

Double Buffering

From OSDev Wiki

Double-buffering is a term that is well known throughout the graphics industry. Most applications (primarily games) seldom work without it as it has many advantages over single-buffering (but also has a few disadvantages).

Difficulty level



Beginner

Contents

- 1 What Is It?
 - 1.1 The Buffers
 - 1.2 Where To Write To
 - 1.3 What Actually Is Seen
 - 1.4 Important Note
 - 1.5 Example
- 2 Advantages
- 3 Disadvantages
 - 3.1 Memory requirement
 - 3.2 Speed
 - 3.3 Tearing
 - 3.3.1 Vertical Synchronization
 - 3.3.2 Triple buffering
 - 3.3.3 Something unique?
- 4 Examples
 - 4.1 Buffer creation
 - 4.2 Double buffering

What Is It?

The Buffers

Double-buffering possibly sounds harder than it actually is. In theory, you have a buffer A and a buffer B, which are usually respectively called the front-and back-buffer. On single-buffered displays, you draw straight to video memory, so suppose video memory is located at address `0xB8000` (like console video memory is), you just start modifying values starting from this address (as you probably know) to show characters on the screen. If you would do this with double-buffering (which isn't a very good idea with console video memory, by the way), `0xB8000` would represent buffer A (the front-buffer) whilst buffer B is created and allocated by your favourite `malloc` function (or `new[]` operator).

Where To Write To

Instead of writing directly to console video memory (at `0xB8000`), you write everything to the address where Buffer B is located. The process of drawing characters is exactly the same as before, except you're now writing to another memory address. When you're done drawing, you copy the contents of the back buffer (buffer B) to the front buffer (buffer A). This process is commonly called 'swapping the

buffers', though swapping isn't interpreted as switching the contents around, but more of in 'switching the order of the buffers'. By 'swapping the buffers', the back buffer is now seen by the user and carries the same pixel values as the front buffer. The back buffer can then be freely modified again until ready to be then swapped again.

What Actually Is Seen

When using double-buffering, the user is looking at Buffer A while you're drawing to Buffer B. When using single-buffering, the user is looking at buffer A at the same time you're modifying that buffer. So that means the user doesn't see any pixels being modified at the moments he or she is looking at the screen. He or she will notice the changes as soon as you swap the buffers.

Important Note

I used console video memory as an example here, as practically every OS developer knows how to use this, but it is still not recommended to use double-buffering on a console display, as it is practically useless and wastes memory (unless you're using some kind of game loop for your OS). Double-buffering can be very useful when you're building your own GUI though. On console displays, you usually also don't need double-buffering since there are no problems with artifacts.

Example

For example, without double buffering consider a simple game rendering like the following:

```
void run()
{
    while(running)
    {
        /* update input */

        /* update game logic */

        clear_screen();
        draw_background();
        draw_levels();
        draw_picksup();
        draw_characters();
    }
}
```

Now imagine if each of those functions drew directly to the frame buffer. The video card and monitor may update just after you have drawn the background, or it may update just after you have cleared the screen. More likely, you're going to end up with a combination of effects with resulting in your screen flickering and seeing through objects.

Now imagine if each of these functions drew to a secondary buffer the size of the frame buffer. Each time you modify a pixel it won't be writing directly to the video card. Instead, after `draw_characters()` you would call:

```
memcpy(framebuffer, backbuffer, width * height * bytesperpixel);
```

so the final image in its entirety is sent to the frame buffer, not just parts of it.

Advantages

Now why would you want to use double-buffering over single buffering in e.g. your GUI, games, or video displays? Well, the main advantages of double-buffering are:

- The user does not see every pixel modification (so if the same pixel gets modified 5 times, the user will only see the last one). This helps preventing 'flickering' (or screen flashes).
- Double-buffering usually takes care of artifacts you or the user might be experiencing.
- Writing to video memory is only performed once, on the buffer swap, instead of repeatedly for every pixel of which some might be overlapped later and might be invisible.

Disadvantages

Even though double-buffering is useful, it also has negative characteristics. Each is listed with an explanation and a possible solution.

