11 11 TEX LATEX

Hermes 1.2 API

Christian Nentwich (brn), c.nentwich@cs.ucl.ac.uk 04/12/1998

Contents

1 Initialisation of the Library

Hermes has to be fully initialised before any conversion routines can be called. Otherwise error codes will be returned. All initialisation routines are reference counted, so feel free to call them repeatedly, they will only be initialised once (or if you closed it down and open it again).

1.1 int Hermes_Init()

Hermes_Init will initialise Hermes and return a non-zero status code on success. In detail, this is what happens:

- Check reference count. If non-zero, return success immediately.
- If assembler support is compiled in, run the processor detection routine. Hold your breath.
- Call the Converter Factory to initialise all converters with the fastest possible configuration.
- Initialise clearing routines using the Clearing Factory.
- Initialise dithering matrices.
- Increase reference count, return success.

1.2 int Hermes_Done()

Hermes_Done will decrease the reference count of HERMES and if it goes down to zero, it will deinitialise the library. That means it will get rid of all the converter routines, etc.

If the reference count ever falls below zero, this routine will return an error because you've done something wrong.

2 Palette Handling

If you want to use 8 bit indexed modes, you will have to use palettes. Also, if you want to use indexed modes, you will have to tell HERMES what palette you are using so it can create lookup tables, and so on.

HERMES needs lookup tables in order to convert efficiently from 8 bit to other formats without worrying you. If you just make a few function calls, HERMES will make sure your 8 bit mode works on any other mode.

As a last remark, palettes are stored as 8 bit per colour component values, NOT 6 bit. Keep it in mind.

2.1 HermesHandle Hermes_PaletteInstance()

This function returns a handle for you to use in subsequent palette handling operations. It will return zero on failure. Store this handle somewhere, you will need it for the following functions. This has to be the first function to be called before any palette handling.

2.2 void Hermes_PaletteReturn(HermesHandle handle)

If you are finished, use this to return your palette. This will free up the memory occupied by the palette and all the cached tables, etc. If you pass an invalid handle, your call will be ignored.

2.3 void Hermes_PaletteSet(HermesHandle handle,int32 *palette)

Surely, if you have a palette, you might want to set it. Use this function to copy 256 4-byte values from the pointer provided into the internal HERMES palette specified by handle. It would be a wise thing not to use this at all, i.e. not to store your own palette, but use the one already in HERMES, otherwise you are storing everything twice. In this case, look at the next function that says how to get access to the internal palette. But read on for caveats.

If you call this function, HERMES will assume that your palette has changed and flag all it's lookup tables invalid. So do NOT call it in a loop or anything unless your palette really does change. Lookup table recreation does not take place in there, however, but only when you actually convert something.

CAVEAT: This is the only function that invalidates the lookup tables of HERMES. If you use the next function, Hermes_PaletteGet, to get a pointer and then modify the palette, HERMES will not know that it has changed. Thus, any conversion to a non-indexed mode will draw garbage. You can use the Hermes_PaletteInvalidateCache in this case to force regeneration of lookup tables upon the next conversion.

2.4 int32* Hermes_PaletteGet(HermesHandle handle)

As said up there somewhere, it is probably wiser to let HERMES store the palette you want to use as it has to do it anyway. You can use this function to get a pointer to a 256-integer array of colour values.

Feel free to modify the palette, but keep in mind what is written above, you will trick HERMES and its lookup tables will be invalid. Call the next function documented below to resolve this.

Also, HERMES will return 0 if you pass an invalid handle. Take care of this as it will probably cause you a segmentation fault.

2.5 void Hermes_PaletteInvalidateCache(HermesHandle handle)

As described above, this function will cause HERMES to mark invalid all its lookup tables. As a consequence, when the next conversion from 8 bit is done, one of them will have to be regenerated. (By the way, 'regenerate a lookup table' sounds like pulling your teeth out. It is not. It's a minor thing, just a 256 byte table, no need to worry)

3 Format Conversion

There are a few steps involved in getting your buffers / images converted. Initialisation, request for conversion, palette setting and the actual conversion.

