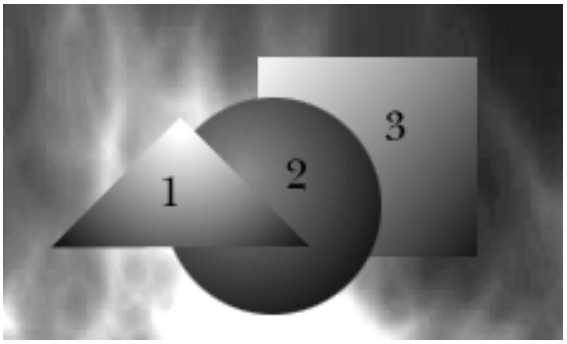
ZBuffers

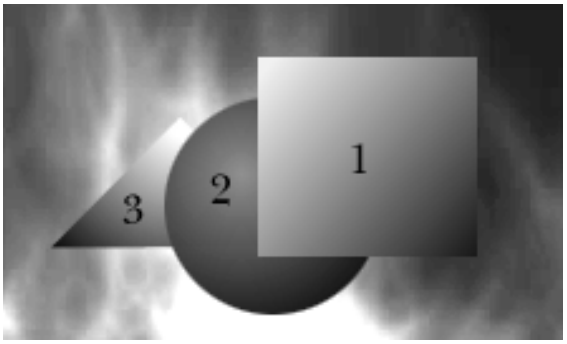
Intro

Z buffers....oh yes...Do you even realize how close they are to L buffers? Closer than you think I bet! Ok Ok, I have no idea what an L buffer is or even if they exist, but here's a pretty good explanation to what a Z buffer is! *A Z buffer is an array of a given data type that represents depth values.* Why do we need to store data on depth? Let me explain. A graphics programmer can do quite a bit of really nice programming without ever noticing the Z factor of the items being blitted to the screen. Imagine giving your program another dimension! You've been using only 2 dimensions thus far (x,y) and come up with some really cool demos. Adding the third dimension (z) gives you a lot MORE programming power! This idea is imbedded into the 3d transformations tutorial (coming soon). If we give each pixel a certain depth factor, we can now move items around on the screen and not have to worry about objects being in front of objects they are supposed to be behind and vise-versa. This is called depth-sorting. Without the Z factor, 3d games would be impossible to program. Imagine Quake II without 3 dimensional graphics :) Now THAT is scary!

How it works



A Z buffer is usually the same resolution as your screen. When it comes time to blit the pixels onto the screen or offscreen buffer, each pixel's depth value is compared to the current value at that pixel location in the Z buffer. If the depth value of the pixel it wants to blit is closer than the z value already in the Z buffer, then replace the depth value in the Z buffer and blit the pixel. If the pixel we want to blit is farther away than the depth value in the Z buffer , just skip it and move on.



What makes this approach so powerful, is that there aren't any special exceptions that need to be taken into account. There are other ways of depth sorting that take up less time and space, but there are always going to be special cases. I think everyone should start their depth sorting routines with Z buffers because they will always work and are easy to understand AND use! After getting some experience with depth sorting, you can develop a faster, more efficient means of sorting the objects that need to be blitted. As you can see to the right, the order in which we blit objects DOES matter. Each object is numbered from the closest to the farthest. As you will read below, we could probably go without a Z buffer for a scene like this.

When you DON'T need a Z Buffer

At this point, if you are just learning about Z buffers, you may be a little overzealous and want to use them in all of your programs. WAIT! You might not even need one! A lot of 2d games simply give all of their sprites and objects a Z depth value for each object. Each pixel within that object will obviously (i hope) have the same depth value. If that is the case, then we just need to take all the objects that need to be drawn, sort them in reverse according to their depth values (blit farthest first) and then draw away. This way the farthest objects will be drawn first, then overwritten by objects that are closer, just as it should be!

When you DO need a Z Buffer

This main point isn't as big as when NOT to use a Z buffer. I say this because if you try to go without one and fail, then you'll probably try a Z buffer and succeed! BUT If you don't need one, you'll most likely be able to use a Z buffer, but alas it will be at the wrong time and you'll lose performance :(The most obvious times to use a Z buffer is when unordered 3d graphics are being blitted to the screen. I say unordered because if you use a partial BSP tree to order the objects into the right drawing sequence, then, as long as you follow that order, you won't even need a Z buffer! So if you don't order your objects ahead of time, certainly use one with 3d graphics. What it all boils down to is this: Make sure that you are clear on how the graphics is being converted from data to screen output. With this in mind, consider using BSP trees or giving an entire object a Z extent. If these methods don't work and produce problems, use a Z Buffer! I'm sure there are many other ways of attaining correct screen output, but these are some of the most popular and some of the easiest.

NOTE: One downside to using a Z buffer is that you have to set it to all 0's after each screen blit. That way you won't be comparing z extents of pixels that have moved since the last screen update. One way to get around this is to use a Clear Reduction Algorithm which can let you get away with not clearing the Z buffer for 32 screenfills! This method won't be explained here because there is an easier way!

A Sign Based Z Buffer

Here is a solution that let's you get away with never clearing the ZBuffer! One prerequisite to the following method is that the entire ZBuffer must be filled every cycle with a positive number. This isn't that big of a deal since a majority of games fill the entire screen no matter what. Even when the playing view is smaller than fullscreen, the background can have a Z extent. Ok, what if we alternated positive and negative numbers in our ZBuffer. If we were filling in positive numbers, we would be comparing against negative numbers from the last frame, so no changes would need to be made. If we are going to start filling in our negative numbers, we have to test to see if our test pixel is less than the one in the ZBuffer. That's it! This alternating is a slick trick so we **never** have to clear our ZBuffer!

Contact Information

I just wanted to mention that everything here is copyrighted, feel free to distribute this document to anyone you want, just don't modify it! You can get a hold of me through my website or direct email.

Email : deltener@mindtremors.com
Webpage : http://www.inversereality.org