Imaging in the GPU

Steve Marschner
Cornell University
CS 569 Spring 2008, 26 February

"Digital imaging" is a fashionable word to describe image processing as applied to applications that used to use film. Digital photography is imaging; scanners and printers are digital imaging devices; X-rays and CT scans are medical imaging. In this lecture I want to cover two things: GPUs in the service of digital imaging, or how we can use the GPU's arithmetic power to process images fast, and digital imaging in the service of interactive graphics, or how ideas that are second nature in digital imaging can make interactive graphics look better.

Imaging basics

You could almost summarize imaging as that set of topics pertaining to the meaning of the values we store in pixels. A number stored in the framebuffer and being sent to the display has a physical interpretation in terms of the light that will be emitted from the corresponding display pixel; a number stored in a texture map that modulates the diffuse component of a material has a physical interpretation in terms of the reflectance of the material being simulated.

The question of pixel value interpretation is involved with the question of number format. There's a dizzying variety of formats available, with vastly different amounts of precision (from 2 bits/pixel to 32 bits/pixel). They fall into two categories: *fixed-point* numbers and *floating-point* numbers. Sometimes these are called low- and high-dynamic-range formats.

The fixed point formats represent numbers between 0 and 1 with uniform precision. The numerical interpretation is that the value c represents the number $c / (2^N - 1)$. Note that this is different from the usual interpretation of numbers as fixed-point fractions, which would say that the value c represents the number $c / 2^N$. This distinction is important for smaller numbers of bits.

The floating point formats represent numbers over a much wider range, with approximately uniform *relative* precision. A 16-bit float and a 16-bit fixed-point number have the same number of distinct possible values; it is the assignment of these values to interpretations as real numbers that's different.

Related to the idea of pixel storage formats is the idea of dynamic range. The usual definition of dynamic range is "the ratio of the brightest thing in the scene to the darkest thing," or if one is talking about a capture device, "the ratio of the saturation level to the noise floor." To represent an HDR image, we need to store it in such a way that we have useful precision over a wide range of intensities. For a displayed image to reliably look good you need precision of about 1 part in 50, since the eye can detect a difference in brightness of about 2%. For images that will undergo any processing before being displayed you need considerable extra precision to avoid artifacts.

(Note that here I'm using the notion of "looks OK" that is traditional in imaging applications: complete absence of any visible artifacts of the image coding. Traditionally the notion of "looks OK" in interactive graphics has been the absence of artifacts so severe that they become the most

visible thing on the screen. But this is changing—we no longer have to accept bad image quality as the cost of interactivity.)

This 2% precision needs to be maintained over a sufficient range of intensities. For images that are just going to be displayed on the screen, suitable means a wide enough range to cover what we can display. Your monitor specs may say that this range is 300:1, but in practice you're lucky if it's 100:1. For images that are used in computation, though, the story is very different. An environment map, for instance, might be reflected in a metal surface, copying its values directly, or it might be reflected from a water surface, scaling it down by a factor of 25. This means that you need a range 25 times larger than the range of the display, and other situations can lead to the need for even larger ranges. Another example is an environment map that will be used for illumination, or a rendered image with bright reflections that will have glare applied.

The "dynamic range" of a pixel storage format depends on the convention for interpreting the *numbers* as *colors* (or intensities). Asking the dynamic range of a fixed-point number format is meaningless because they can all represent the number zero, and if someone tells you their dynamic range is "65,535:0" then treat anything else they say with suspicion.

The important piece that's left is the relationship between numbers stored in the image and intensities. There are two basic approaches: linear and gamma-corrected. In the linear interpretation, pixel value k represents the intensity $kI_{\max}$ where $I_{\max}$ is the maximum displayable intensity. In a gamma-corrected image, pixel value k represents intensity $k^{\gamma}I_{\max}$. The number γ needs to be specified, and it's usually equal to 2.2. It makes an enormous difference whether an image is gamma corrected or not.

