CS 314: Signal Processing, Collision Detection

Robert Bridson

November 6, 2008

1 Using your Ears

Last time we discussed two difficulties in texture mapping: dealing with magnification (by using smooth interpolation) and with minification (by using some sort of averaging). The arguments were pretty light on technical grounds, appealing to subjective opinion about what looks good and what looks bad, and to a fairly hazy idea of what the human visual system does to justify this a little. Unfortunately, while experts certainly do have much more extensive knowledge about human vision, we are apparently still a fair distance away from physiological and psychological research informing mathematical techniques in computer graphics: we just don't know really how to quantify what a good image is.

However, there is a related domain where we are on firmer mathematical footing: **signal processing**, for audio in particular. Of course our ears work a bit differently than our eyes, but they do share some sensitivities, and in fact are even more attuned to the issues of "edges" and interpolation and related facets of minifaction and magnification.

The rest is not finished text, just point summary of what was discussed in class.

- The Fourier transform lets us represent any function (called a signal in 1D, or an image in 2D) as a sum of sinusoids of different amplitudes and frequencies.
- To a large extent our ears, and to a lesser extent our eyes, perceive these amplitudes and frequencies. When we hear music for example, our brain doesn't perceive it in terms of time-varying air pressure, but more closely as a sum of different frequencies (pitches).

2 Magnification

- In audio, magnification is equivalent to taking a low-sampling-rate recording, say phone audio at 8KHz, and making a high-sampling-rate output, say CD quality at 44.1KHz. In this example, we go from 8000 samples in one second of the input and produce 44,100 samples in the output—a factor of a bit more $5.5\times$ magnification.
- Since our ears hear things, more or less, in terms of frequencies and amplitudes of sinusoids, let's look at magnifying one sinusoid. That is, we've taken a perfect sine wave, sampled it at 8KHz, and then from those discrete samples we want to generate new samples at a higher rate (44.1KHz) that ideally exactly match the original sine wave.
- If we use nearest-neighbour, i.e. piecewise constant, interpolation the result has discontinuous steps. If we take the Fourier transform—or just simply listen to this result—we see the original sine wave (which is good) plus a spiky error with lots of high frequency artifacts (which is bad). We hear this as weird high frequency distortion, and see it in images as distracting edges.
- Smoother interpolation can drastically reduce the error artifacts, making them much quieter—if smooth enough, quiet enough we no longer hear them—because we get a better approximation of the original sine wave.

3 Minification

- In audio, minification is the opposite: take a high-sampling-rate recording like a CD (44.1KHz) and produce a low-sampling-rate output (like a phone at 8KHz). Let's look at a sine wave again.
- The problems occur when the sine wave is higher frequency than we can reliably represent in the low-sampling-rate output (i.e. 4Khz or higher). Drawing what happens if we just take the values straight from the high-sampling-rate input—see the figure 4.2 in the Shirley text—makes it clear that the true high frequency signal can end up in the low-sampling-rate output "aliased" as a much lower frequency signal. This alias sounds like low frequency distortion, and is really distracting.
- The ideal thing to do is just eliminate any high frequencies that might alias to low frequency artifacts in the output. This is called "anti-aliasing".
- Simply taking an unweighted average over a block of samples, like we did for textures, damps out the higher frequencies but doesn't completely eliminate them—so we still have some degree of aliasing.

- Using a smoother weighted average can do a better job.

4 Quantization

- Another aspect of the problem we haven't touched on yet is **quantization**. We can conceptually work with image intensities or audio pressure levels which are continuous real numbers, and practically work with very high resolution floating-point numbers that are close enough to continuous real numbers that we can usually just pretend they are the same. However, a real display or a real audio system (any most real file formats) don't support floating-point: instead, they may use only a small fixed number of bits to encode each sample.

- For images, 8 bits per colour per pixel is the most common; for audio 16 bits is now the most common. For displaying an image on a typical monitor this means only 256 possible different values of red, green and blue are possible. For a CD this means only 65536 possible pressure levels are possible.