Memory requirement

You need to create a second buffer with the same size of the video memory (and for large displays such as 1280x1024x32 this can be an expensive cost).

This is more of an issue on embedded and older systems with limited memory. In the example of 1280x1024x32, this would mean your application or video driver would need to allocate an extra 5 megabytes, which isn't an expensive once off cost in a modern system that has between several hundred megabytes to several gigabytes of memory.

Also, virtually every modern desktop computer has a dedicated graphics accelerator card which has its own inbuilt memory. While older graphics cards may only have 8MB, many newer high-end models can have close to several gigabytes. This is more than sufficient to store the back buffer in, as well as being optimized for this purpose. An OS should allow the video driver to allocate and manage the double buffer since each graphics vendor will usually have their own implementation.

Note that the buffer doesn't always have to have the same size as the video memory, there are also other ways to do double-buffering, for one you could also have some sort of fast run-length encoded compression system. Then again, that would have an extra performance cost if not done efficiently.

You don't have to double buffer the entire screen. For example, if only part of the screen is constantly updating (the output of a media program or a video game) then only that section of the screen needs to be double buffered and the rest of the screen can be drawn directly to the front buffer. This method saves both memory and performance.

Speed

You need to copy the back buffer to the front buffer every time the buffer needs to be swapped (in games with a frame-rate of 30, 30 times each second). So better make sure your memcpy and other memory functions are optimized!

However, some graphics cards have the ability to specify the address in memory where the buffer used for rendering is stored. If you're redrawing the entire scene each frame (which is often the case when rendering video or 3D), you simply need to swap two pointers (the pointer to the buffer you're drawing to, and the pointer to the buffer being drawn on screen).

Tearing

Tearing (example image (<http://www.nhancer.com/help/images/VSync.jpg>)) is an artifact when the buffer is being updated while the graphics card and monitor are midway through updating the screen. The side effect is that parts of multiple frames end up being sent to the monitor at the same time. Tearing is most noticeable at animation at extremely high frame rates (100fps+) with low refresh rates (60Hz) and less noticeable vice versa, though it happens whenever the refresh rate does not match the frame rate.

Tearing can usually be ignored in general applications (office suites, web browsers, terminals) since the entire screen isn't constantly being updated. However, it still occurs, for example dragging a window around fast in a window manager that does not account for tearing (example (<http://yfrog.com/j1nouveauwindowmovep>)).

Vertical Synchronization

Vertical synchronization (http://en.wikipedia.org/wiki/Vertical_synchronization) , more commonly known as v-sync, is when the frame rate is synchronized to match the vertical refresh rate of the screen. This means the double buffer is copied to screen's buffer in a small period between frames known as the vertical blanking interval (http://en.wikipedia.org/wiki/Vertical_blanking_interval) . By updating the screen's buffer between frames, you ensure that only full frames are shown at a time.

To aid in synchronizing, virtually all video cards since the first home computers have the capability to enable an interrupt at the beginning of each vertical blanking interval (known as VBLANK).

An example of vertical synchronization is as follows:

```
/* handle VBLANK, called by the interrupt handler */
void vga_handleVBlank()
{
    if(vga_framerendered)
    {
        memcpy(vga_screen, vga_backbuffer, vga_width * vga_depth * vga_bytewidth);
        vga_framerendered = false;
    }
}

/* called when you finish drawing to the double buffer */
void vga_sync()
{
    vga_framerendered = 1;

    /* stick in a loop until the interrupt is fired */
    while(vga_framerendered);
}
```

There is no reason you HAVE to synchronize, another method could be you place a spinlock on the buffer, if you fail to acquire the lock simply skip that frame until you do. This may work fine in a video game where you redraw the frame over and over again so if you miss drawing something to the screen it will be redrawn on the following frame, but in a modern multitasking GUI environment this is unacceptable.

In a GUI environment synchronization becomes somewhat more complex. For example see below.

