# *Revolution*

# *Video Interface Library (VI)*

**Version 1.00**

# *Video Interface Library (VI)*

**Version 1.00**

## Contents



## Code Examples

## Figures



## Tables

# Revision History

| Revision No. | Date Revised | Items (Chapter) | Description | Revised By |
|---|---|---|---|---|
| Version 1.00 | 3/22/2006 | - | First release by Nintendo of America, Inc. | - |

# 1 Overview

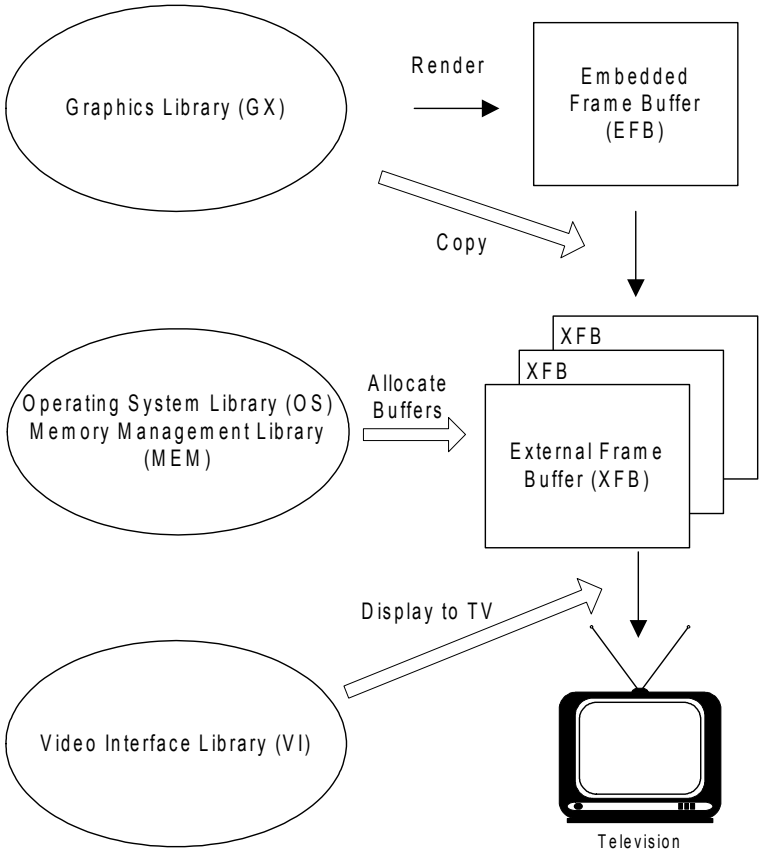
The Video Interface library (VI) contains functions to control the Video Interface hardware. This document shows how to use the VI library and what you can accomplish with it.

# 2 Relationship of VI to other system libraries

The main job of VI is to display the external frame buffers (XFB) generated by the Graphics library (GX). External frame buffers are allocated in main memory; the number allocated depends on whether you are using a double- or triple-buffer display technique.

**Note:** GX renders the embedded frame buffer (EFB) and then copies it to the XFB after rendering.

**Figure 1 - Relationship between system libraries**




**Note:** In this document, the term “frame buffer” (or simply “FB”) always refers to an *external* frame buffer because the VI library touches only this type. We use the abbreviations XFB and EFB to differentiate between external and embedded frame buffers, respectively, when referring to both types.

# 3 VI main features

Here are the main features of the VI hardware and library:

- NTSC, PAL, and M/PAL video formats.
- Additional video format for European market, called “EURGB60”.
- Strong support for interlaced display, as well as support for traditional non-interlaced display.
- Arbitrary positioning and sizing of image on television.
- Support for light gun target location.
- Horizontal scaling to maximize frame buffer coverage on the television.
- 4:2:2 YCbCr pixel format to reduce data size in frame buffer.

## 3.1 NTSC, PAL and M/PAL video formats

Like previous game consoles, the Nintendo GameCube Video Interface supports three video formats: NTSC, PAL and M/PAL. NTSC is the format used in Japan and North America, PAL is used in Europe, and M/PAL is used in Brazil. With NTSC and M/PAL formats, one field is rendered at 60Hz (i.e., one second = 60 fields). With PAL, one field is rendered at 50Hz.

## 3.2 Additional video format for European market, called “EURGB60”

Nintendo GameCube supports an additional video format for European market, which is called “EURGB60”. The EURGB60 format supports 60Hz 480-line signal, so you can easily convert your NTSC games to this mode when you are preparing European version. This special format can be seen on PAL TVs that support 60Hz signal with RGB SCART connector. This kind of TV is now becoming very popular.

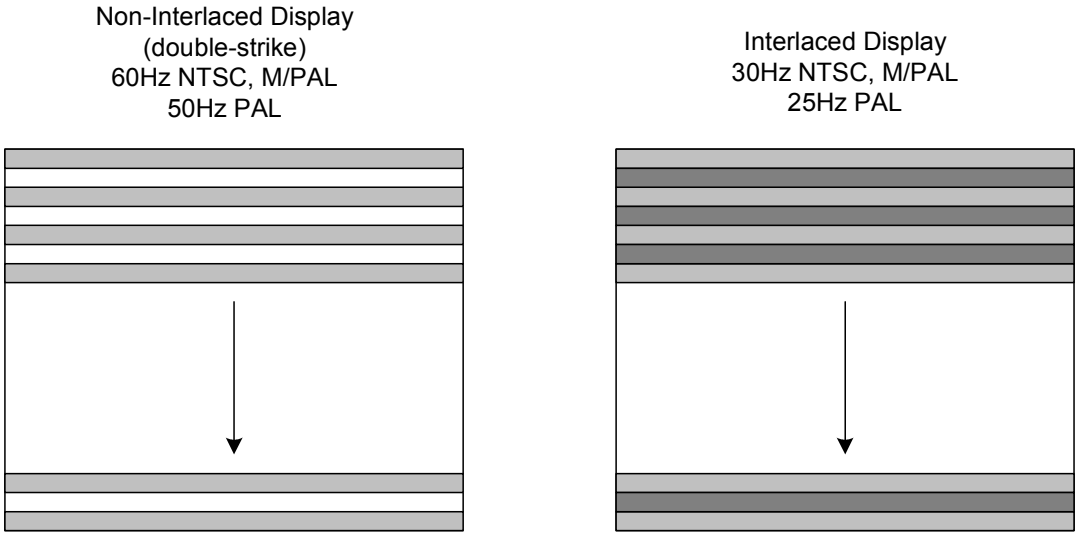
As the name indicates, EURGB60 can only be displayed through an RGB cable. The mode does not work with a composite AV cable.

**Note:** Since some percentage of European people are using TVs that can’t show this format, European games still need to support standard PAL format.

## 3.3 Strong support for interlaced display

Nintendo GameCube offers much better performance than prior generations of console systems, both in number of polygons per frame and in fill rate. This greater capability allows us to create higher-resolution images to impress the player. As a result, the VI library needs to strongly support interlaced display, rather than focusing on the traditional non-interlaced (double-strike) display commonly used in earlier consoles.