The routines below will normally be called in the following order to do what they are supposed to do:

- Hermes_ConverterInstance
- Loop starts here:
- Hermes_ConverterRequest
- Hermes_ConverterPalette
- Hermes_ConverterCopy
- Go to Loop
- Hermes_ConverterReturn

You will have to call Hermes_ConverterPalette every time before copying because your palette might have changed. And if there is a conversion from 8 bit to a direct colour format then HERMES has to update its lookup tables.

WARNING: Do not attempt to save time by not calling Hermes_ConverterPalette or Hermes_ConverterRequest in your program's main loop. You have nothing to be afraid of, HERMES caches everything for you. If nothing changes, HERMES returns immediately. Omitting the calls will only cause problems.

3.1 HermesHandle Hermes_ConverterInstance(unsigned long flags)

This will generate a new converter instance and return it to you. If an error occurs, 0 will be returned instead. Don't forget to check. At the moment, you can pass the following flags:

- HERMES_CONVERT_DITHER - Use dithering where appropriate and a converter is available. That will make every effort to dither but if it's an exotic conversion and no one has written the converter yet, it will fall back to normal conversion

Additional notes: Hermes keeps a dynamic array of converter instances that will grow if it runs out of space. You can change the default size and growth of the array in HermConf.h before compiling.

Keep the handle this function returns somewhere safe, you will need it for any other conversion function (and you can use it for locking if you want to be thread safe!)

3.2 void Hermes_ConverterReturn(HermesHandle handle)

Return the conversion routine specified by handle to HERMES to free up its spaces. This will get rid of all caches, lookup tables, etc. associated with this converter. Try not to create and destroy converters too often, it is costly. After all, the main effort in HERMES was to shift everything from loop time to initialisation time.

HERMES will ignore attempts to return non-existing handles.

3.3 int Hermes_ConverterRequest(...)

Complete function declaration: Hermes_ConverterRequest(HermesHandle handle, HermesFormat *source, HermesFormat *dest).

This will instruct Hermes to find a converter from the given source format to the given destination format and store it in the structure associated with handle. (Get a converter handle using Hermes_ConverterInstance).

As mentioned before, don't be shy, call it every time before you call Hermes_ConverterCopy unless you are really sure that the size of your window, colour format etc. doesn't change. You don't have to be afraid as any calls without changes will return success immediately.

The way HERMES will find your converter is the following:

- Check if anything has changed (colour formats, etc.). If no, return
- If the two colour formats are equal, return optimised converters
- Look for a specialised converter for the two formats
- If nothing has been found so far, find a generic converter

This function will return a non-zero value if a converter could be found, otherwise zero will be returned.

3.4 int Hermes_ConverterPalette(...)

Complete function declaration: Hermes_ConverterPalette(HermesHandle handle, HermesHandle sourcepal, HermesHandle destpal)

handle is the converter handle you got by using Hermes_ConverterInstance(). sourcepal and destpal are two palette handles which you have to get using Hermes_PaletteInstance().

This function will instruct HERMES to use the specified palettes for format conversion. At the moment, only sourcepal is of interest. It will be used whenever you want to convert from an indexed 8 bit format to any destination format. You may pass the sourcepal handle as destpal for the time being, it has no effect.

If converting from 8 bit to a non-indexed format, HERMES has to create a lookup table. In order to prevent any cycle-wasting, HERMES will cache those lookup tables for you. Whenever you change the palette, however - using Hermes_PaletteSet - the lookup tables have to be recalculated. In any case, generating lookup tables is a minor effort.

If any of the handles you pass are invalid or some other error occurs, HERMES will return zero on this function. Otherwise any non-zero value is returned.

3.5 int Hermes_ConverterCopy(...)