Unrelated, there is also a horizontal blanking interval, which is rarely utilized. The HBLANK interrupt fires when the display has finished drawing a line. This was used by some early systems to load in a new colour palettes per-line, enabling a wider range of colours on screen at once than the hardware was designed for. HBLANK synchronization is considered depreciated on modern hardware as displays operating over digital connectors such as DVI can receive the entire frame at once, rather than line by line with a blanking interval in-between. Modern video card may emulate HBLANK with DVI displays to a varying degree, mostly to remain compatible with legacy VGA software, however don't assume it'll be available or reliable in every video mode.

Triple buffering

Triple buffering (http://en.wikipedia.org/wiki/Triple_buffering) is when three or more buffers are being used, at the cost of more memory to store these buffers in.

In a real-time application (such as a game) utilizing triple buffering there are two back buffers and one front buffer. The system renders one frame to one back buffer, the next frame the other back buffer, the next frame to the original back buffer, etc, constantly interchanging between the two. When the VBLANK interrupt fires, the last frame fully rendered is copied to the front buffer (done efficiently, this can be a case of testing one value and swapping two pointers inside the interrupt handler without any locking).

In a system already utilizing double buffering, triple buffering is simple to implement. In most cases it can be added to the video driver without the underlying system requiring any modifications.

If losing a few MBs of memory is not an issue then triple buffering provides several advantages. First off, there is no need to synchronize frames, so the drawing algorithm can run as often and as fast as it can. Secondly, some real time applications *must* perform a consistent number of update cycles each second, in which case synchronizing with the refresh rate (which will cause the program to slow down to the speed of the monitor) or skipping frames (the time it takes to execute a single update will not be consistent; some will render frames, some won't) is not an option.

Something unique?

In GUI applications, where multiple programs are rendering to multiple front and back buffers each at their own speed you need to be somewhat creative.

For example, a method that will work is: each window could consist of a double buffer (or a single buffer that is updated on a redraw/paint event), as well as a double buffer for the entire screen. That way each program can draw on its back buffer and send it its own front buffer when it has finished drawing (either swapping pointers or `memcpy`) as often as it wants (wrapping a lock around the back buffer), then

when the GUI has detected the back buffer has changed, it copies the window's front buffer (locking it temporarily) into the screen's back buffer, then when the VBLANK fires the screen's back buffer is copied to the front buffer if the screen's back buffer has changed.

Ultimately, if memory was not an issue then the best system would be to triple buffer each window and the entire desktop, allowing every window, the desktop manager, and the video card to update at it's own speed. In some GUIs each window is allowed to draw directly to the desktop's back buffer. For example in early versions of Windows programs drew directly to the desktop's back buffer, this meant that all windows overlapping that window also had to be redrawn. The disadvantage of this is that if an application doesn't respond to a redraw request straight away then you can get a visual artefact (example (<http://i.msdn.microsoft.com/dynimg/IC405533.png>)).

Each window manager usually has its own method. But like mentioned above, tearing is a relatively minor issue in most general GUI programs that can often be overlooked, since you will be dealing with text and images that won't be updating. Therefore, if the front buffer is in the middle of outputting to the monitor while copying the next frame to it, you might notice a slight flicker that lasts for all but 1 monitor refresh (which is no less than 1/60th of a second on most monitors).

Examples

Articles with all theory are boring, so here are some examples of how to use double-buffering.

Buffer creation

```
/* Supposing the display is 800x600 with 32 bpp (meaning 32/8 = 4 bytes per pixel) */
unsigned char *BackBuffer = ((unsigned char *) (malloc(800 * 600 * 4)));
```

This implementation absorbs the same amount of memory the real video memory does, in this case, $800 * 600 * 4 = 1920000$ bytes = about 1,83 MB. While using single-buffering needs 1,83 MB of RAM with our video resolution (only the display itself), double-buffering would require $2 * 1,83$ MB = about 3,66 MB. The higher the resolution, the more memory is required. There are of course implementations that can use up less than that with special techniques, but for some OS developers, high resolutions, and especially with double-buffering, are expensive features.

If 3,66MB does not seem like much then if you imagine a video playing program that implements triple buffering running on a modern $1920 \times 1080 @ 32\text{bpp}$ LCD display, then the total amount of video memory to store the 3 buffers would be 24,47MB. Fortunately, as display resolution increased so does memory, so using 25MB of memory on a modern computer to play a video may be acceptable.