**Figure 2 - Video display modes**




As shown in the figure above, the TV scans only the even (gray) lines in double-strike mode; this results in slight but noticeable black lines between each line of the picture. By contrast, the TV scans both even (light gray) and odd (dark gray) lines in interlaced mode, which creates more area in which to see a higher resolution picture.

## 3.4 Arbitrary positioning and sizing of image on television

You can change the position and size of image onscreen very easily.

## 3.5 Support for light gun

The Nintendo GameCube system hardware supports a light gun (“Zapper”) feature.

**Note:** As of today (9/3/02), light gun hardware is not yet available.

## 3.6 Horizontal scaling to maximize frame buffer coverage onscreen

VI can scale the image horizontally, which means that you can have a narrower image in the frame buffer in order to save main memory. However, using this feature has its downside: the image shown onscreen might not be clear enough because of the effects of scaling. You should decide carefully whether to use horizontal scaling by checking the resulting image on the television screen.

## 3.7 4:2:2 YCbCr pixel format to reduce data size in frame buffer

Televisions receive composite input in YUV format. 4:4:4 YCbCr is a YUV format modified so that every component fits as unsigned 8-bit data. We are using 4:2:2 YCbCr format in XFB, which is down sampled from 4:4:4 YCbCr so that Cb and Cr are only sampled once in two horizontal pixels. Having the frame buffer use this pixel format is efficient because each pixel is described by two bytes (by contrast, RGB consumes three bytes for each pixel). The GX hardware performs RGB to YCbCr conversion when it copies the image from EFB to XFB.

4:2:2 YCbCr format is detailed in "12 Video output" on page 125 of the *Graphics Library (GX)* section.

# 4 Basic programming

In this chapter, we explain how to use the basic features of the VI library. The following basic APIs are introduced:

**Table 1 - VI basic APIs**

| Function | Purpose |
|---|---|
| `VIInit` | Initialize (detailed in "4.2.1 VIInit" on page 12). |
| `VIConfigure` | Configure video and frame buffer (detailed in "4.2.3 VIConfigure" on page 12). |
| `VISetNextFrameBuffer` | Set frame buffer address shown next (detailed in "4.4 Setting the frame buffer mode" on page 15). |
| `VISetBlack` | Black out screen (detailed in "4.5 Black out screen" on page 15). |
| `VIFlush` | Notify VI to update registers on next field (detailed in "4.3 Flush" on page 15). |
| `VIWaitForRetrace` | Wait for the next retrace interrupt (detailed in "4.6 Waiting for the next retrace" on page 16). |

**Note:** Do not forget to call `VIFlush` after you call `VIConfigure`, `VISetNextFrameBuffer`, `VISetBlack`, or `VIConfigurePan` (to be explained in "6.4 Pan" on page 28). Otherwise, the changes you made by calling these functions will not take effect. See "4.3 Flush" on page 15 for details.

## 4.1 Code sample: color.c

The following line-numbered excerpt is from a demo program called `color.c`, which paints the entire screen with a single color, then changes the color once per second. It displays in 640x480 interlaced mode and renders the whole screen at once, which means it prepares 480 lines for each frame buffer and renders all 480 lines at one time (this is known as double-field frame buffer mode; see "4.2.4 Types of frame buffers" on page 13 for details).

### Code 1 - Color.c

```
1       void main(void)
2       {
3           u32         frame;
4           u32         code;
5           u32         fbSize;
6           u8*         xfb;
7           u32         first;
8
9           OSInit();
10          VIInit();
11
12          // Calculate frame buffer size.
13          // NOTE: Each line width should be a multiple of 16.
14          fbSize = (u32)(VIPadFrameBufferWidth(rmode->fbWidth)
15              * rmode->xfbHeight * VI_DISPLAY_PIX_SZ);
16
17          allocateFB(fbSize);
18
19          VIConfigure(rmode);
20
21          // Need to "flush" so that the VI changes so far takes effect
22          // from the following field
23          VIFlush();
24          VIWaitForRetrace();
25
26          // Since the tv mode is interlace after VIInit,
27          // we need to wait for one more frame to make sure
28          // that the mode is switched from interlace to non-interlace
29      #ifdef NON_INTERLACE
30          VIWaitForRetrace();
31      #endif
32
33          first = 1;
34          frame = 0;
35          code = 0;
36
37          while(1)
38          {
39              xfb = (frame & 0x1)? xfb2 : xfb1;
40
41              if (frame % 60 == 0)
42              {
43                  code = (code + 1) % 8;
44              }
45
46              fillColor(code, fbSize, xfb);
47
48              VISetNextFrameBuffer((void*)xfb);
49
50              if (first == 1)
51              {
52                  VISetBlack(FALSE);
53                  first = 0;
54              }
```

```
55
56              VIFlush();
57              VIWaitForRetrace();
58
59              frame++;
60          }
61      }
```

Here is an explanation of the algorithm:

**Lines 14-17:** Allocates memory for frame buffers. This sample uses two double-field frame buffers (see "4.2.4 Types of frame buffers" on page 13 for details). In this sample, `rmode->fbWidth` and `rmode->xfbHeight` contain width and height data, respectively. `VI_DISPLAY_PIX_SZ` shows how many bytes are needed for one pixel and is defined as 2.

**Note:** We need to align the start address of each line so that each width is a multiple of 16. `VIPadFrameBufferWidth` is a useful macro to get the smallest multiple of 16 that is greater than or equal to the argument.

**Line 19:** Configures the size of the frame buffer, its mode (double-field or single-field), where on the TV screen to show the image, the TV format (NTSC, PAL, EURGB60 or M/PAL), display mode (interlaced or non-interlaced), and so on. Refer to "4.2.3 VIConfigure" on page 12 for details on how to use `VIConfigure()`.

**Line 23:** Notifies the VI device driver to effect the changes from the next field. `VIFlush` must be called when you use any of the following APIs:

- `VIConfigure`
- `VISetNextFrameBuffer`
- `VISetBlack`
- `VIConfigurePan`.

**Line 24:** Waits for the next field.

**Line 30:** This code runs in interlaced mode by default, so this line is not usually compiled. However, when we compile this code to run in non-interlaced mode (i.e., with the definition `NON_INTERLACE`), this line commands the program to wait at least two fields when we switch between interlaced and non-interlaced modes.

## 4.2 Initialization

Follow these steps to initialize the Video Interface library:

1. Run `VIInit()` before using any of the VI functions.
2. Allocate the appropriate number of frame buffers in main memory (e.g., two for a double-buffer display).
3. Call `VIConfigure()` to set the desired video mode, position the frame buffer, etc.

### 4.2.1 VIInit

Before calling any VI functions, you need to call `VIInit` to initialize the VI hardware and library.

**Code 2 - VIInit()**

```
#include<revolution/vi.h>

void VIInit(void);
```

**Note:** You should not assume that a field will start immediately upon calling this function. You should also not assume that a field will always start on an even or an odd line.

### 4.2.2 Frame buffer allocation

The frame buffer must be allocated from main memory, and it must follow these alignment rules:

- Each pixel is two (2) bytes and encoded in YCbCr format.
- The start of the frame buffer must be 32-byte aligned.
- The width of each frame buffer must be a multiple of 32 bytes. (Although it is possible to display frame buffers with widths that don't align perfectly to 32 bytes, the buffers must then be padded to 32 bytes.)

