Image Dithering: Eleven Algorithms and Source Code

Dec 28, 2012 • by Tanner Helland

Dithering: An Overview

Today's graphics programming topic - dithering - is one I receive a lot of emails about, which some may find surprising. You might think that dithering is something programmers shouldn't have to deal with in 2012. Doesn't dithering belong in the annals of technology history, a relic of times when "16 million color displays" were something programmers and users could only dream of? In an age when cheap mobile phones operate in full 32bpp glory, why am I writing an article about dithering?

Actually, dithering is still a surprisingly applicable technique, not just for practical reasons (such as preparing a full-color image for output on a non-color printer), but for artistic reasons as well. Dithering also has applications in web design, where it is a useful technique for reducing images with high color counts to lower color counts, reducing file size (and bandwidth) without harming quality. It also has uses when reducing 48 or 64bpp RAW-format digital photos to 24bpp RGB for editing.

And these are just image dithering uses - dithering still has extremely crucial roles to play in audio, but I'm afraid I won't be discussing audio dithering here. Just image dithering.

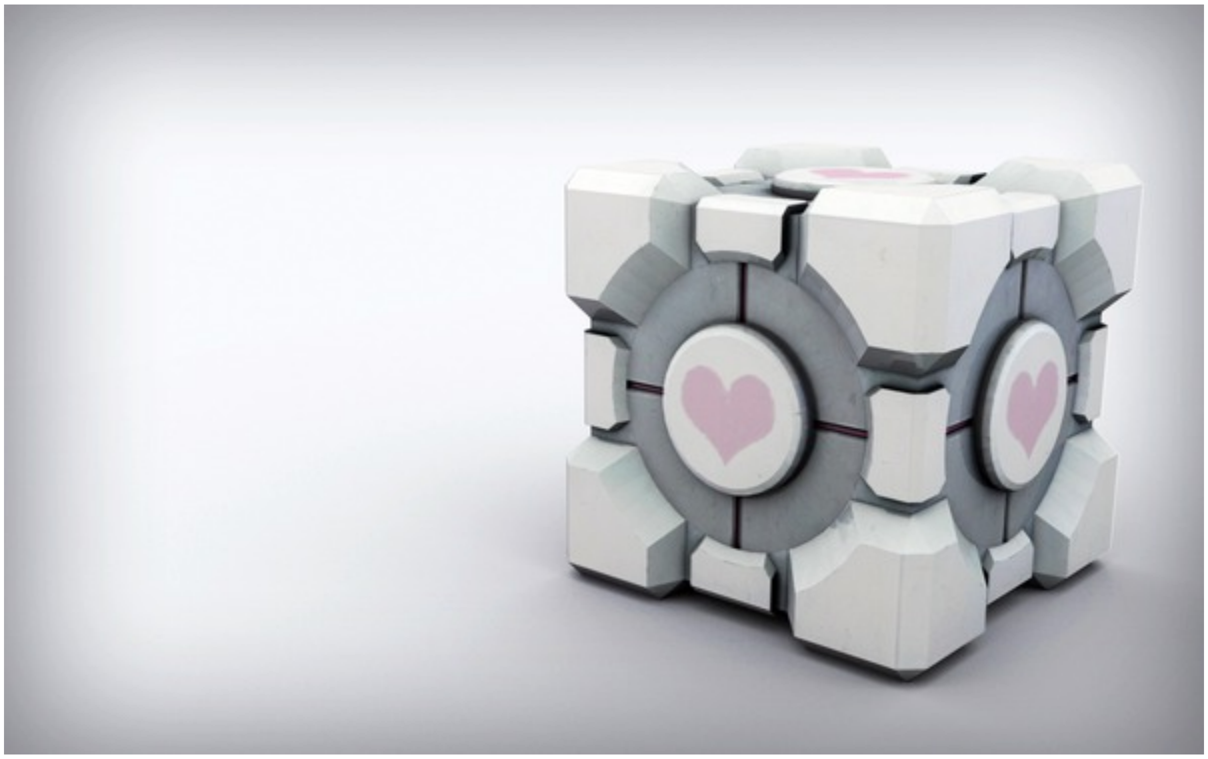
In this article, I'm going to focus on three things:

- a basic discussion of how image dithering works
- eleven specific two-dimensional dithering formulas, including famous ones like "Floyd-Steinberg"
- how to write a general-purpose dithering engine

Update 11 June 2016: some of the sample images in this article have been updated to better reflect the various dithering algorithms. Thank you to commenters who noted problems with the previous images!