An interesting computation is the maximum dynamic range that can be encoded with a given number of bits while maintaining good enough precision. Constant relative precision is obtained by spacing the logarithms of the numbers uniformly. For back-of-the-envelope calculations the approximation $\log x = x - 1$ is good for increments of 1–2%. So for 2% relative precision we need to space the quantization levels 0.02 units apart in (natural) log space. If we have 255 steps we can cover 5.1 natural-log units, which is 164:1. If we have 65,535 steps we can cover over 1300 log units, which is absurdly large. At 0.1% precision we can cover 65.5 natural-log units, which is more than 10^{28}:1.

So 8 bits per channel is barely enough represent a band-free image for display on the screen, but we can clearly store a high dynamic range in 16 bits, with the right kind of quantization. Let's look at some real quantization schemes.

If we use linear quantization with 16 bits, we have good (2%) precision from 50 to 65,535, leading to a usable dynamic range of about 1300:1. This makes 16 bit linear safe for moderate dynamic ranges, with some headroom.

If we use 8-bit linear quantization, we have OK (5%) precision from 20 to 255, or 13:1. With gamma-2 quantization for 8 bits we have OK precision from 40 to 255, which is 40:1, good enough to describe an image that will be printed, or viewed on a monitor in a non-darkened room. The performance is actually better in the real world where there is some viewing flare. A small amount of flare (0.5% of $I_{\max}$) brings the dynamic range for OK precision up to about 150:1. This is OK for LDR images.

16-bit floating-point quantization maintains high precision (1 part in 1024) over the range from 2^{-14} to 2^{15}, which is about 500,000:1 and good precision (2%) over about 2,000,000:1.

Floating point numbers are a good practical compromise between uniform relative precision and efficient computation. Since shader computations are already FP, it makes sense to adopt FP whenever we want nonuniform quantization over a wide DR.

So the (fully interpreted) datatypes most commonly used are:

1. 8 bit gamma-corrected (sRGB) (LDR)
2. 16 bit linear (LDR with headroom)
3. 16 bit floating point (HDR)
4. 23 bit floating point (way too much precision for images!)

Just slinging pixels without paying attention to what they mean is no longer the state of the art, even in interactive applications!

Imaging operations on the GPU

Pixelwise operations

Many of the most common adjustments to images are simple functions applied independently to the pixels. That is, output pixel (i, j) is determined solely by the value of input pixel (i, j). This makes pixelwise operations wonderful candidates for a parallel computer (such as the fragment processing stage of a GPU). Some examples:

1. gamma correction
2. tone adjustment (contrast enhancement; Levels and Curves in Photoshop)
3. color correction (both color balancing and color space transformation)
4. image compositing (layering and blending)
5. tonal special effects (sepia toning, "cross processing," and so forth)

In the fixed-function pipeline there are pixel transfer modes for doing pixelwise operations, but they can also be done, and more flexibly, using fragment shaders. Since we already know how to use those, I'll focus on that approach.

We want to write a fragment program and have it executed on every pixel of the image. There are (at least) two ways to get the image data into the fragment program. One is to load the image into a texture, draw a big rectangle on the screen, and access the image using texture lookups when you process the resulting fragments. If you are careful with the screen and texture coordinates, and use nearest-neighbor texture access, you can arrange for a one-to-one mapping between pixels in your image and fragments processed by the fragment program. Another (less roundabout but not always more effective) way to turn your image into fragments is to use *glDrawPixels*, part

of the less-used "raster operations" side of OpenGL. In this way your image is directly converted into a bunch of fragments (one per pixel unless you ask for something different), and the pixel values come into the fragment shader via the fragment color.

Writing the fragment program for something simple like gamma correction or a matrix-based color space transformation is very simple. In particular, it's very easy to apply a color matrix in a fragment shader, because all the linear algebra machinery is already there for processing vectors.