Since the function `OSAlloc()` in the Operating System library (OS) allocates only 32-byte aligned buffers, for expedience you can set up the OS memory heap and then use `OSAlloc()` to allocate the frame buffers. In addition, the function `VIPadFramebufferWidth()` can be useful in creating properly-padded frame buffers.

### 4.2.3 VIConfigure

`VIConfigure()` sets various configurations, such as:

- TV video format (NTSC, PAL, EURGB60, M/PAL).
- Display mode (interlaced or non-interlaced).
- Image size onscreen (*viWidth*, *viHeight*).
- Image size on frame buffer (*fbWidth*, *xfbHeight*).
- Image position onscreen (*viXOrigin*, *viYOrigin*).
- Frame buffer mode (double-field or single-field; see "4.2.4 Types of frame buffers" on page 13 for details).

**Code 3 - VIConfigure()**

```
#include<revolution/vi.h>
#include<revolution/gx.h>

void VIConfigure(GXRenderModeObj* rm);
```

You can customize your configuration as you like; however, we encourage you to get the working prototypes first by using the defaults predefined in the `GX[Ntsc|Pal|Eurgb60Hz|Mpal]XXXX` global variables. These variables are extern'd in `/revolution/include/revolution/gx/GXFramebuffer.h`. Here are some of the variables:

**Table 2 - Default configuration variables**

| Variable | Characteristics |
|---|---|
| GXNtsc240Ds | NTSC.<br>Non-interlaced (double-strike).<br>Image size on frame buffer: 640x240.<br>Image size onscreen: 640x480.<br>Image position centered onscreen.<br>Single-field frame buffer mode. |
| GXNtsc240Int | NTSC.<br>Interlaced.<br>Image size on frame buffer: 640x240.<br>Image size onscreen: 640x480.<br>Image position centered onscreen.<br>Single-field frame buffer mode. |
| GXNtsc480Int | NTSC.<br>Interlaced.<br>Image size on frame buffer: 640x480.<br>Image size onscreen: 640x480.<br>Image position centered onscreen.<br>Double-field frame buffer mode. |

See “Render Modes” in the online *Revolution Function Reference Manual* for more details.

### 4.2.4 Types of frame buffers

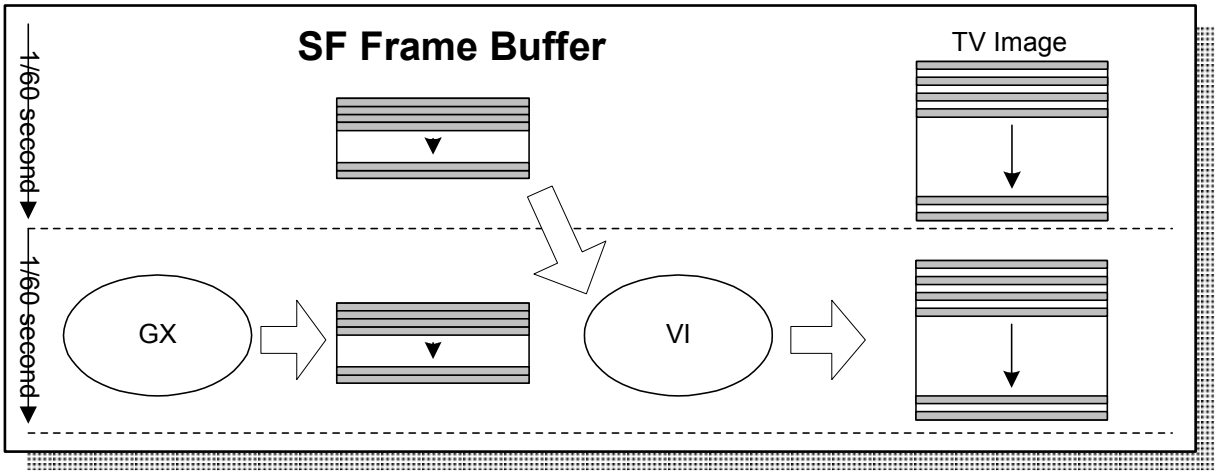
The GX library can generate two types of frame buffers, double-field (DF) and single-field (SF). DF frame buffers contain both even and odd lines that can be displayed on TV. SF frame buffers contain only even *or* odd lines.

The diagrams in "Figure 3 - Frame buffer types" on page 14 show several ways in which GX can pass the frame buffer(s) to VI to produce an image on an NTSC television. The suitability of a given method depends on the specifications of the game. Typically, fill-rate performance and high-resolution definition are the criteria for selecting a particular method.

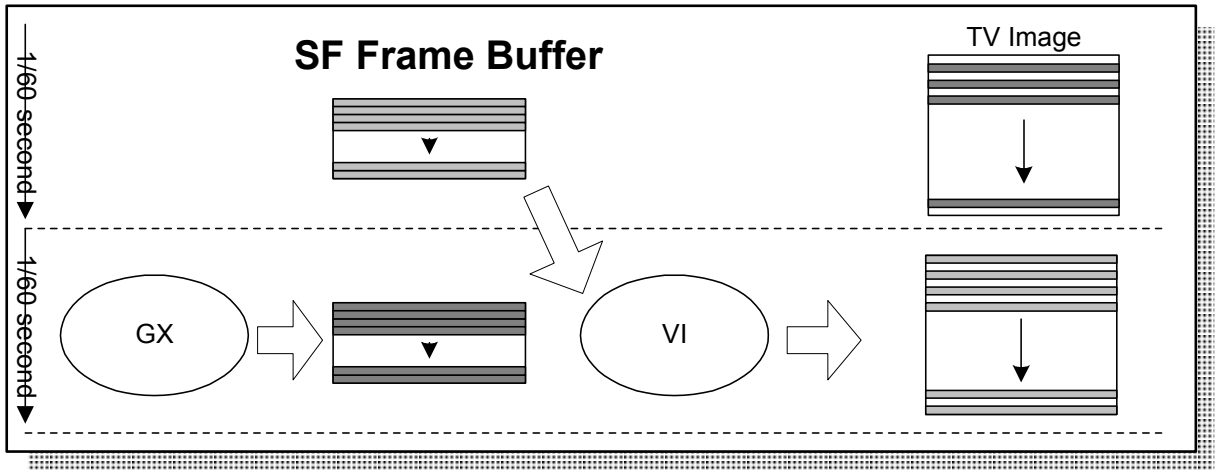
The diagrams, from top to bottom, correspond to *GXNtsc240Ds*, *GXNtsc240Int*, and *GXNtsc480Int*, respectively. For example, if you want to make your game use the method indicated in the first (top) diagram, pass the variable *GXNtsc240Ds* to `VIConfigure()`.