Complete function declaration: Hermes_ConverterCopy(HermesHandle handle, void *s_pixels,int s_x,int s_y,int s_width,int s_height,int s_pitch, void *d_pixels,int d_x,int d_y,int d_width,int d_height,int d_pitch)

Now, this is the real things and it has so many parameters that I might as well make a list out of them :)

- HermesHandle handle - the converter handle from Hermes_ConverterInstance
- void *s_pixels - pointer to the beginning of your source buffer. It is void because it might be any kind of format
- int s_x - x coordinate of top left corner in the source buffer, in pixels, not bytes!
- int s_y - y coordinate of top left corner in the source buffer
- int s_width - width of the area of the source buffer to be converted, in pixels!
- int s_height - height of the area of the source buffer to be converted
- int s_pitch - the pitch of the source buffer, more below.
- void *d_pixels - pointer to the beginning of the destination buffer
- int d_x - x coordinate of top left corner in the destination buffer, in pixels, not bytes!
- int d_y - y coordinate of top left corner in the destination buffer
- int d_width - width of the converted area in the destination buffer, int pixels, not bytes!
- int d_height - height of the converted area in the destination buffer
- int d_pitch - pitch of the destination buffer, more below

And, before I forget it, coordinates and widths up there are in pixels, not bytes! (That should prevent you from making mistakes now :)

The **pitch** of a buffer is basically the number of bytes in a scanline. This is normally equal to the width of the scanline in pixels times the number of bytes in a pixel. However, one some platforms this is not true.. and somebody might want to align their scanlines and they would find this quite comfortable.

In other words, the pitch is the amount in bytes to add to the beginning of a scanline in order to get to the beginning of the next scanline. Read again: amount in BYTES this time, NOT pixels. Beware!

There are a few things that can cause this function to return 0 (which indicates an error):

1. You specified an invalid handle
2. The source width or source height does not match the destination width or destination height AND no stretch converter can be found

From this, the policy of this function should be apparent but shall be made a bit more explicit:

- If the dithering flag has been set, find a dithering routine, otherwise fall back to normal

- If s_width!=d_width or s_height!=d_height then a stretching routine has to be used
- Otherwise do a normal conversion

It is probably also worthwhile noting that even though this all has a **Converter** prefix, if the source and destination formats are equal, then straight copying will be used.

4 Surface Clearing

HERMES provides support for clearing surfaces (i.e. blocks of memory filled with pixels) quickly and cleanly. Clearing routines exist in C, x86 assembler and MMX assembler versions and thus are fast. However, keep in mind that if you have the chance of using a hardware accelerated routine for this, it will still be faster.

4.1 HermesHandle Hermes_ClearerInstance()

Not much to say about this, same as the other *Instance() functions. You have to call this to obtain a handle before you use any other clearing routines.

This routine can return zero upon failure.

4.2 void Hermes_ClearerReturn(HermesHandle handle)

Again, if you are finished clearing surfaces (and that means completely finished, call it only at the end of a program, not every time you have just cleared a surface) use this function to free up any memory inside Hermes.

4.3 int Hermes_ClearerRequest(HermesHandle handle,HermesFormat *format)

This function will tell HERMES to prepare a special clearer for a subsequent clear operation. Again, you may call this as often as you want, once it has been set up, subsequent calls will be cached.

Zero will be returned if your handle is invalid or the format you requested cannot be cleared (all formats are implemented, thus you will have to request a pretty fucked up format in order to fail this :). Upon success, any non-zero value may be returned.

4.4 int Hermes_ClearerClear(...)

ClearerClear is the function that actually does the work. The request function has to be called before this, otherwise it will fail. The following variables have to be passed to the function:

- HermesHandle handle - the handle you have obtained by calling Hermes_ClearerInstance().
- void *pixels - the buffer to clear. It is void because it might be any format that has been requested in Hermes_ClearerRequest()

- int x1 - Upper left corner x coordinate IN PIXELS not bytes
- int y1 - Upper left corner y coordinate in pixels
- int width - Width of area to be cleared in pixels
- int height - Height of area to be cleared in pixels
- int pitch - Amount IN BYTES to add in order to get from the beginning of a scanline to the beginning of the next. See the conversion section for more on this.
- int32 r - For direct colour, the red component, between 0-255 !!
- int32 g - For direct colour, green between 0-255
- int32 b - For direct colour, blue between 0-255
- char8 index - For indexed modes, the colour index to use for clearing. For direct colour modes, the alpha value!

Hermes_ClearerClear returns any non-zero value on success. If zero is returned, the clearing has failed, most of the time because the handle you have passed was invalid.