COMP 2659 Course Project – Stage 6: Double Buffered Graphics

Given:	Friday, March 6, 2015
Target Completion Date:	Friday, March 13, 2015

Overview

In this stage you will synchronize your game's frame rate (i.e. the rate at which successive frames of animation are produced) with the physical refresh rate of the video hardware and monitor. The goal is to achieve animated graphics that are free of visual distortions like flickering and tearing using a technique called "double buffering".

Background: The Design of Video Games, Part 3

In computer animation, flickering and tearing occur when the frame rate is not synchronized with the refresh rate of the monitor. In other words, the video hardware is scanning the frame buffer at the same time as the frame buffer contents are being modified by the rendering routine.

To avoid this problem, it is necessary that these two processes occur at the same frequency (or that one frequency is a multiple of the other), and that they are in phase (i.e. that one doesn't start partway through the other).

One possible solution is to perform all the rendering for frames between screen refreshes, i.e. during the vertical blank intervals of a CRT monitor. This works well when the rendering process is very fast and can be guaranteed not to exceed the duration of the vertical blank (after which time the video hardware will begin scanning the frame buffer anew). However, this guarantee can be problematic in practice.

A better solution is to use the "double buffering" technique. A static image is held in the frame buffer while the next frame is rendered to a second "back buffer", off screen. Once this frame is complete, the "pages" are "flipped" at the next vertical blank. The former back buffer becomes the front buffer scanned by the video hardware during the next refresh, whereas the former front buffer can be used as the back buffer for rendering the next frame of animation.

It is important to note that the contents of the back buffer are *not copied* to the front buffer[1]. Instead, the video hardware register which contains the frame buffer start address is simply updated. The video hardware obtains the frame buffer start address from this video base register at each vertical blank, so that it is ready to start its next scan of the frame buffer at the correct location.

The use of double buffering does not guarantee smooth graphics. For example, if the rendering process was slow, at about 10 fps, the Atari ST's video hardware would perform about 7 screen refreshes for each page flip. Therefore, the animation would appear jumpy, albeit solid. In this scenario, effort would then need to be invested in optimizing the rendering process so that the animation was closer to (or ideally equal to) 70 fps.

[1] Copying an entire frame of memory can be as slow as rendering a frame, or slower if the rendering is optimized.

Requirement: Double Buffering

Double buffering must be implemented correctly, with absolutely no flickering or tearing. In this phase, all changes must be made to the main game module only.

First, it is necessary to allocate a 32,000 byte buffer. For hardware reasons that will be discussed later, frame buffers on the Atari ST must be 256-byte aligned (i.e. the start address must be a multiple of 256_{10})[2]. Initially, this buffer will serve as the back buffer (until the first flip).

Next, research the `Setscreen` system call (TOS XBIOS system call #5). This is the TOS routine which must be used to alter the frame buffer start address: it updates the video base register consulted by the video hardware at each vertical blank. Your main game loop must call `Setscreen` to schedule the flipping of the current front and back buffers each time the game program has finished rendering the next frame to the back buffer.

Common Problems

Stack smashing: the segment of memory reserved by TOS for the program's stack is not very large, and is less than 32,000 bytes wide. Therefore, the memory for the second frame buffer must *not* be allocated on the stack (i.e. as a local variable). This would cause the stack to overflow into the code segment. When the program began plotting to that frame buffer, it would corrupt existing code, resulting in odd behaviour and spurious crashes.

Incorrect parameters: In TOS, `Setscreen` can be used for more than one purpose. It takes three parameters, but only one of them (the physical frame buffer start address) is relevant to this project. The others should be passed as -1 to indicate that they are not being changed.

Premature rendering: `Setscreen` does *not* flip the pages immediately. When called, it updates the video base register so that the flip will happen at the next vertical blank, and then it returns immediately. Your code must therefore wait before starting to render the next frame. How long? Until after the next vertical blank has occurred[3]. If you forget this important detail and start rendering the next frame too soon, into what you *think* is the back buffer, you will still see flickering or tearing. This is because the buffer you think is the back buffer is really still the frame buffer, at least for a brief period of time.

Crash-on-quit: upon termination, the program must restore the original frame buffer. If not, 50% of the time, the program-allocated buffer will be at the front when TOS regains control. This region of memory is not where it expects the frame buffer to be, and strange behaviour and spurious crashes may result.

A Note on Achieving Smooth Animation at a High Frame Rate

The easiest way to implement double buffering is to erase the *entire* back buffer, and to fully render each frame, as described in stage 4 without 4b. However, since this is the least time-efficient solution, the game will very likely not achieve a high enough frame rate. In other words, the animation will be flicker and tear-free, but may be "jumpy". It is usually faster (although more complicated) to only re-plot the portions of the buffer which have changed. The problem when double buffering, though, is that the back buffer to be updated contains the view of the world from *two* frames ago.

[2] Hint: allocate a region of memory slightly larger than 32,000 bytes, and then find an appropriate buffer start address within that range.
[3] Hint: it's easy to check this using the 70Hz "vertical blank counter", i.e. the TOS-maintained longword at address 0x462. This counter is incremented at each vertical blank.

One solution is as follows: right before rendering each frame, record a “snapshot” of the relevant parts of the model. The frame of animation in each buffer is paired with its corresponding snapshot, even though the model continues to evolve. When it comes time to render the next frame to the current back buffer, the current state of the world can be compared to the earlier snapshot to determine which portions of the buffer need to be re-plotted.