**Figure 3 - Frame buffer types**

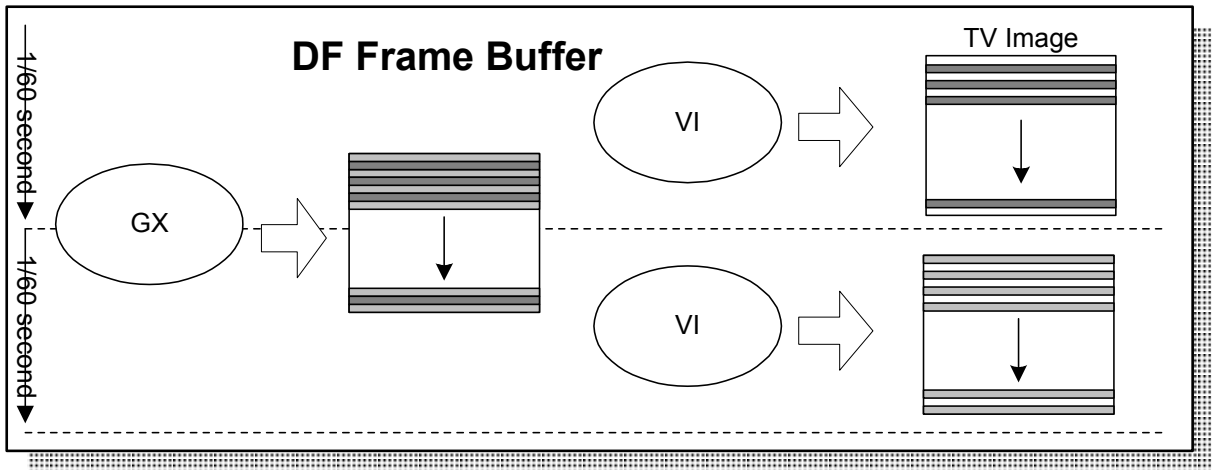
**GX renders 640x240 at 60Hz, VI switches buffer at 60Hz and displays in double-strike mode.**




**GX renders 640x240 at 60Hz, VI switches buffer at 60Hz and displays in interlaced mode.**




**GX renders 640x480 at 30Hz, VI switches buffer at 60Hz and displays in interlaced mode.**



## 4.3 Flush

`VIFlush` notifies the VI device driver to effect changes starting from the next field. It must be called when you call `VIConfigure`, `VISetNextFrameBuffer`, `VISetBlack`, or `VIConfigurePan`.

### Code 4 - VIFlush()

```
#include<revolution/vi.h>

void VIFlush(void);
```

**Note:** If you change the TV mode, whether from interlaced to non-interlaced (double-strike) mode or vice versa, you will need to wait at least two fields after calling `VIFlush` to make sure that the hardware changes the mode properly.

## 4.4 Setting the frame buffer mode

### Code 5 - VISetNextFrameBuffer()

```
#include<revolution/vi.h>

void VISetNextFrameBuffer(void* fb);
```

If you are using double- or triple-frame buffers, you need to call `VISetNextFrameBuffer()` to swap frame buffers. Once this function is called, the following conditions remain in effect until the function is called again:

- If the frame buffer mode is single-field, the same image will be used for the subsequent fields.
- If the frame buffer mode is double-field, the appropriate lines of the image are used for the subsequent fields; e.g., if the field is supposed to show odd lines, then odd lines of the frame buffer are used for the field.

## 4.5 Black out screen

You can use `VISetBlack()` to black out or show the screen.

### Code 6 - VISetBlack()

```
#include<revolution/vi.h>

void VISetBlack(BOOL black);
```

If you call this function with `TRUE` as the argument, as in `VISetBlack(TRUE)`, the screen will turn black. You can cancel this “black mode” by calling `VISetBlack(FALSE)`.

**Note:** As with other functions, the screen turns black at the next field after you’ve called `VIFlush`. In addition, the screen is always in “black mode” after a call to `VIInit`, so you need to call `VISetBlack(FALSE)` at least once after you’ve prepared the first image on the frame buffer.

## 4.6 Waiting for the next retrace

`VIWaitForRetrace()` is an easy way to synchronize your code with the VI hardware.

**Code 7 - VIWaitForRetrace()**

```
#include<revolution/vi.h>

void VIWaitForRetrace(void);
```

If you have registered a background job, the Operating System switches the context to start or resume the job after you call `VIWaitForRetrace()`. See “Operating System (OS)” in the *Nintendo GameCube Programmer’s Guide* for more information about background jobs.

# 5 European game support

## 5.1 Difference between NTSC and PAL

As most developers know, there are two big differences between NTSC and PAL: timing and the number of lines per field.

1. **Timing.** In the PAL system, only 50 fields are displayed per second, so one field is 20 milliseconds. This is about 20% longer than NTSC (16.6 ms/field). When you port your games to PAL, you might want to think about this timing issue. If you don't allow for it, your game is likely to run 20% slower than on the NTSC system.
2. **Number of lines per field.** In the NTSC system, one field contains 525 lines of which only 480 are "active"; that is, you can draw a picture only in those 480 lines (the other 45 lines are used for vertical retrace, etc.). In the PAL system, there are 625 lines in one field, and 574 out of 625 are active.

**Table 3 - Differences between NTSC and PAL formats**

| | Number of fields/second | Length of one field | Number of active lines |
|---|---|---|---|
| **NTSC** | 60 | 16.6ms | 480 |
| **PAL** | 50 | 20ms | 574 |

## 5.2 Solving the timing problem

Solving the timing problem completely depends on the games themselves; there is no general solution.

It is always a good idea when you design the NTSC version to plan in advance to make the game system scalable between 50Hz and 60Hz. Doing this can minimize the need for porting later.

If you are going to use 2D sprite animation you should be aware of this problem because porting 2D sprite animation from 60Hz to 50Hz may be difficult.

## 5.3 Solving the "extra lines" problem

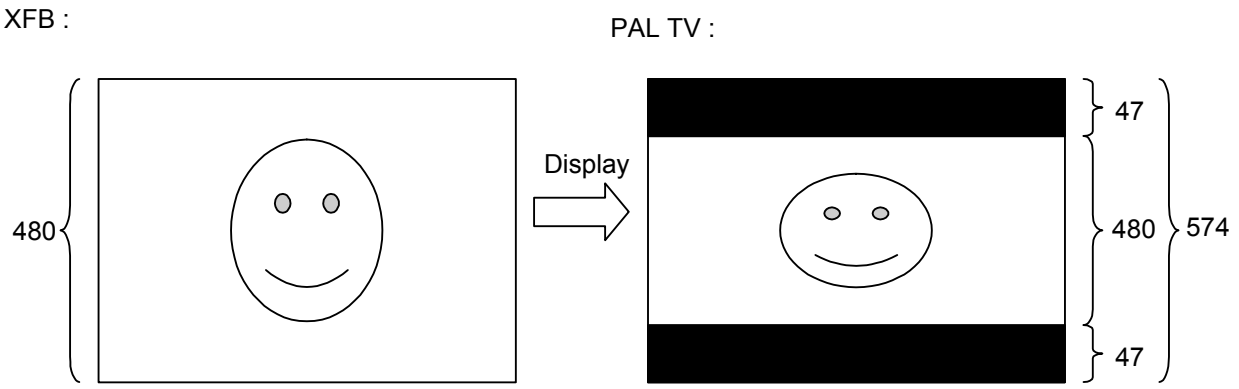
PAL games need to draw some extra lines compared to NTSC games, and this can be a problem when you port your NTSC games to PAL. Here are several solutions to solve this "extra lines" problem.