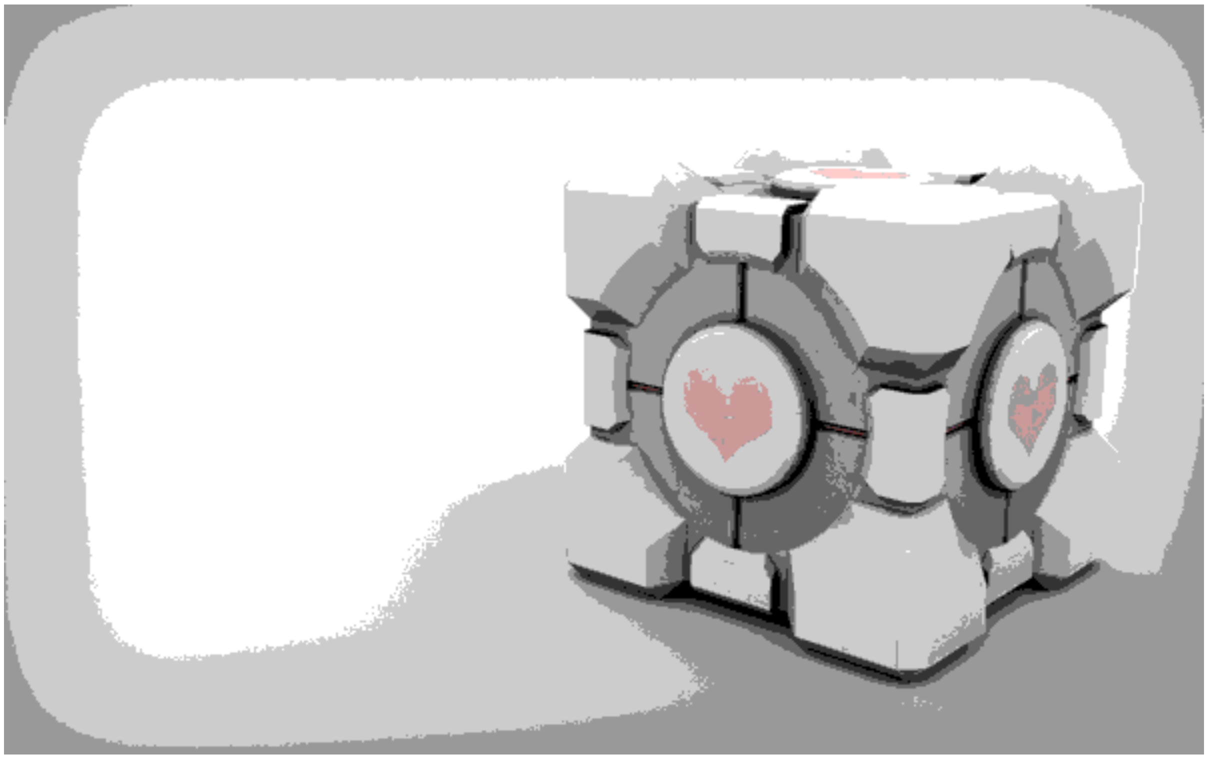
Dithering: Some Examples

Consider the following full-color image, a wallpaper of the famous "companion cube" from *Portal*:



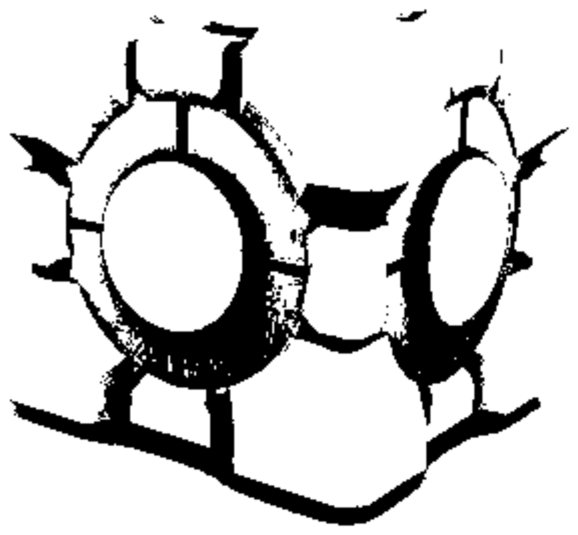
This will be our demonstration image for this article. I chose it because it has a nice mixture of soft gradients and hard edges.

