Effective browsing of long audio recordings

Camille Goudeseune
Beckman Institute
University of Illinois at Urbana-Champaign
Urbana, Illinois, 61801 USA
cog@illinois.edu

ABSTRACT

Timeliner is a browser for long audio recordings and features that it derives from such recordings. Features can be either signal-based, like spectrograms, or model-based, like categorical classifiers.

Unlike conventional audio editors, Timeliner pans and zooms smoothly across many orders of magnitude, from days-long overviews to millisecond-scale details, with zero latency, zero flicker, and low CPU load. Also, to suggest which details are worth zooming in to examine, Timeliner's agglomerative hierarchical caches propagate feature-specific details up to wider zoom levels. Because these details are not averaged away, "big data" can be browsed rapidly and effectively. Several studies demonstrate this.

Categories and Subject Descriptors

H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems—*audio input/output*; H.4.3 [**Information Interfaces and Presentation**]: Communications Applications—*information browsers*; H.5.5 [**Information Interfaces and Presentation**]: Sound and Music Computing—*signal analysis, synthesis, and processing*

General Terms

Algorithms, Human Factors

Keywords

Deep Zoom, Big Data, HTK, Mipmap

1. INTRODUCTION

The Timeliner application is a multi-parameter zoomable timeline. Its source data is an audio recording, hours or even days long. It displays features derived from the recording as stacked images, horizontally synchronized with the waveform and a time axis (Fig. 1). Features include spectrograms [4], Mel-frequency cepstral coefficients (MFCC's) [13], spectrograms transformed to reduce the visual salience of non-anomalous events [12], and output log likelihoods from a bank of supervised event classifiers [20, 21].

Timeliner initially presents the whole recording to the user, who can then zoom in to interesting areas to rapidly find even very brief anomalous segments. Commodity laptops running Timeliner smoothly zoom across six orders of temporal magnitude.

Conventional audio editors compute the appearance of each horizontal extent of a pixel by undersampling data from the corresponding time interval. But in Timeliner, such intervals might be minutes long. During a fast pan or zoom, the flickering due to such naïve (and extreme) undersampling would entirely obscure the data. This would negate the benefit of such continuous pan-and-zoom gestures, of formulating database queries many times per second.

To restore smoothness and to improve performance for long recordings, Timeliner builds a multiscale agglomerative cache for the recording and for each derived feature. Such a cache efficiently reports the minimum, mean, and maximum data values found during a temporal subinterval, for either scalar data such as the recording itself, or for vector data in HTK format [19]. Final rendering assigns a hue, saturation, and value to each texel, using a color map particular to that kind of data.

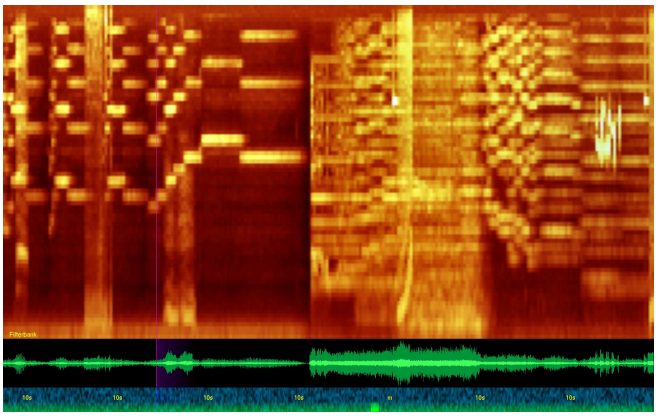

Figure 1: Timeliner's display: features (here, a single spectrogram), waveform (green), and time axis (blue).

The cached HTK-format data is itself summarized pyramidally and moved to graphics memory. This yet again improves performance by exploiting the GPU, and frees up

ACM Multimedia 2012 Nara, Japan

main memory to load longer recordings.

1.1 System Requirements

File parsing and interfacing to external utilities are both implemented in the scripting language Ruby. Intense numerical computation uses C++, both for speed and to reduce memory usage. Graphics are rendered with OpenGL and its utility toolkit GLUT.