As stated previously, it is always a good idea when you design the NTSC version to make considerations for the PAL version as well.

### 5.3.1 Showing black bars at the top and bottom of the screen (not recommended)

This is a very simple solution, but we don't recommend it because the resulting PAL image will look vertically compressed compared to how it looks in NTSC format. If your game draws a full 480 lines in NTSC, you will need to draw 94 extra lines in PAL (574 – 480 = 94). One way to make up the difference is to show 47 black lines on the top and bottom of the screen. You can do this by changing the value of *viYOrigin* in `GXRenderModeObj` to be passed to `VIConfigure()`.

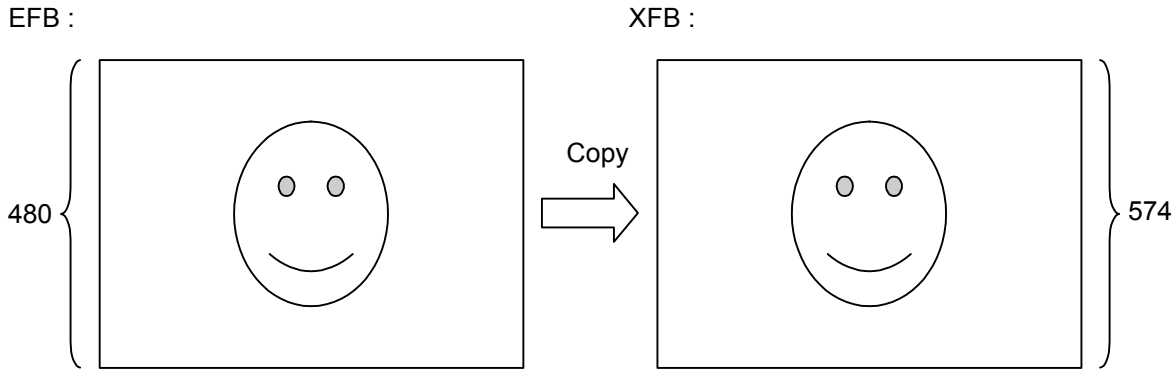
**Figure 4 - Using black bars at screen top and bottom**




### 5.3.2 Using Y scaling

The GX API has a feature that scales vertically when copying frame buffer data from the EFB to the XFB. You can take advantage of this feature to solve the “extra lines” problem. If your NTSC game draws 480 lines and you want to draw 574 lines for PAL, specify `GXGetYScaleFactor(480, 574)` to `GXSetDispCopyYScale()` before you copy (this assumes you are not using Y-scaling for the NTSC version).

**Figure 5 - Using Y scaling**




We’ve modified `smp-onetri` to use Y-scaling to show off this feature; please take a look at the demo in this patch. Y-scaling is a better solution than using black bars in terms of how it displays; the image will look almost the same as the original NTSC game. The disadvantage here is that you need to prepare a bigger XFB to hold the 574 lines.

**Notes:**

- It is not recommended that you pass 480/574 directly to `GXSetDispCopyYScale()`. See the `GXSetDispCopyYScale()` and `GXGetYScaleFactor()` pages in the *Revolution Function Reference Manual* (HTML) for more details.
- You cannot use this method if the CPU is drawing any part of the picture directly to XFB.

### 5.3.3 Rendering more lines

Simply rendering 574 lines is another solution. The quality of the result should be better than with the Y-scaling solution, but the disadvantage is that rendering will take more time than it does on the NTSC system. (However, this may not be an issue since you have 3.3ms more for each field.)

## 5.4 EURGB60 mode

Nintendo GameCube supports “EURGB60” mode, which is 60 fields/second and 525 lines (480 active lines) for European TVs using an RGB signal. We are supporting this mode now because many modern TVs sold in Europe support this timing unofficially or officially.

**Note:** Not all TVs used in Europe support this mode; some percentage of European consumers still use the older type of TVs.

Due to a hardware restriction, the Nintendo GameCube does not support this 60Hz mode for composite signals (hence the emphasis on “RGB” in the mode name).

Porting NTSC games to EURGB60 is pretty easy because the timing and resolution are the same. Change *viTVmode* in `GXRenderModeObj` from `VI_TVMODE_NTSC_*` to `VI_TVMODE_EURGB60_*`.

Supporting EURGB60 mode is not required; you can decide whether or not to support this mode on a per-title basis. The following are the advantages to supporting EURGB60 mode:

- Game players will feel more ease in terms of controlling characters, etc. on 60Hz games than on 50Hz games.
- It is easy to make the game the same speed as the NTSC version.
- Some people may notice less flickering at 60Hz than at 50Hz.

**Note:** You still need to support standard PAL timing even if EURGB60 is supported; as described above, not all the TVs used in Europe support EURGB60 timing.

## 5.5 Demos

The `onetri` demo has been modified so that you can use the A Button to switch TV modes between PAL (using Y scaling), PAL (using no scaling, with black bars) and EURGB60. You can find this demo under `videmo/src/smp-onetri_PAL.c`.

If your TV supports only NTSC, this demo will not show the proper output in PAL mode. Similarly, if your PAL TV doesn’t support EURGB60, the demo won’t show the proper output when you switch to NTSC.

Since this demo doesn’t adjust the frame rate, the PAL modes will run slower than the EURGB60 mode.

# 6 Further programming

## 6.1 Render mode customization

Although we recommend that you start by using the predefined rendering modes, we encourage you to customize your configuration once you've become comfortable with the API. In this section, we will show you how to set custom values for `VIConfigure()`.

### 6.1.1 Maximum screen space

To define the image size and position onscreen, we must first define the maximum space in which we can locate the image. We call this the *maximum screen space.* The size of this space, in pixels, is:

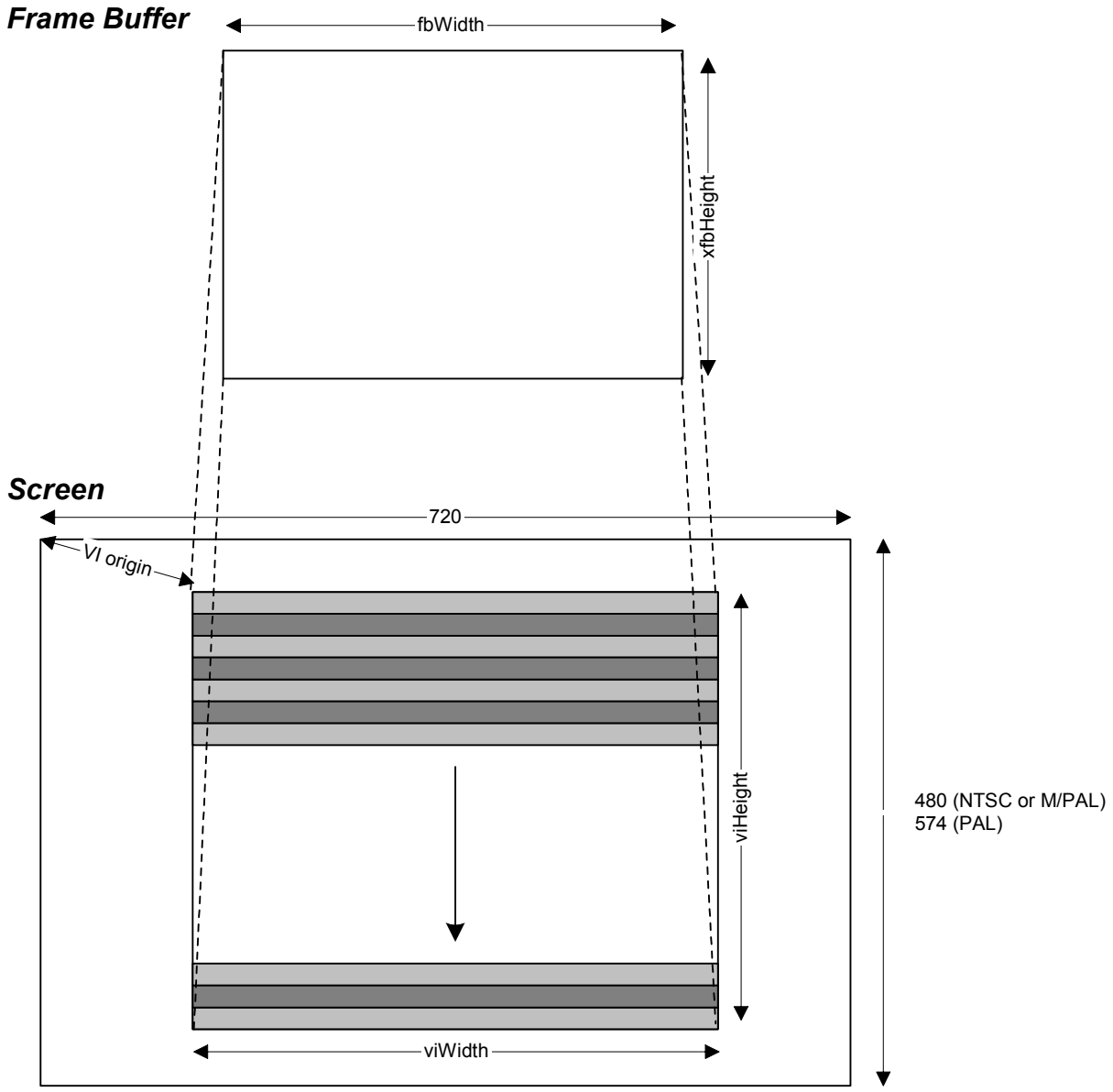
- 720x480 for NTSC, EURGB60 and M/PAL.
- 720x574 for PAL.

You can place images anywhere in the maximum screen space by using the function `VIConfigure()`.

### 6.1.2 Frame buffer size, TV screen size, and position configuration

There are six fields in the `GXRenderModeObj` structure concerned with frame buffer size, TV screen size, and position configuration. They are *fbWidth*, *xfbHeight*, *viXOrigin*, *viYOrigin*, *viWidth*, *viHeight*. The following figure shows the relationship between these values:

**Figure 6 - Relationship between the six values of GXRenderModeObj**




`GXRenderModeObj` uses the notation *xfbHeight*, specifically, to differentiate this value from the embedded frame buffer height, noted as *efbHeight*. The structure contains both values because the GX library can perform Y scaling when it copies the image from EFB to XFB. However, the VI library doesn't touch *efbHeight*.

If you specify *viWidth* as greater than *fbWidth*, VI performs the horizontal scaling. You cannot specify *viWidth* as lesser than *fbWidth*.

Since VI cannot perform Y scaling, the following simple ratios must be maintained between *viHeight* and *xfbHeight*:

- For double-field frame buffer mode, the value of *viHeight* should be the same as *xfbHeight*.
- For single-field frame buffer mode, the value of *viHeight* should be twice as large as *xfbHeight*. (This is true for non-interlaced (double-strike) mode; e.g., if *xfbHeight* is 200, *viHeight* should be 400.)

It is not strictly necessary to specify a value for *viHeight*, because this can be calculated easily from the value of *xfbHeight* and the FB mode, and `VIConfigure` verifies this value.

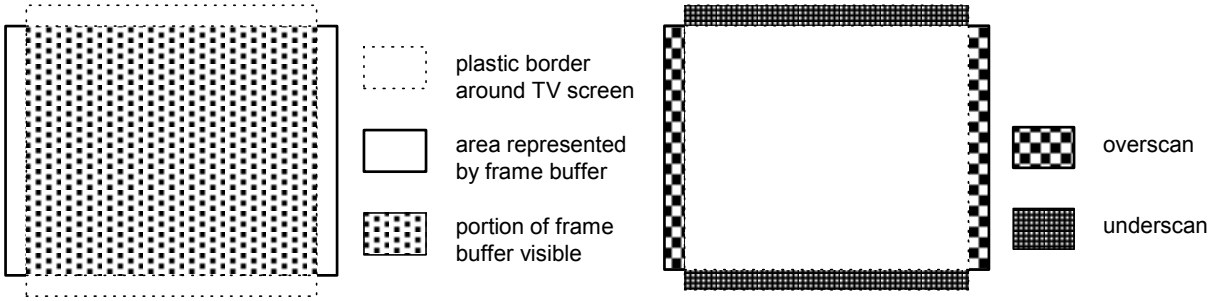
Here is an example. Let's say that you would like to create a cinema wide-screen aspect ratio of 9:16 with a width of 640, using a single-field frame buffer. To do so, you would specify the following values:

- *fbWidth* = *viWidth* = 640.
- *xfbHeight* = 180.
- *viHeight* = 360.
- *viXOrigin* = (720 – 640)/2.
- *viYOrigin* = (480 – 360)/2.

### 6.1.3 Overscanning and underscanning

Different brands of televisions exhibit slight differences in the amount of overscanning and underscanning for a given signal. *Overscanning* means that the TV's electronic beam is "overpainting" pixels in the area behind the plastic border surrounding the television tube. *Underscanning* means that some visible portion of the TV screen has no painted pixels. We have pre-selected parameters that typically make for a good display on most televisions.

**Figure 7 - Overscanning and underscanning**




In general, console manufacturers set horizontal defaults to over- or underscan slightly. With previous consoles, Nintendo has typically set horizontal defaults to overscan. This is because horizontal underscanning can cause a slight distortion of the vertical line separating the frame buffer image from the outside of that image, depending on the scan line colors within the frame buffer.

Some games underscan vertically in order to create an aspect ratio similar to that of a wide-screen movie, as shown in the example in "6.1.2 Frame buffer size, TV screen size, and position configuration" on page 22. The combination of overscanning horizontally while underscanning vertically is therefore a common one, and we present an example in the next section.