Double buffering

```
uint8_t * VidMem;
uint8_t * BackBuffer;

unsigned short ScrW, ScrH;
unsigned char Bpp, PixelStride;
int Pitch;
```

```

/*
 * Initializes video, creates a back buffer, changes video modes.
 * Remember that you need some kind of memory allocation!
 */
void InitVideo(unsigned short ScreenWidth, unsigned short ScreenHeight, unsigned short BitsPerPixel)
{
    /* Convert bits per pixel into bytes per pixel. Take care of 15-bit pixels!
     * PixelStride = (BitsPerPixel | 7) >> 3;
     * The pitch is the amount of bytes between the start of each row
     * This should work for the basic 16 and 32 bpp modes (but not 24bpp)
     */
    PixelStride = (BitsPerPixel | 7) >> 3;
    Pitch = ScreenWidth * PixelStride;

    /* Warning: 0xEEEEEEEE serves as an example, you should fill in the correct address here
     * VidMem = ((byte *) 0xEEEEEEEE);
     * BackBuffer = ((byte *) (malloc(ScreenHeight * Pitch)));

    ScrW = ScreenWidth;
    ScrH = ScreenHeight;
    Bpp = BitsPerPixel;

    /* Switch resolutions if needed... */
    /* Do some more stuff... */
}

/*
 * Draws a pixel onto the backbuffer.
 */
void SetPixel(unsigned short X, unsigned short Y, unsigned short Colour)
{
    int offset = X * PixelStride + Y * Pitch;

    /* Put a pixel onto the back buffer here. */
    /* Remember to write to the BACK buffer instead of the FRONT buffer! */
    /* Take care of writing exactly PixelStride bytes as well */
}

/*
 * Swaps the back and front buffer.
 * Most commonly done by simply copying the back buffer to the front buffer
 */
void SwapBuffers()
{
    /* Copy the contents of the back buffer to the front buffer. */
    memcpy(VidMem, BackBuffer, ScrH * Pitch);
}

/*
 * An example of how to use these functions.
 */
void ProgramLoop()

```

```

{
    while(Program_Is_Running)
    {
        /* Handle events, update window coordinates and other thi


        /*
        * You should probably implement some sort of FillScreen
        * or something that clears the back buffer before drawin
        * back buffer to 0 (black) will suffice for a basic syst
        */
        memset(Backbuffer, 0, ScrH * Pitch);

        /* Draw everything: a GUI, windows, other things. This ex
        SetPixel(50, 50, 0xFFFFFFFF);
        SetPixel(25, 25, 0xFFFFFFFF);

        /* When done drawing, swap the buffers and let the user s
        SwapBuffers();

    }
}

```



I believe this example is mostly clear. As there are too many different ways of changing video modes or putting pixels, I'm going to let you fill that in yourself. If you want more information, you can go to the GUI or the Drawing In Protected Mode pages.

Note also that the above code might be very 'irregular'. As you know, there is no limit on the drawing, so it is possibly that the first draw takes 30 milliseconds, while the second draw takes 40 milliseconds or the first few seconds, you have a frame-rate of 100 frames per second whilst the next couple of seconds, you have a frame-rate of only 20 frames per second. Because of this, drawing times are usually capped. In the most common GUIs (such as the one in Windows), this is not really of any importance, since only parts of the screen are redrawn if they are 'invalidated' (usually referred to as 'Invalidated rectangles'), but in games, the screen is being constantly redrawn, usually up to 60 times per second. Today's APIs usually clamp this value to the screen refresh rate, this is why they usually have values such as '60' or '75'. Normally, a frame-rate of 25 to 30 suffices to have a 'smooth' display.

I've used some sort of loop here (it looks like a game loop), but remember that most GUIs on most OSes do not work with game loops, they usually work with some sort of invalidation system. Meaning a control is redrawn only if it is invalidated, which happens on occasions such as moving or resizing the control.

Retrieved from "http://wiki.osdev.org/index.php?title=Double_Buffering&oldid=17612"

Categories: Level 1 Tutorials | Video

- This page was last modified on 16 February 2015, at 10:40.
- This page has been accessed 42,227 times.