form of extra information drawn over the view is required, which can be as simple as a few lines (such as the axes display in RealiView or the STE), or as complicated as a full-blown cockpit, with cockpit instrumentation and head-up display (HUD).

The RealiMation API provides a number of functions which allows your application to show this sort of functionality. This document will describe the functions available to a developer, giving examples of their use. A basic familiarity with RealiMation is assumed.

The 2D functions can be divided into a number of general types, each addressing a different area of funtionality :

- Simple 2D drawing functions
- 2D Image function
- Screen space polygons

A developer can select any mixture of functions in order to best achieve the desired result. We will describe each type in further detail, giving examples of their use.

2D Drawing Functions Work In Channel Space

All RealiMation 2D functions take channel coordinates (these normally correspond directly to pixel coordinates) - they can be used independently of the number and size of any views attached to the channel.

In order to reduce processing requirements, and hence speed up display, coordinates passed into these 2D functions are not checked for validity against the channel size. It is the developer's responsibility to ensure that invalid coordinates are not passed into the functions.

Simple 2D Drawing Functions

These routines provide simple 2D drawing commands, producing lines, circles, filled rectangles and polygons :

```
RT2DCircle   (rtID id, DWord rad,         const rtPixel *pix);
RT2DLine     (rtID id, const rtPixel *p1, const rtPixel *p2);
RT2DRectangle(rtID id, const rtPixel *p1, const rtPixel *p2);
RT2DPolygon  (rtID id, DWord n,           const rtPixel *pix);
```

The drawing colour is defined by the function `RT2DSetColours()`. For the above functions, only the foreground colour is used. However, you can also draw text on the channel, using the `RT2DText()` function. By using the `RT2DSetBackgroundMode()` function, you can decide whether to draw a filled rectangle round the text (in the background colour set by `RT2DSetColours()`), or have the background showing through.

2D Image Function

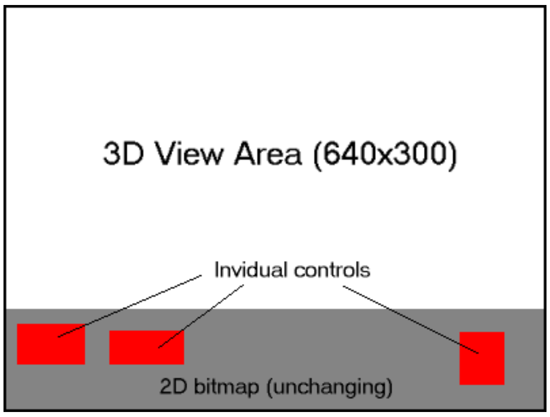
```
RT2DBitmap (rtID id, const rtPixel *pix,
            const rtBitmap *bitmap,
            DWord res0, void *res1);
```

This function allows the application writer to effectively 'Blit' a rectangular area of memory onto the channel. This area is copied directly onto the frame buffer, so the image needs to be in the correct pixel format (i.e. correct bit-depth and RGB masks) for the display channel. To help this, RealiMation provides a number of functions for querying the correct pixel format from the channel (`RTInqBestBitmap()`), and utilities to help you change and scale the image (`RFScaleBitmap(), RFChangeBitmapDepth()`).

Since the image is copied directly to the frame buffer, it avoids any contact with the texture units and texture memory on the graphics card (if present). This means you will get exactly what is in the image, with no additional filtering. It also means that texture memory on the card does not get used up

However, this does mean that the entire image must be transferred to the frame buffer every time it is needed to be drawn. On some hardware, this transfer can take a significant amount of time (depending on the exact hardware and size of image). `RT2DBitmap()` is really designed either for large images which get drawn only a small number of times (such as the sides of cockpits, or splash panels showing instructions), or for small areas, such as individual controls in a cockpit display.

As an example, consider the following scenario. The channel is an area 640x480 pixels. Along the bottom is a strip 180 pixels high, showing various controls against a background representing a cockpit. The rest of the screen being a 3D view on the outside world. The majority of the bottom strip does not change, but small areas of it (representing individual controls) do:



First, we set the size of the 3D view to be 640x300 :

```
RTSetViewPortSize      (view_id, 640, 300);
RTSetViewPortPosition  (view_id, 0, 0);
```

This limits the view to the top section of the screen.

Next, outside the main loop, we use the `RT2DBitmap()` function to draw an image representing the unchanging part of the display.