On a modern LCD or LED screen - be it your computer monitor, smartphone, or TV - this full-color image can be displayed without any problems. But consider an older PC, one that only supports a limited palette. If we attempt to display the image on such a PC, it might look something like this:



This is the same image as above, but restricted to a websafe palette.

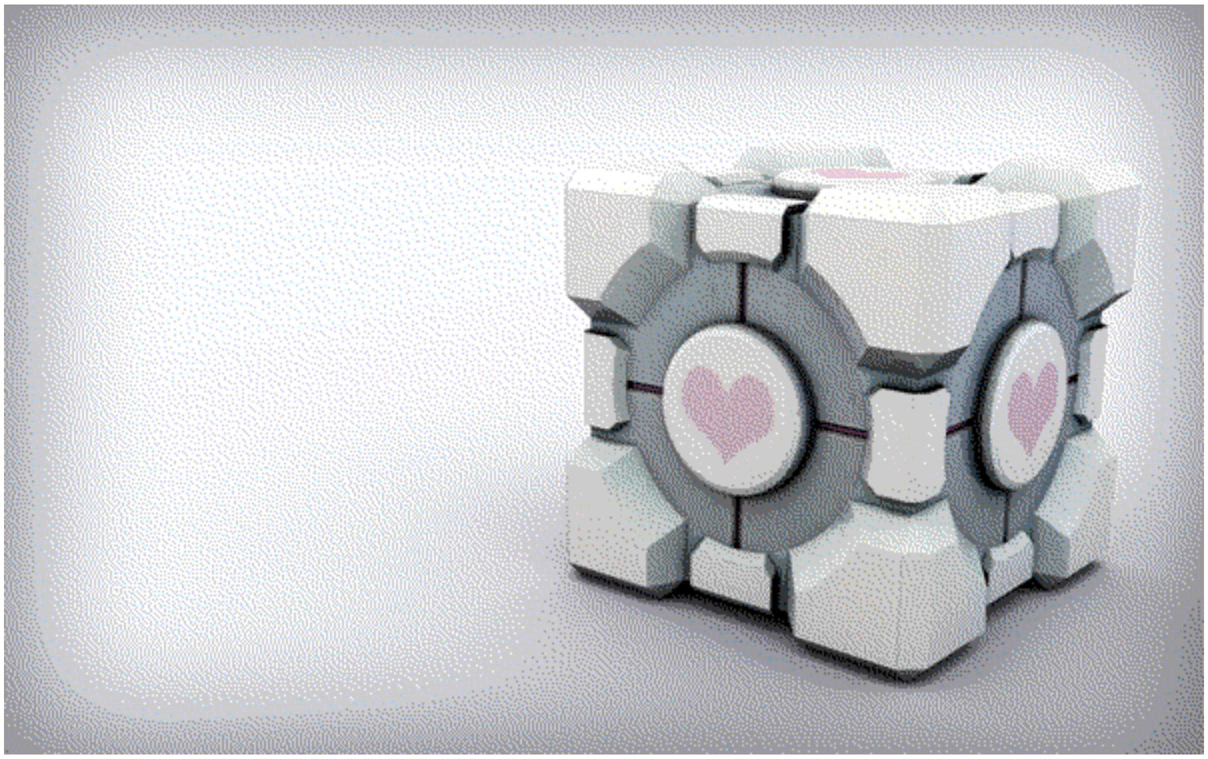
Pretty nasty, isn't it? Consider an even more dramatic example, where we want to print the cube image on a black-and-white printer. Then we're left with something like this:



At this point, the image is barely recognizable.

Problems arise any time an image is displayed on a device that supports less colors than the image contains. Subtle gradients in the original image may be replaced with blobs of uniform color, and depending on the restrictions of the device, the original image may become unrecognizable.

Dithering is an attempt to solve this problem. Dithering works by approximating unavailable colors with available colors, by mixing and matching available colors in a way that mimicks unavailable ones. As an example, here is the cube image once again reduced to the colors of a theoretical old PC - only this time, dithering has been applied:



A big improvement over the non-dithered version!

If you look closely, you can see that this image uses the same colors as its non-dithered counterpart - but those few colors are arranged in a way that makes it *seem* like many more colors are present.

As another example, here is a black-and-white version of the image with similar dithering applied: