1. Introduction

The purpose of these notes is to provide some additional information regarding graphics topics that may be useful for your project. In particular, it considers the following topics:

1. Double buffering and page flipping
2. Vertical retrace synchronization
3. Triple buffering
4. Colors

2. Background

Before discussing these topics, it is convenient to review some basic facts regarding computer graphics technology.

In these notes, we will consider only raster display technology. With this technology, the screen is composed of a 2-dimensional grid of points, or pixels, and each screen is drawn starting on the top left pixel, from left to right and from top to bottom. The number of times per second the screen is drawn is known as the refresh rate. Common values are between 50 and 70 Hz, to prevent flickering.

In graphics mode, the screen is also abstracted as a 2-dimensional grid of points, or pixels, each of which is characterized by its color. The color of each pixel is stored in a buffer in video RAM, the **frame buffer**, i.e. a buffer that contains the full screen image or **frame**. Thus, by changing the contents of the frame buffer, we are able to change what is displayed on the screen.

3. Double Buffering and Page Flipping

An important aspect in computer graphics is animation. This is obviously true for games, but even in other applications, animation may play an important role. Animation of an object in the screen is obtained by drawing the object in slightly different positions in consecutive frames.

If a program changes the bytes in the frame buffer containing an animated object while these bytes are being scanned for display on the screen, the image produced on the screen may show some visual artifacts that may lead to poor image quality. How bad it may be, will depend on the rate at which the objects position is changed, and the number and size of the objects.

A technique commonly used to prevent these visual artifacts is **double buffering**. The idea is to use another buffer in addition to the frame buffer. The application renders each frame first to this buffer. Once the frame has been completely rendered on that buffer, it then copies it to the frame buffer, from which it is read by the "video controller" for displaying it on the screen.

A common implementation of double buffering nowadays is **page flipping**. The key idea is to just change the value of the registers of the video controller that point to the start of the frame buffer rather than copying from one buffer to the other when the frame is completely rendered. In this case, the second buffer is also in video RAM, i.e. there are two frame buffers, the primary, or front buffer, and the secondary, or back buffer. While the contents of the primary buffer is being displayed, the application renders the next frame in the secondary buffer. When the next frame is ready, the application just swaps the two frame buffers: the secondary frame buffer becomes the new primary and is displayed on the screen, whereas the primary frame buffer becomes the secondary and is used to render the following frame. Basically, this replaces the copying of a fairly large amount of memory with the changing of the value of a couple of registers in the video controller. (The amount of memory that needs to be copied depends on the screen resolution and on how the color of each pixel is stored in the frame buffer.)

Double buffering with page flipping imposes two requirements. First the video card must have enough memory to hold the two buffers. Second, there must be a way to change the start address of the frame buffer on the video controller. Although, the amount of memory to store the entire HDTV screen (1920x1080) with

4 bytes per pixel is a bit short of 8 Mbytes, i.e. less than 1% of the memory of a video card with only 1 GB, in the early 90's many video cards had less than the almost 2 Mbytes required to store an SVGA screen (800x600) with 4 bytes per pixel.

With respect to the second requirement, the VGA specification defines a two registers that hold the address of the memory location with the first byte to display on the screen.

Start Address High Register (Offset `0x0C`, address `0x3D5` for color modes)
: These are the high-order 8 bits of the start address. The 16-bit value, from the high order and low order Start Address registers, is the first address after the vertical retrace on each screen refresh.

Start Address Low Register (Offset `0x0D`, address `0x3D5` for color modes)
: These are the low-order 8 bits of the start address.

Because these registers are 16-bits long only, they are of very limited use for high resolution modes. Even using these registers for page flipping in the low resolution VGA mode `0x13` requires its use in a non-standard way as described by David Brackeen in its 256-color VGA Programming in C tutorial

Function `0x07`, Set/Get Display Start, of the VBE 2.0 specification can, in principle, be used for implementing page flipping for higher resolution modes and in a more portable way. However, not all VBE implementations support function 0x07. Indeed, although VirtualBox (and VMware Player) emulates a video card that reports VBE 2.0 compliance, it does not support function 0x07. Therefore, with VirtualBox you will not be able to use page flipping to implement double buffering.

4. Vertical Synchronization

Although double buffering can eliminate almost all visual artifacts that appear when objects are moved around on the screen, in some cases the results may not be satisfactory. The reason is that there is still a chance for an object to be modified while it is being scanned for display on the screen. In order to reduce to 0 that probability, the frame buffer must be modified only during the vertical blank interval, i.e. the time interval between the display of the last pixel of one frame and the first pixel of the following frame.

The duration of the vertical blank interval depends on the technical characteristics of the display and of the video card and on the latter's configuration. For CRTs, a reasonable ball park number is 500 microseconds. This time is related to physical time constants: the vertical movement of the beam is controlled by a magnetic field that is created by electric current in deflection coils, to reverse this magnetic field one needs to reverse the current. The rate at which that is possible depends on the coils inductance and the voltage that is applied to these coils. In an LCD screen, however, there are no such time constants in play. Nevertheless, for compatibility reasons, it appears that the vertical synchronization interval is also used with LCD displays. On my laptop, using `xvditune` to find out the timing parameters of my X server, the vertical blank interval is about 170 microseconds, according to the information provided in this how-to on X an Modelines.