The `pixel_image` is an `rtBitmap` structure, containing a pointer to the image information. This image can either be constructed 'by-hand' (using `RFConstructBitmap()`), or more likely, loaded using the `RFLoadBitmap()` functions, and then changed to the correct bit-depth for the display.

```
rtPixel  bitmap_position;
bitmap_position.x  =  0;
bitmap_position.y  =  300;
RT2DBitmap  (channel_id,  &bitmap_position,  &pixel_image,  0,  NULL);
```

One point to note, on display drivers capable of double-buffering (which includes most), this only writes the image into one of the buffers. If you leave the code like this, then every time you call `RTSwapPage()`, the 2D bitmap will appear and disappear, seeming to flicker. You also need to draw the same image on the other buffer. If you call `RTSwapPage()`, followed by another call to `RT2DBitmap()`, with the same parameters, then the image will be present in both buffers.

Finally, during the main program loop, you can call `RT2DBitmap()` to update the small areas covered by the individual controls. By keeping these images small, you can reduce the amount of overhead required to update the controls.

In release v4.3.5, new functionality was added to allow more flexibility in this type of 2D overlay. The screen space polygons are the result of this, and can easily be used instead of `RT2DBitmap()` in the above example to display the individual controls.

Screen Space Polygons

```
RT2DScreenPolygon (rtID channel_id, rtScreenPolygon *poly,
                   rtID material,   DWord flags);
```

The 2D screen space polygon is perhaps the most powerful of all the 2D functions provided by RealiMation. It allows the user to send 2D, optionally textured, polygons (specified in channel coordinates) direct to the rendering engine. This means they can take advantage of any texture options (such as bilinear filtering, alpha-channels, etc.) available for textures on ‘normal’ 3D polygons, along with any hardware acceleration. In effect, this provides an ‘immediate mode’, allowing the developer direct rendering of non z-buffered polygons.

The rtScreenPolygon structure provides the basic coordinates for the polygon on screen. Please note, these coordinates must be valid for the channel - in order not to add extra processing overhead, no check is made for valid coordinates. The use of invalid coordinates is not defined - many display drivers do protect themselves, but this is not guaranteed and may lead to a crash.

In addition, the rtScreenPolygon structure also contains a list of texture coordinates which are used if a material with a texture image is applied to the polygon.

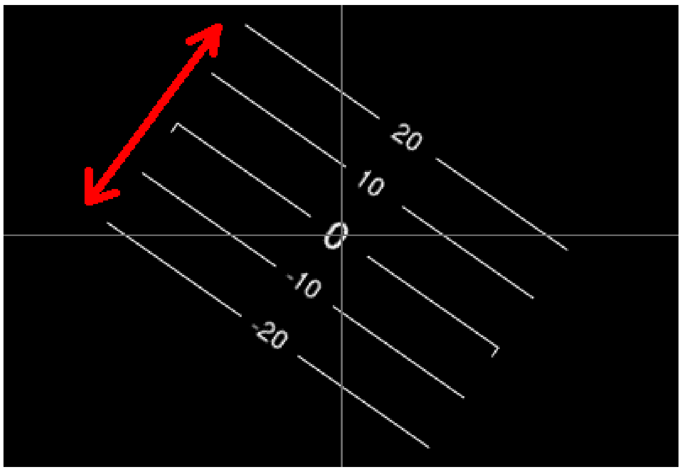
The `material` id parameter points to a standard RealiMation material, as would be used by any 3D polygon. However, since 2D screen space polygons are not lit, the reflectance values are not used. The colour, transparency and image values are used, allowing the polygon to blend in with the 3D view beneath. An example of this would be simulating a sheet of glass over an area (or all) of the channel.

The use of a texture image with an alpha-channel can allow ‘cut-outs’ to be placed in the overlays. This can produce overlays such as window frames or targeting sights, where the underlying 3D view can be seen through gaps in the 2D overlay.

Instead of using the colour definition from the material, it is possible to specify individual colours at each vertex of the polygon. This can generate 2D polygons with colours blending across them. Simply use the RTScreenPolyUseColours flag in the last parameter, and make sure the colours array in the rtPolygon structure is filled in. This colour information will override the material colour, but the other properties of the material (e.g. transparency, images) will still be used.

To show one use of screen space polygons, take the example of a flight simulator application. Over the top of the 3D view of the world, we want to place a head-up display, giving information on the aircraft attitude.

The HUD consists of a series of horizontal, parallel lines, and can rotate with the aircraft’s roll, and move up/down with the aircraft’s pitch, within certain limits. The HUD is designed to be overlaid on top of a 3D view, so the user can clearly see the view underneath :



We create an image showing the horizontal bars and numbers in a separate paint package, complete with alpha-channel which has the spaces between the bars at 100% alpha.

Coordinates for the polygon corners are based on a square, centred on the middle of the view `(x_centre, y_centre)`, and then rotated about that centre by the roll angle of the aircraft. So, the coordinates for the top left corner of the polygon would be :

```
x = -RMCos(roll_angle) * hud_width / 2 -
       RMSin(roll_angle) * hud_height / 2 + x_centre;
y = -RMSin(roll_angle) * hud_width / 2 +
       RMCos(roll_angle) * hud_height / 2 + y_centre;
```

This takes care of the roll angle, all we need to do is make the display move up and down with the pitch angle of the aircraft. We can do this by altering the texture coordinates of the polygon, so that only a small section of the bitmap is displayed on the screen polygon.

So, with 0 degrees pitch, we could set the texture coordinates as

```
top left (0, 0.25)          top right (1, 0.25)
bottom_left (0, 0.75) bottom right (1, 0.75)
```

and we only get a horizontal band across the centre of the bitmap shown. By altering the y part of the texture coordinates in relation to the pitch angle, we can get the HUD to move up/ down :

```
// Calculate the amount of texture offset -
//
//     'texels_per_degree' is a floating point constant
//      which relates to the amount of spacing between the
//      bars in the image, and will depend on the image used
//
//      As an example, if we have an image which is 256 pixels
//      high, in which 4 pixels represents 1 degree of pitch,
//      then
//
// Ord texels_per_degree = 4.0F / 256.0F;
//
//      ie, what fraction of the height of the image does
//      1 degree cover.

Ord texture_offset = RMRadToDeg(pitch_angle) * texels_per_degree;

// Set up the texture coordinates for the default position
// (ie, 0 degrees pitch)
RMV2Set (screen_poly.tex_pts[0], 0.0F,  0.25F);
RMV2Set (screen_poly.tex_pts[1], 1.0F,  0.25F);
RMV2Set (screen_poly.tex_pts[2], 1.0F,  0.75F);
RMV2Set (screen_poly.tex_pts[3], 0.0F,  0.75F);
```

```
        // Now move the coordinates to give the correct offset
        for (int i = 0; i < 4; i++)
        {
            screen_poly.tex_pts[i].y += tex_offset;
        }
```

Since the polygon is being drawn by the rendering engine (rather than direct to the frame buffer like `RT2DBitmap()`), there are a number of considerations which can affect the end result if you are using a textured screen space polygon. These may be desired effects or not, depending on the exact circumstances :

- The image will be downloaded onto the texture memory used by the graphics card. This means that you don't have to repeatedly transfer the image to the graphics card every time you need to draw it. This, combined with the fact that you can make use of hardware acceleration, makes the screen space polygon much faster than `RT2DBitmap()`.
- However, many graphics cards can place restrictions on the sizes of images they can use for textures. Many limit the maximum size, and require that each dimension of the image must be a power of two (e.g. 64, 128, 256, etc.). If the image used for a texture does not satisfy these conditions, then `RT2DScreenPolygon()` will still work, but the image may be resized internally. This may result in extra filtering being applied to the image, resulting in potentially unexpected blurring of the image. Details of limitations imposed by individual drivers can be found using the `RTInqDriver()` function.

Differences in sizes between the pixel coordinates in the `rtScreenPolygon` structure, and the size of the image, can result in bilinear filtering or mip-mapping being applied to the image (if the relevant image flags are set up). As an example, if the image is 64x64, and `RT2DScreenPolygon()` is used to draw a square region of 128x128 (with texture coordinates set up to *(0, 0)*, *(0, 1)*, *(1, 1)* and *(1, 0)*), then the resulting polygon will be bilinear filtered, again resulting in potentially unexpected blurring of the image.

When to call the RealiMation 2D Functions

The order in which the functions are called (and hence the drawing order) will affect the end result. Later 2D commands draw over earlier ones, like a painter painting over his/her earlier work.

The normal use would be to call the drawing functions between the call to `RTDisplayView()` and `RTSwapPage()` :

```
RTDisplayView (view_id);

// Place the 2D commands here
rtPixel p1, p2;

p1.x = 10;
p1.y = 10;
p2.x = 90;
p2.y = 90;

RT2DRectangle (channel_id, &p1, &p2);

// Now call swap page to display the actual result
RTSwapPage (channel_id);
```

This approach allows 2D overlays to be 'built-up'. One example of this was described earlier, with small individual controls being drawn over a 2D backdrop representing a cockpit.

Summary

However complicated or simple the overlays are, they can significantly enhance a 3D application. RealiMation provides several methods of generating overlays, each with it's own individual features. This document has shown the functions which can be used to generate 2D overlays, along with examples and ideas of their use.