If you want to apply pixelwise transformations that are complex enough that they start to get slow as pixel shaders (or you just don't feel like developing pixel shader implementations of all your image operations), you can build and use lookup tables instead. The GPU is the ideal processor for this, because the machinery of texture mapping provides a built-in super-fast table interpolation engine.

For table-driven transformations the distinction between channel-independent operations and more general pixelwise operations is important. If the three channels get mapped independently, you can implement the operation using three lookups into a 1D texture (and if your images are not more than 10 bits, you can comfortably enumerate all the pixel values in the table, avoiding any approximation error). A nifty trick is to create the texture by loading a 1D grayscale ramp in Photoshop and applying whatever operation you wanted to apply to the image.

For non-channel-independent operations, you can use a 3D texture for your table lookup. You can no longer plan on enumerating all the possible values, though. But usually the function you want to implement is smooth enough to be well approximated by a trilinear interpolant on a very small grid, less than 30 samples in each direction. The 3D trilinear table is widely used in color transformations for nonlinear devices like printers, and the GPU is a very fast way to implement it.

Spatial operations

Although many, and perhaps most, of the everyday image-manipulation tools are pixelwise operations, some are spatial, in the sense that they involve a pixel and its neighbors. For instance, blurring, sharpening, edge detection, and flare simulation all involve spatial calculations. These are a bit more expensive to compute, both on the CPU and the GPU, but they are still amenable to efficient GPU implementation.

The canonical spatial operation, which encompasses most methods for the applications I just mentioned, is *linear filtering* or *convolution filtering*. I assume you remember from 465 or your other graphics or signal processing class what this means. Like pixelwise transfer operations, convolution is supported in a limited way by the fixed-function pipeline, but we can also implement it using fragment shaders.

For convolution (and other spatial operations), since the computation for each pixel needs to access several pixels, the image needs to come in as a texture. Then several texture reads are all that's required to get access to the pixel values. Again we need to take care with the texture coordinates to ensure that the pixels are sampled one-for-one.

Separable filters can be implemented efficiently in two passes, just as in software, and can be pretty big thereby.

Other (nonlinear) spatial filtering operations can be coded up in fragment shaders as well.

Note that there exists a mechanism for computing the histogram of an image, a gather-type operation that is hard to do with the regular pipeline. This can be quite useful for vision algorithms.

Imaging for interactive graphics

Gamma correction

You want to pay attention to gamma correction. Sometimes when we're worrying about other aspects of the rendering process, we make some tacit assumptions about what the numbers mean, but usually these assumptions are very wrong!

Textures used for reflectance: these are very likely gamma-quantized. You should un-gamma them before you use them for lighting calculations.

Textures for other shading parameters: depends on the source. Probably linear.

The framebuffer: it will go to the monitor unaltered by default. Therefore it's interpreted as gamma-quantized, and you should gamma correct your values before writing them.

Environment maps: if you read these in an FP format they are normally linear already.

Note that the practice of ignoring gamma correction leads to canceling errors that make you think everything is OK, but it's really not!

Reading nonlinearly quantized values from textures and correcting them in the shader is fine, if expensive. But it doesn't prevent the system from doing linear interpolation when it shouldn't. MIP maps!

HDR imaging

For some operations, you need the extra precision of floating-point pixel values stored in textures.

Tone mapping

Your shaders are probably producing HDR output, even if you're not thinking about it that way. You need to tone-map this output, and this is being done pretty crudely if you're not paying attention.

Further reading

Here is what I was reading as I was preparing these notes.

GPU Gems 2, Chapter 24 by Jeremy Selan on "Using Lookup Tables to Accelerate Color Transformations."

GPU Gems 3, Chapter 24 by Larry Gritz and Eugene d'Eon on "The Importance of Being Linear."

Rost, *OpenGL Shading Language*, Chapter 19 on "Shaders for Imaging."

The OpenGL 2.1 Specification

The spec for the ARB_texture_float extension
(http://www.opengl.org/registry/specs/ARB/texture_float.txt)