Depending on how double buffering is implemented, on the video mode used and on the characteristics of your HW (CPU clock rate, bandwidth of the bus used by the video card), this time may be sufficient to replace the current frame without any visual artifacts. Although page flipping is just a matter of changing the contents of a register, if you use the INT 0x10 VBE interface, the overhead will be much higher than if you use VBE's protected mode interface, but still 100 microseconds should be more than enough. When you use copying, you must take into account the possibility that it may be suspended upon an interrupt. Therefore, disabling interrupts while making the copy to the frame buffer may be essential to ensure that vertical synchronization works as intended.

Now, the critical issue is: how to synchronize the application with the display vertical retrace? The beginning of vertical retrace is an external event that occurs asynchronously with the application, therefore we have the two standard approaches: either via interrupts or via polling.

The VGA Way

For VBE implementations that are VGA compatible, the following bit can be polled, or used to generate interrupts:

Bit 3 of the Input Status Register 1 (address `0x3DA`, for color modes)

Vertical Retrace - A logical 0 indicates that video information is being displayed; a logical 1 indicates a vertical retrace interval. This bit can be programmed to interrupt the system processor on interrupt level 2 at the start of the vertical retrace. This is done through bits 5 and 4 of the Vertical Retrace End register.

The Vertical Retrace End register is one of the registers in the set of CRT Controller Registers which are accessed via port `0x3D5` in color modes, and has offset/index `0x11`. Bits 5 and 4 of this register are defined as follows:

Bit 5

Enable Vertical Interrupt - A logical 0 enables a vertical retrace interrupt. The vertical retrace interrupt is on IRQ2. This interrupt line may be shared so the Input Status register 0, bit 7, should be checked to find out if the VGA generated the interrupt. As with bit 4, do not change the value of the other bits in this register.

Bit 4

Clear Vertical Interrupt - A logical 0 clears a vertical retrace interrupt. At the end of the active vertical display time, a flip-flop is set in the VGA for vertical interrupt. The output of this flip-flop goes to the system board interrupt controller. An interrupt handler has to reset this flip-flop by writing a 0 to this bit, then setting the bit to 1 so that the flip-flop does not hold interrupts inactive. Do not change the other bits in this register. The register is readable so a read can be done to determine what the other bits are before the flip-flop is reset.

Finally, bit 7 of the Input Status Register 0 is defined as follows:

Bit 7 of the Input Status Register 0 (address `0x3C2`)

CRT Interrupt - A logical 1 indicates a vertical retrace interrupt is pending; a logical 0 indicates the vertical retrace interrupt is cleared.

It appears that the use of IRQ 2 for the video card has its origins on the EGA standard, when PCs had only one PIC. Later, when a second PIC was added, that IRQ line was used to chain the slave PIC to the master PIC. So, the IRQ line used, if any, will certainly be different. Actually, the vertical retrace interrupt is not implemented in all video cards that claim to be compatible to VGA. For example, a VGA Function Specification by Cyrix, dated January 1998, states that bits 4 and 5 of the Vertical Retrace End register are not implemented. May be because of that, most examples of vertical synchronization with VGA, see for example David Brackeen's tutorial, use polling rather than interrupts. The only implementation of vertical synchronization in VGA with interrupts that I found was the SVGA Linux library, but in an email one of its developers states that it does not work on all video cards at that time.

The information in this section about the bits of VGA registers is based on supposedly technical reference of one of IBM's PS/2 models. In spite of several attempts, I was not able to find any document that could be considered the specification of the VGA standard.

The VBE 2.0 Way

In principle, VBE 2.0 implementations do not really need the VGA interface. VBE function 0x07 Set/Get Display Start, sub-function 0x80, Set Display Start during Vertical Retrace, allows to swap the frame buffer in the following vertical retrace. However, this function is not supported by VirtualBox's (and VMware Player's) video card emulation.

5. Triple Buffering

It appears that many game aficionados do not like vertical synchronization. This is because with double buffering, if the program takes longer than the refresh period to generate a frame, it will have to wait until the next vertical retrace before it can start rendering the following frame. I.e., vertical synchronization may add a delay of up to the screen refresh period, pushing the frame rate down. This may affect the game's physics, and most importantly may lower responsiveness to user input.

This problem may be solved by adding a third buffer, i.e. **triple buffering**. At any time, there is one buffer, the primary, whose content is being displayed on the screen, and two back buffers, the secondary and tertiary buffers. Once the program completes rendering a frame in the secondary buffer, it can immediately start rendering the next frame on the tertiary buffer, without having to wait for the next vertical retrace. If the next vertical retrace occurs before that frame is complete, a likely event, the secondary buffer becomes the primary buffer, the tertiary buffer the secondary buffer and the previous primary the new tertiary buffer. Else (the tertiary buffer is completed before the next vertical retrace), the tertiary buffer becomes the secondary buffer and vice-versa, skipping one of the frames, and the rendering of a new frame on the new tertiary buffer may begin.

VBE 3.0 adds new sub-functions that provide support to implement triple buffering. The use of these new functions is clearly explained in Section "Using Hardware Triple Buffering", in pg. 9 of VBE 3.0 Specification.

It is not clear whether VBE 2.0 support for double buffering via page flipping can still be used to implement triple buffering. VBE 3.0 specification hints at two potential issues with VBE 2.0 sub-functions:

1. Potential incorrect page flipping with 24 bpp modes (what a 24 bpp mode is, is explained in the next section), because of the way the starting address is specified: VBE 2.0 sub-functions use (x,y) coordinates, whereas VBE 3.0 use byte offsets. This issue is independent of the use of vertical synchronization.
2. Page flipping synchronized with vertical retrace with sub-function 0x80 may be blocking. In other words, this sub-function may return only after page flipping has been performed. If so, it is useless to use a third buffer: once it returns, we may as well use the previous primary frame-buffer to render the next frame.

6. Colors

This is the last topic we will cover in these notes. As mentioned above, in graphics mode each pixel is characterized by its color. By and large, the most common color model in the computer graphics world is the RGB model, which represents each color in terms of its (additive) composition, as perceived by the human eye, of the red, green and blue (RGB) primary colors.

Thus, with the RGB model, the color of each pixel is represented by the value of each of the 3 primary color components. The value of each component ranges from 0 to a maximum value: if all components have value 0, the result is black, if all components have their maximum value, the result is the brightest white.

In computers graphics, the maximum value is the maximum unsigned integer that can be represented in the number of bits assigned to the corresponding color. Thus, the total number of colors depends on the total number of bits to represent the 3 components, and is known as the **color depth**, which is often expressed in bit per pixel, or bpp.

The most convenient, and widely used, way of storing the color of each pixel is the **packed pixel** model. In this model, the bits encoding the color of a pixel are stored in consecutive bytes rather than scattered in memory. In VBE, starting with version 2.0:

packed pixel model
: is reserved for modes that use a color look-up table, also known as palette. In this model, the frame buffer does not store the RGB components of a pixel, but rather an index to a table that maps that index to the RGB components

direct color model
: denotes modes that store the RGB components of a pixel's color directly in the frame buffer

Packed Pixel Modes

Virtually all packed pixel modes supported by VBE 2.0 use an 8 bit index, that is, the palette has 256 colors. By default, this palette uses 6 bits, which is standard in VGA, to encode each of the RGB components, thus providing an 18-bit color depth, i.e. about 262 thousand colors. However, function `0x08` of the VBE interface

can be used to switch the number of bits used to encode each RGB component from 6 to 8, thus providing a 24-bit color depth, also known as "truecolor".

Function `0x09` of the VBE interface can be used to load the palette, i.e. to program the available colors. By dynamically changing the palette an application may use more than 256 colors, however at any time there are only 256 colors available, unless the palette is changed while the screen is being displayed, which may lead to some undesirable visual effects. If the VBE implementation is VGA compatible, it is also possible to change the palette by writing to the appropriate VGA registers. How this can be done is described in David Brackeen's tutorial.

Direct Color Modes

Common direct color modes use a color depth of either 16 or 24 bits. Modes with a 16-bit color depth usually either use 5 bits per primary color, or use 5 bits for each of red and blue, and 6 bits for green. If 5 bits are used for each color, the 16th bit may either be unused or else used for transparency. Modes with 24-bit color depth use 8 bits per primary color. Many of these use 4 bytes rather than 3 bytes per pixel, thus ensuring that each pixel is double-word aligned speeding up memory access. In some of these modes, the 4th byte is not used, but in other modes, the additional byte is used for the **alpha channel** which encodes the transparency of the pixel, a zero value being a fully transparent pixel and a maximum value (`0xFF`, if 8 bits are used) an opaque pixel. To find out the layout of the RGB information used in the different modes, an application can use the fields `RedMaskSize, GreenMaskSize, BlueMaskSize, RsvdMaskSize, RedFieldPos, GreenFieldPos, BlueFieldPos and RsvdFieldPos` of the VBE Mode Info struct returned by VBE function `0x01`, Return VBE Mode Information.

Further Reading

David Brackeen's 256-Color VGA Programming in C tutorial
: As its title suggests, this is a tutorial that focus on VGA. However, it is a nice short introduction to many of the topics discussed here, with implementation details for VGA.

Andre LaMothe's Black Art of 3D Game Programming
: The first 9 chapters of this book are not 3D specific rather lay its foundation, discussing the topics addressed here and many more, that may be useful for your project. It also shows a VGA bias.

Michael Abrash's Graphics Programming Black Book Special Edition
: This appears to be more advanced than Andre LaMothe's book, with a heavy focus on performance. However, the book organization is not clear, because the title of the chapters are not always that clear, making it harder to find what we are looking for. This stems probably from the fact that this book appears to be based on journal articles written by the author. Unlike LaMothe's book, this free online version does not include pictures. As the other two references, this is still VGA biased.