Fast Image Reference Docutute v 0.2

A reference work and guide for MixailV's awesome FastImage .dll for Blitz3D
Created by Shifty Geezer, © 2007. www.softwaregeezers.com

This guide is a sort of reference cum tutorial for MixailV's FastImage library. It walks through setting up drawing and using commands, with detailed reference to command parameters and functions. It is not a translation of MixailV's own documentation in Russian! It's a 'reverse engineered' document from checking the example files and trying out the functions scientifically. It is a verbose reference that makes no assumptions, explaining everything even when it's mind numbingly obvious (like x, y, and width parameters!). It doesn't cover every command yet.

Conventions in this reference doc.

The Fast Image function names have been preceded with FI_. I have implemented this change in the .decls file and Include, and use the functions as named here. If you use the default files, drop the FI_ prefix from all functions.

Fast Image functions are marked in blue
Blitz3D functions are marked in sky-blue
Parameters are listed in red
Flag comments are green
Custom terms for are green

Times New-roman is used for user actions

    Indented Arial Narrow is used for code and definitions

Call FI_InitDraw with this command…

```
FI_InitDraw SystemProperty("Direct3Ddevice7")
```

Load a texture for your Fast-image

```
tex = LoadTextureImage("...", Flags)
```

Set 'Flags' to '1 + 2' for colour and alpha blending

Create a Fast-image from the texture

```
FI_CreateImageEx (tex, width, height, flags)
tex = texture object from loaded texture
width = width of fast_image
height = height of fast_image
flags =
    FI_AUTOFLAGS = -1       - Centre handle, filtered
    FI_NONE = 0             - Top-left handle, unfiltered
    FI_MIDHANDLE = 1        - Centre handle, unfiltered
    FI_FILTEREDIMAGE = 2    - Top-left handle, filtered
    FI_FILTERED = 2         - Same
```

Filtered images have smooth scaling and rotation (bilinear filtering). Unfiltered are pixelated and shimmer.

To start FI drawing, call FI_StartDraw

FI_StartDraw

Set blend type

FI_SetBlend (flags)

flags :

FI_SOLIDBLEND = 0	- Matte colour only
FI_ALPHABLEND = 1	- Alpha blended colour
FI_LIGHTBLEND = 2	- Additive blending
FI_SHADEBLEND = 3	- Seems broken? Should be multiply?
FI_MASKBLEND = 4	- Matte image masked at alpha < 0.5
FI_MASKBLEND2 = 5	- Alpha image masked at alpha < 0.5
FI_INVALPHABLEND = 6	- Seems broken?

Sets the drawing blend type for all subsequent drawing operations.

Set alpha level

FI_SetAlpha (float#)

float# ranges from 0.0 (fully transparent) to 1.0 (fully opaque)

Sets the alpha blend amount for all subsequent draw operations.

Draw complete Fast-image

FI_DrawImageEx (FImg, x, y, frame)

FImg = Fast-image object

x = horizontal position in pixels

y = vertical position in pixels

frame = animation frame of anim-texture

Draws image FImg to the current framebuffer.

Complex draw of a Fast-image, allows drawing of parts of an image, scaling, and tiling

FI_DrawImagePart (FImg, x, y, width, height, tx=0, ty=0, twidth=0, theight=0, frame=0, wrap=0)

FImg = Fast-image object

x = horizontal position in pixels

y = vertical position in pixels

width = width of Fast-image to draw

height = height of Fast-image to draw

tx = left coordinate of rectangular region of image to copy

ty = top coordinate of rectangular region of image to copy

twidth = Width of region to copy

theight = height of region to copy

frame = frame of anim-texture

wrap = wrapping mode flags :

FI_NOWRAP = 0	- No wrapping
FI_WRAPU = 1	- Image is duplicated in horizontal
FI_MIRRORU = 2	- As FI_WRAPU but also image is alternately flipped horizontally
FI_WRAPV = 4	- Image is duplicated in vertical
FI_MORRORV = 8	- As FI_WRAPV but also image is alternately flipped vertically
FI_WRAPUV = 5	- Duplicates in both directions
FI_MIRRORUV = 10	- Flips duplicates in both directions

This function requires a fair bit of explaining, but it's pretty straightforward and quite powerful. FI_DrawImagePart draws an area of size (width, height) at position (x,y). I will call this area Screen-Area. The function fills this Screen-Area rectangle with a part of the Fast-image defined by (tx,ty) to (tx+twidth,ty+theight). I will call this the Image-Portion. Without any wrapping flags, the Image-Portion is scaled to fit the Screen-Area. Taking a 256x256 Fast-image, here's some examples :

Halve the sprite :
Grab an Image-Portion of 256 x 256 and render to a Screen-Area of 128 x 128
width, height = 128; twidth, theight = 256

Double sprite :
Grab an Image-Portion of all the image, 256 x 256, and render to a Screen-Area of 512 x 512
width, height = 512; twidth, theight = 256

Double centre portion :
Select the central area Image-Portion and render at twice size
tx, ty = 64; twidth, theight = 128; width, height = 128

If the Image-Portion is larger than the Fast-image, or lies outside the Fast-image boundaries, it's filled with blank. Thus if you have a 256x256 Fast-image, and select an Image-Portion of 1024x256, you will have 3x the image width of blank space to the image's right. If you render that to the Screen-Area of size 256x256, the image will be squashed to fit, with a skinny image :

Squash large Image_Portion :
Twidth = 1024, theight = 256; width, height = 256

If you render the Screen-Area the same size as the Image-Portion, you'll have a lot of blank space to the image's right.

Lots of blank space :
Twidth, width = 1024; theight, height = 256

Now if you enable the UV wrapping with the wrap flags, the Image-Portion is tiled accordingly. To tile a 256x256 Fast-image 4x in the horizontal direction, set an Image-Portion 4x the width of the Fast-Image and set wrap to FI_WRAPU. This can be rendered to Screen-Area either in normal size, setting the Screen-Area width to the same as the Image-Portion width (twidth) :

Tiled image :
Twidth, width = 1024; theight, height = 256; wrap = FI_WRAPU

… or you can squeeze four copies of the image into an smaller width to make them skinny. Setting a Screen-Area width of 512 will halve the width of the drawn images :

Tiled image squeezed into narrower space, halving their width:
Twidth = 1024; width = 512, theight, height = 256; wrap = FI_WRAPU

Finish Fast Image drawing with FI_EndDraw

FI_EndDraw

The above allows you to add sprites with alpha blending to any application. It's awesome! You can also combine 3D between any layers of 2D operations you want. For example, draw a

background in 2D, render 3D objects, and add 2D elements over the top. Just remember to set Blitz3D's camera cls mode to not clear the colour.

```
CameraClsMode cam, 0, 1
Main Loop
        FI_StartDraw
                ...
                (FI Drawing instructions)
                ...
        FI_EndDraw
        RenderWorld
        FI_StartDraw
                ...
                (FI Drawing instructions)
                ...
        FI_EndDraw
End Main Loop
```

With this, you can draw a 2D tiled background say, populate it with 3D game characters or objects, and add super fast and sassy 2D particle effects on top before drawing the UI elements. Or render a 3D world and populate it with 2D characters. You could even set the camera draw limits to render different depths in different passes, and add 2D elements in-between.

That's the very basics covered. Here's some more options -

Creating a custom Blend mode

FI_SetCustomBlend SrcBlend, DestBlend

SrcBlend = Blend type on source values

DestBlend = Blend type on destination values

This creates a custom blend using DirectX 7's blend modes. The colour that appears on screen is:
Final Color = SourceColor * SourceBlendFactor + DestColor * DestinationBlendFactor
SourceColor is the colour of the Fast image pixel being drawn, and DestColor is the current framebuffer colour.
The SourceBlendFactor and DestinationBlendFactor are set from the SrcBlend and DestBlend values. Each value corresponds to a different algorithm in the blend mode.

D3DBLEND_ZERO	(0, 0, 0, 0)	
D3DBLEND_ONE	(1, 1, 1, 1)	
D3DBLEND_SRCCOLOR	(R_s, G_s, B_s, A_s)	
D3DBLEND_INVSRCCOLOR	(1–R_s, 1–G_s, 1–B_s, 1–A_s)	
D3DBLEND_SRCALPHA	(A_s, A_s, A_s, A_s)	
D3DBLEND_INVSRCALPHA	(1–A_s, 1–A_s, 1–A_s, 1–A_s)	
D3DBLEND_DESTALPHA	(A_d, A_d, A_d, A_d)	
D3DBLEND_INVDESTALPHA	(1–A_d, 1–A_d, 1–A_d, 1–A_d)	
D3DBLEND_DESTCOLOR	(R_d, G_d, B_d, A_d)	
D3DBLEND_INVDESTCOLOR	(1–R_d, 1–G_d, 1–B_d, 1–A_d)	
D3DBLEND_SRCALPHASAT	(f, f, f, 1); f = min(A_s, 1–A_d)	
D3DBLEND_BOTHSRCALPHA	Obsolete	For DirectX 6.0 and later, you can achieve the same affect by setting the source and destination blend factors to D3DBLEND_SRCALPHA and D3DBLEND_INVSRCALPHA in separate calls.
D3DBLEND_BOTHINVSRCALPHA	SRC (1–A_s, 1–A_s, 1–A_s, 1–A_s), DST (A_s, A_s, A_s, A_s);	The destination blend selection is overridden. This blend mode is supported only for the D3DRENDERSTATE_SRCBLEND render state.

The above is a table of D3D's types. You can add these as constants to your FastImage.bb include file with the following :

```
Const D3DBLEND_ZERO = 1
Const D3DBLEND_ONE = 2
Const D3DBLEND_SRCCOLOR = 3
Const D3DBLEND_INVSRCCOLOR = 4
Const D3DBLEND_SRCALPHA = 5
Const D3DBLEND_INVSRCALPHA = 6
Const D3DBLEND_DESTALPHA = 7
Const D3DBLEND_INVDESTALPHA = 8
Const D3DBLEND_DESTCOLOR = 9
Const D3DBLEND_INVDESTCOLOR = 10
Const D3DBLEND_SRCALPHASAT = 11
Const D3DBLEND_BOTHSRCALPHA = 12
Const D3DBLEND_BOTHINVSRCALPHA = 13
```

The standard alpha blend mode is achieved with D3DBLEND_BOTHSRCALPHA or
FI_SetCustomBlend D3DBLEND_SRCALPHA, D3DBLEND_INVSRCALPHA
This is a linear interpolation of the source and destination colour values based on source alpha.

You can add the colour values of source and destination without regard for alpha by using
FI_SetCustomBlend D3DBLEND_ONE, D3DBLEND_ONE

For multiply blending (no alpha) use
FI_SetCustomBlend D3DBLEND_DESTCOLOR, D3DBLEND_ZERO

Change the colour filter

FI_SetColor red, green, blue
red = red multiplier
green = green multiplier
blue = blue multiplier

Filters all subsequent draw operations by multiplying each colour component by (amount/255). Thus a value of 255 renders all colours at original brightness. A value of 128 halves the brightness of that colour component. To render an image in it's red, green and yellow component, and lose or blue data, you'd use :
FI_SetColor 255, 255, 0

Whites will be yellow, yellow will be yellow, purples will be red, and cyans will be green. You can think of this function as holding up a colour filter over the image.

Change the filter for each corner

FI_SetCustomColor top_ left, top_ right, bottom_ left, bottom _ right
top_left = colour multiplier for top-left corner
top_right = colour multiplier for top-right corner
bottom_ left = colour multiplier for bottom-left corner
bottom_right = colour multiplier for bottom-right corner

This works the same as above, but with a colour+alpha value for each corner of the image. The format of each value is $aarrggbb, with hexadecimal entry for each value. You can create gradients easily, or fade out an image on one side. Eg. To fade out the right hand side, set the filter values on the right to 0 for the alpha :
FI_SetCustomColor $FFFFFFFF, $00FFFFFF, $FFFFFFFF, $00FFFFFF

To gradate the image from colour at the top to black at the bottom, use :
FI_SetCustomColor $FFFFFFFF, $FFFFFFFF, $FF000000, $FF000000

And to colour the top through a yellow filter, and the bottom through a blur filter, with the bottom-right transparent :
FI_SetCustomColor $FFFFFF00, $FFFFFF00, $FF0000FF, $FF0000FF

Rotate Drawing operations

FI_SetRotation angle#
angle# = angle of rotation, clockwise, in Cartesian degrees

Sets the rotation of all subsequent drawing operations by angle# degrees clockwise.

‘Simple’ operations

In the drawing operations for shapes, like rectangles and ovals, there are two modes. The normal version of the function rotates the shape by the previously set FI_SetRotation amount. The simple versions ignore rotation commands and are faster.