Timeliner runs natively on Linux, in particular Ubuntu. (Early versions ran natively on Windows; a backport to native Windows is in progress.) Timeliner requires certain packages for Ubuntu 10.04 LTS: `sudo aptitude install freeglut3-dev gsl libgsl0-dev libsndfile1-dev ruby1.8 rubygems1.8 sox; sudo gem install gsl inline mmap narray rspec RubyInline`. Timeliner also requires HTK 3.4.1 (to handle files over 2 GB on 64-bit architectures, after running `./configure`, in all the subdirectories' makefiles, remove `-m32` from the CFLAGS definitions) [19]. Finally, to display features derived from pre-trained neural networks, install the software suite QuickNet [7].

2. PREPROCESSING

A Ruby/C++ preprocessor prepares data for the actual C++ audio browser. This lets the latter start very quickly, a convenience for consecutive or even simultaneous browsing sessions. If the preprocessing happens immediately after the unavoidably slow act of recording, its own duration is negligible.

The preprocessor computes features from the recording. It then converts both these features and the recording into serialized files, that is, pre-parsed data structures that the browser will load directly from disk. The preprocessor is given the following:

- a source recording in a format understood by the audio utility `sox`, such as .mp3 or .wav
- the directory that will contain the serialized files
- how many channels to use when computing spectrogram and MFCC features
- optionally, a directory of short "easter egg" sound files to be hidden in the source recording (see Sec. 7).

When serializing an HTK-format file, the preprocesser reads it as a memory-mapped file, instead of reading the whole file into memory and then needing yet more memory to store the parsed version (sampling rate, number of samples, arrays of floating-point numbers, *etc.*). For uniform graphical presentation, the preprocessor also normalizes each HTK feature's data to the range $[0, 1]$.

Because HTK incorrectly rounds sampling duration to the nearest exact multiple of 100 ns, common sampling rates other than 8 kHz or 16 kHz (exactly 1250 or 625 multiples of 100 ns, respectively) result in a drift in reported time. This may be acceptable for recordings lasting a few seconds, but not for a few days. The preprocessor therefore simply resamples the source recording at 16 kHz.

2.1 HTK-format features

Features are computed from the source recording with a sliding Hamming window. The window's size and skip can be tuned for particular kinds of recorded material. Each feature is written to disk in HTK format.

Spectrogram features are computed with HTK's filterbank feature. Saliency-maximized spectrograms convolve a standard spectrogram with a saliency-optimized filter computed by an external Matlab script [12].

Daubechies wavelet features use the Gnu Scientic Library (GSL) implementation. At the specified sampling rate, 32-sample intervals from the recording are convolved with a Hamming window and passed to GSL. The result from GSL is then stored as an HTK-format file.

Features based on neural networks (typically event classifiers) use tools from the SPRACHcore software package [6], in particular the utilities `feacat` and `qnsfwd` from its neural net code QuickNet [7]. First, `feacat` builds a "pad" file from the source recording. Then, using a file containing weights from a pretrained neural network, `qnsfwd` converts the pad file to an "act" file. From each line of this file, the floating-point weights of features are extracted and written to an HTK-format file.

One specialized feature is an unusual classifier, for use with easter eggs (see Sec. 7). If easter eggs are specified, they are combined with the source recording. When an egg is placed, its audio signal comes from a uniformly randomly chosen source egg, and its amplitude is also scaled randomly. The number of eggs placed is chosen to fit a specified density of eggs per unit time, usually no more than a few eggs per minute. Eggs are distributed randomly and uniformly without overlap, subject to the constraint that at least a specified minimum duration separates consecutive eggs.

For convenience when testing, an oracular "perfect" classifier indicates the locations of easter eggs. At each sample period this feature's value is 1.0 or 0.0, as that period does or does not contain an easter egg.

3. AGGLOMERATIVE CACHE

Recall that Timeliner's agglomerative caches are used to efficiently extract summary values, such as the maximum or the mean, from a subinterval of values from either the recording or a derived feature.

3.1 Construction

The cache is built as a rooted binary tree, starting from the leaves. A leaf node corresponds to one sample of data, and stores that sample's value. In the next layer of the tree, each node's payload stores the minimum, mean, and maximum of the two values of its children. The payload also stores the number of samples that its children represent (in this case, two). This pattern propagates up to the root node of the tree.

Elementary arithmetic computes a parent node's payload from its children's: the parent's minimum is the minimum of its children's minima, *etc.* It is only worth noting that the mean can be computed only if the payload includes the number of samples. (To avoid roundoff error, the mean is calculated in double precision. Nowhere else is roundoff evident, so the C++ code uses single precision to double how much data fits in memory.) When a layer in the tree has an odd number of nodes, the last node in the next layer up of course gets only one child, from whom its payload is copied verbatim.

Memory is significantly conserved by giving the leaves a payload with only one value, instead of the four used by non-leaf nodes. This single value is vacuously its own minimum, mean, and maximum. Similarly, the number of samples is

vacuously one and thus need not be stored.

To conserve even more memory, but at the cost of coarser temporal resolution, leaves may store more than one sample. For example, storing n consecutive samples in each leaf reduces the number of nodes (and hence the cache's memory footprint) n-fold, but limits zoom-in to a resolution n times coarser. This trade-off is particularly useful for the cache of the displayed waveform, if visible submillisecond detail would not assist a browsing task.

For vector-valued data such as a spectrogram's coefficients during a given sampling period, or even just stereo recording, each node can store not just one payload but a whole set of payloads, one for *each* element of the vector.

3.2 Querying

Given any time interval, that is, a contiguous subset of samples (a duration), the agglomerative cache returns that interval's payload, as generalized from the definition of a node's payload. Unsurprisingly, the cache does so by combining payloads from nodes. The number of nodes visited is proportional to the depth of the binary tree. This is of course logarithmic with respect to the recording's total duration, and independent of the interval's own duration, so queries are fast.

Combining payloads starts at the root node. At each node visited, if the node's time interval is *disjoint* with the query interval, it is discarded. If the node's interval is *contained* within the query, it is kept. Finally, if the node's interval overlaps *partially* with the query, testing continues with the node's children, down to the leaves. Going back up the tree as this recursion unwinds, payloads are combined with the same elementary arithmetic used to construct the cache in the first place.

3.3 Color maps

To reduce calling overhead, the C++ functions that query the cache return not just individual payloads but entire arrays of them. These payloads may also be immediately convolved with a color map, thereby returning an array of colors ready to be converted into an OpenGL texture map.

A color map converts an individual payload to a hue, saturation, and value (HSV), which in turn is converted to a red-green-blue triple. The extra step is warranted because HSV color space better matches human visual perception. (Recall that hue means "rainbow color," saturation means lack of grayness, and value means brightness.) We now present some practical examples of color maps.

Spectrograms that resolve the tracks of individual sinusoids benefit from a color map that emphasizes maxima and discards minima, such as mapping maximum linearly to value, and mean linearly to red—yellow hue. This yields a familiar black-body-radiation continuum of black—red—yellow—white with prominent peaks (Fig. 1). A conventional grayscale spectrogram would map mean to value and leave saturation at zero, but even this can be helped by letting the maximum slightly influence the value. This is because everyday spectra are asymmetric, with more peaks than notches. (Sequences of notch-dominated absorption spectra are found occasionally, in fields like time-domain spectroscopic optical coherence tomography [18]. If such sequences get long enough to warrant browsing in Timeliner, they would be well visualized by a mapping from mean-with-minimum to value.)

Classifiers of anomalous events, such as the easter egg oracle, should ideally emphasize only maxima. If the classifier suffers from false positives, the maximum can be diluted with the mean. A weighted sum of maximum and mean then maps to value and/or saturation.

Sharp thresholds in a feature can be emphasized by using a step function mapping to hue. This produces color bands like those in newspaper weather maps that indicate temperature to the nearest 10°F.

If both the minima and maxima of a feature are of interest, then the mean can be discarded. Value can then come from either the maximum or the minimum, whichever is farther from the global average. To then distinguish maxima from minima, maximum should map to a hue range such as yellow—blue. If saturation duplicates value as well, then the interesting extremes will be pure colors, delimited by grayer middle regions.

4. GRAPHICAL RENDERING

Timeliner's unusual attributes force all of its displays—the recording's waveform, the features derived from the recording, and even the time axis itself—to be rendered in unusual ways. These methods are elaborated here.