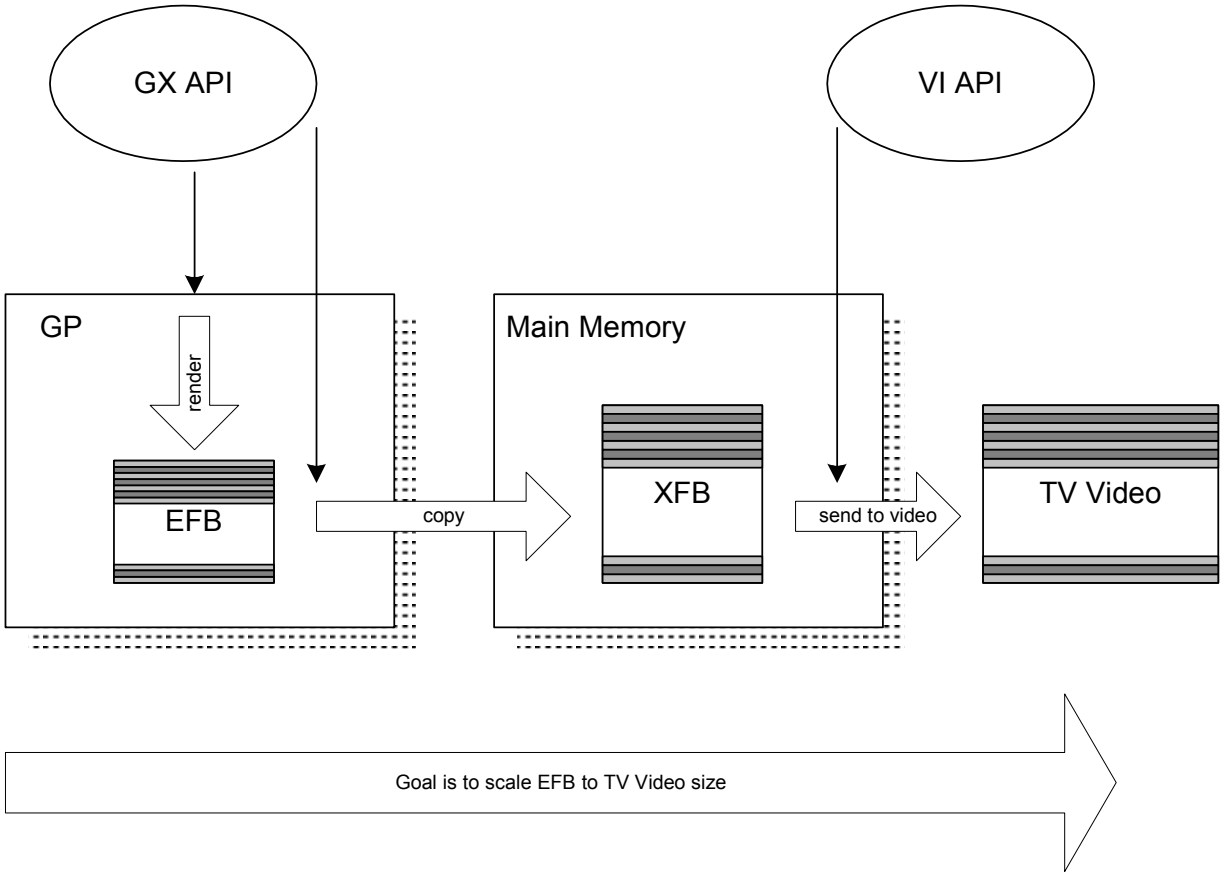
### 6.1.4 Scaling

You can scale horizontally if you specify *viWidth* as greater than *fbWidth*. Remember that you cannot specify a *viWidth* value smaller than *fbWidth*. For example, if you would like to have a 512x240 frame buffer and scale horizontally 640/512 = 1.25, using a single-field frame buffer, specify the following values:

- *fbWidth* = 512
- *viWidth* = 640
- *xfbHeight* = 240
- *viHeight* = 480
- *viXOrigin* = (720 – 640)/2
- *viYOrigin* = (480 – 480)/2

Keep in mind that the VI library cannot scale vertically; refer to the *Graphics Library (GX)* section for information on Y scaling. The following figure summarizes how to scale both vertically and horizontally, using the GX and VI libraries:

**Figure 8 - GX and VI functional flowchart**




In the figure above, *EFB* is the embedded frame buffer within the graphics processor (*GP*), *XFB* is the external frame buffer in main memory, and *TV Video* is what appears on the television screen. Since we want to scale EFB to the TV Video size in both *x* and *y* dimensions, the hardware must:

- Perform *y*-scaling while EFB is copied to XFB.
- Perform *x*-scaling while XFB is sent to the television screen.

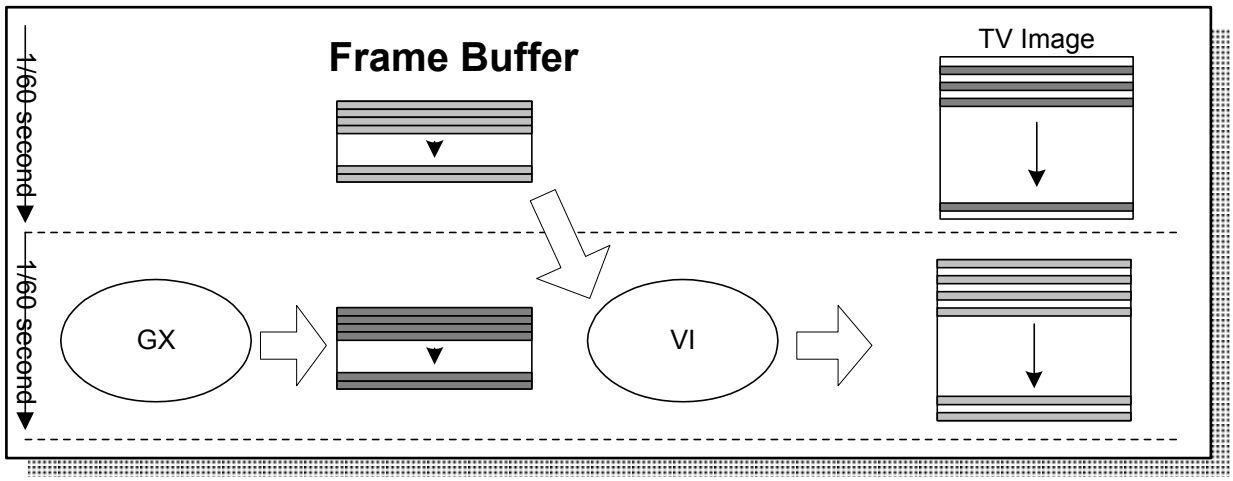
Therefore, the main memory buffers must have enough memory to accept a *y*-scaled frame buffer.

## 6.2 Field rendering

Using VI in interlaced mode puts more information onscreen for the player to view, so it is an effective way to get a higher-definition image. We can achieve this high definition in many games without increasing the fill rate by using a concept called field rendering. The field rendering technique requires that the application render even and odd fields alternately.

**Figure 9 - Field rendering**

**GX renders 640x240 at 60Hz, VI switches buffer at 60Hz and displays in interlaced mode**




If rendering takes longer than 1/60 second in NTSC or 1/50 second in PAL, it will fall behind video display. Under such circumstances, many games should be able to use the previous rendered field without suffering from any detectable visual anomaly.

**Note:** However, this previous field will be in the wrong phase (i.e., odd instead of even, or even instead of odd).

As long as rendering is fast enough, there should be no problem. Typically, rendering falls behind video display due to an increase in complexity, which may be caused by special effects in the game (e.g., explosions from weapons). This problem may solve itself because these special effects, sometimes accompanied by screen shakes (such as in a fighting game), can be very distracting. Such explosions and screen shakes may easily mask out any visual problems caused by out-of-phase field display. However, if your game doesn't have distracting special effects but the rendering still falls behind periodically, display anomalies may be noticeable.