- The process of converting a continuous real number (or high precision floating-point number) into one of the small set of possible output values is called **quantization**

- The obvious quantization method of quantizing by just rounding to the nearest possible output value is actually pretty bad. There are two problems to highlight. One is that the output signal is going to jump discontinuously from one value to another, making it similar to the nearest-neighbour magnification we know is bad—we can expect high frequency distortion or artifacts. The other is that if the signal is quieter than the smallest nonzero value, it will quantize to just zero—e.g. instead of the audio getting gradually quieter in the fade at the end of a song, it will instead suddenly cut out into pure silence.

- Surprisingly, these problems can be fixed by adding random numbers (noise) to the output before quantization, of just enough magnitude to allow the noise to randomly bump the quantized value between two adjacent output values. For the first problem this doesn't change the fact that there will be high frequency artifacts, but the randomization will make them sound like white noise instead of distortion—and our brain is very tolerant of noise, happily ignoring it and letting us perceive the true signal underneath. For the second problem, the addition of noise means even when the signal gets very quiet, we still get nonzero values—and the probability of getting a zero versus a nonzero value is influenced by the very quiet signal. By averaging appropriately, our brain actually decodes that probability and ends up perceiving the very quiet signal plus noise—which it can ignore.

- This process of adding noise is called **dithering**.
- Dithering is crucial for audio, but also important for high quality rendering (where 256 values just isn't enough) and for rendering on low-bit-depth displays like black-and-white cell-phone screens.
- While random noise is probably optimal for audio, for images you have more leeway to dither in other ways. For example, laser printers and newspapers use half-toning, a pattern of black dots of varying sizes; your eyes average over the dots to get a perception of a varying grey-scale image plus (if you look closely enough) a "noisy" screen that can be ignored. Computer displays do something more sophisticated called "error diffusion" to get more accurate results than just random numbers can provide.

5 Collision Detection

Now that we've basically seen all the core topics of rendering to some degreee, let's talk about something completely different: **collisions**. Detecting collisions is often a crucial part of animation, interactive 3D applications, and even geometric editing. (Later we'll also talk about collision response, what to do once you've detected a collision, but we'll start by just figuring out when they happen.)

What is a collision? This can actually have several different definitions depending on the application:

- if two objects overlap in space (also known as intersection or penetration)
- if two objects are closer than some small threshold distance (also known as proximity)
- if during the motion of two objects from one time to another they touch or overlap, even if at the start and end they are well separated (also known as continuous collision detection).

Each type of geometric description of either of the two objects involved in a decision will probably need different treatment for all three of these options, making this a bit of a combinatorial explosion. We're not going to exhaustively consider this, but take a few examples and try to highlight a few general principles that apply generally.

The first and foremost point to make for collisions is that it's fine to cheat: as long as the results look (and feel) good in graphics, it's a good solution. We saw an example of this earlier in rendering, where we might cheat the normals of a triangle mesh for the shading formulas to better approximate the look of a smooth object, even though these are not the true geometric normals of the triangle mesh. Very

often we can do the same with collisions: simplify and speed up the code by using simpler geometric models for collisions than the ones we render.

As an example, very small objects (say droplets of water in a waterfall animation, or bullets in a violent game) can be approximated as single points. Their motion during the time interval from one frame of animation to the next can be further approximated as a straight line segment (i.e. assuming the velocity is constant during that time interval). If the surrounding space is stationary—it just sits there as these points move around—then continuous collision detection reduces to ray-tracing: define a ray whose origin is the initial position of the point and whose direction is the velocity, and check if this ray collides with the scene geometry for t values less than the length of the time interval. We already know a lot about ray-tracing, including how to use BVH's to speed it up, and the importance of dealing with floating point rounding errors.

Note that in this example, even after many simplifications and assumptions we ended up with something as least as complicated as ray-tracing, specifically the need for BVH's to accelerate the computation and the need to be careful with rounding error. More general collision detection routines hit those problems even harder than ray-tracing: if at all possible, you *really* want to simplify whatever you can for collisions.