For more information about field-based rendering, refer to "12 Video output" on page 125 of the *Graphics Library (GX)* section.

### 6.2.1 Querying next field type

For field rendering, use `VIGetNextField()` to get the data on the next field to draw.

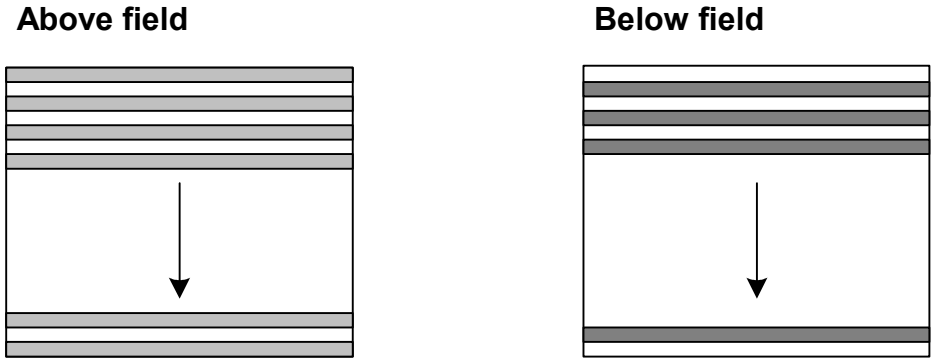
**Code 8 - VIGetNextField()**

```
#include<revolution/vi.h>

u32 VIGetNextField(void);
```

This function's return value is either `VI_FIELD_ABOVE` or `VI_FIELD_BELOW`. `VI_FIELD_ABOVE` indicates that the next field contains the odd lines (e.g., 1st line, 3rd line, ...) of the whole image to be drawn. `VI_FIELD_BELOW` means that the next field contains the even lines (e.g., 2nd line, 4th line, …) of the image to be drawn.

**Figure 10 - Above field and below field**




You can use the return value of `VIGetNextField()` as one of the arguments of `GXSetViewportJitter()` directly:

**Code 9 - GXSetViewportJitter()**

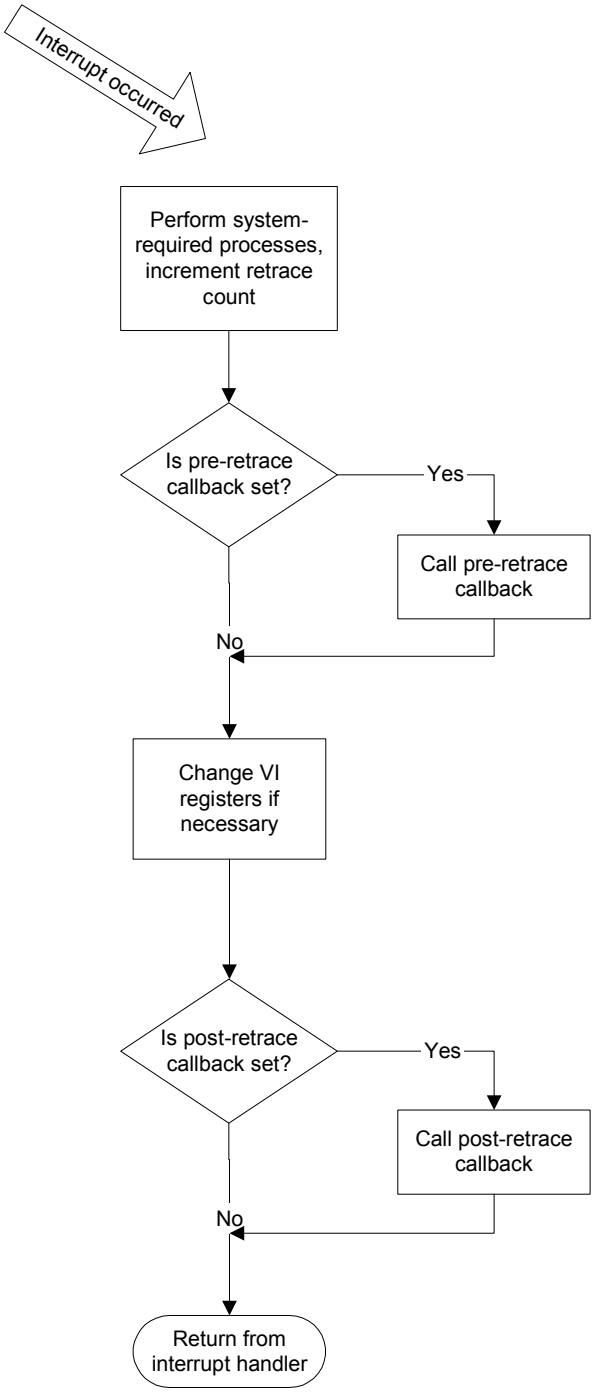
```
GXSetViewportJitter(0.0f, 0.0f, 640.0f, 240.0f, 0.0f, 1.0f, VIGetNextField());
```

## 6.3 Retrace callback

You can register two types of retrace callbacks, pre-retrace and post-retrace. Pre-retrace callbacks are called before the VI retrace interrupt handler updates the VI registers. Therefore, you can use them to change the configuration, swap frame buffers, call `VIFlush`, etc., in time for the next field. Keep in mind, however, that VI registers should remain set until a certain amount of time passes after the interrupt, so no complicated jobs should be performed in a pre-retrace callback. Save the more complicated tasks for post-retrace callbacks.

The following figure shows the relation between pre- and post-retrace callbacks:

**Figure 11 - Relationship between pre- and post-retrace callbacks**



### 6.3.1 Pre-retrace callbacks

**Code 10 - VISetPreRetraceCallback()**

```
#include<revolution/vi.h>

typedef void (*VIRetraceCallback)(u32 retraceCount);

void VISetPreRetraceCallback(VIRetraceCallback callback);
```

As noted earlier, the pre-retrace callback is suitable for performing configuration changes, swapping frame buffers, calling `VIFlush()`, etc.

The retrace count of the next field is passed to the pre-retrace callback when the latter is called.

**Note:** If you call `VIFlush()` in a pre-retrace callback, the change will take effect with the next field.

### 6.3.2 Post-retrace callback

**Code 11 - VISetPostRetraceCallback()**

```
#include<revolution/vi.h>

typedef void (*VIRetraceCallback)(u32 retraceCount);

void VISetPostRetraceCallback(VIRetraceCallback callback);
```

Post-retrace callbacks are suitable for reading controller information, updating audio sequences, invoking a thread, etc.

The retrace count of the next field is passed to the pre-retrace callback when the latter is called.

## 6.4 Pan

To be documented.

## 6.5 Get TV format

**Code 12 - VIGetTVFormat()**

```
#include<revolution/vi.h>

u32 VIGetTVFormat(void);
```

`VIGetTVFormat()` returns which TV format a game uses; return values are `VI_NTSC`, `VI_PAL`, `VI_EURGB60` or `VI_MPAL`.