4.1 Mipmaps of features

Instead of expensively computing a texture map from the feature data many times per second, Timeliner precomputes texture maps just once. Because it cannot compute an infinite continuum of these, it computes them at only a finite number of resolutions. At any given zoom level, the two texture maps closest to that resolution are interpolated to produce the final display. Each texture map is one "level" of a mipmap, which is a venerable anti-aliasing technique for texture maps applied to three-dimensional scenes. Timeliner abuses this technique in a merely two-dimensional situation. (Like Timeliner's own agglomerative cache, the mipmap was motivated by wanting to eliminate flicker induced by subsampling [16].) As a performance bonus, the inter- and intra-level interpolation of textures is calculated by the GPU instead of the CPU.

Because OpenGL's well-known two-dimensional mipmaps cannot scale in only one of two dimensions, Timeliner resorts to OpenGL's more seldom used one-dimensional mipmaps. These mipmaps are tiled to cover the width of the full data. Each tile is as wide as possible, usually 8192 texels, to minimize overhead in graphics memory.

4.1.1 Information rate

Because a mipmap interpolates between a finite number of zoom levels, it merely approximates the output of the agglomerative cache. But this subtle quantization noise is practically invisible, especially when actively panning and zooming. The corresponding advantage is typically a tripled information rate, measured in texels per second (on commodity laptops with so-called dedicated graphics, as opposed to a lower-priced "integrated graphics" subsystem that steals main memory for the GPU).

4.1.2 Non-optical mipmap levels

Conventionally, each level of a mipmap is derived from its next higher-resolution level, often by just averaging texels. Advanced filters and windows sometimes elaborate this [2, 3], but the basic mechanism of computing one level from the

previous one remains. Such a mechanism fails here because the source data is not optical. In a deliberate violation of optical principles, we wish to preserve the visibility of interesting details across all resolutions. Thus, each level of the mipmap must be computed directly from the original data. Each texel, at each mipmap level, spans a particular interval of data. That data, efficiently summarized into that interval's payload by the agglomerative cache, then directly yields that texel's color.

The texel's color is rendered in OpenGL with classical techniques like `glColor4f` and `GL_MODULATE`; experiments with `glColorTableExt` and `EXT_paletted_texture` are in progress.

4.1.3 Conserving graphics memory

Each texel is stored in only 8 bits, instead of in the more conventional 32-bit red-green-blue format (GPUs internally pad 24-bit RGB to a 32-bit boundary). Unfortunately this constrains the form of a color map, rather like replacing an arbitrary function $f : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ with a composition $f : \mathbb{R}^3 \rightarrow \mathbb{R}^1 \rightarrow \mathbb{R}^3$. The $\mathbb{R}^1$ in this analogy, a mere 8-bit texel, appears to be an information bottleneck. Having already summarized an interval of data $x_0, x_1, ..., x_n$ into a three-number payload, this proposes to resummarize those three into just one number before once more expanding that number into a range of colors. Measuring information rate more precisely, in bits per second rather than texels per second, reducing each texel's size from 24 visible bits of color to only 8 would seem to reduce the information rate threefold. But in practice the reduction is less severe, for several reasons.

First, practical color maps do not exploit all three elements of their input payload. To extend the analogy, the first half of a particular color map's composition, $\mathbb{R}^3 \rightarrow \mathbb{R}^1$, is actually more like $\mathbb{R}^2 \rightarrow \mathbb{R}^1$ or even $\mathbb{R}^1 \rightarrow \mathbb{R}^1$.

Second, because the payload's three elements are somewhat correlated in everyday circumstances, squeezing the payload into 8 bits instead of into 24 does not really hide two thirds of the raw information. Such squeezing hides more of the statistical distribution of the $\{x_i\}$, but not much more. For example, if for a particular feature we are mostly interested in the mean of the $\{x_i\}$, but would also like to know a little about their spread, then a color map's input could come mostly from their mean, and slightly from the difference between their maximum and minimum. Such a color map that uses, say, 6-bit resolution from one payload element and 2-bit resolution from another, in terms of reduced bits of information is perceptually closer to $8 + 4 \rightarrow 6 + 2$ (that is, 1.5:1), than to the literal $8 + 8 \rightarrow 6 + 2$, or to the pedantic $8 + 8 + 8 \rightarrow 6 + 2$ (that is, 3:1). The threefold reduction may be only half as bad as the pedant fears. Such a low bit depth from a payload element may produce visible color banding in a static image. But again, such artifacts escape notice while browsing quickly.

Third, these smaller texels generally increase GPU performance. One quarter as many bytes need to be fetched from the GPU's memory, so the GPU's internal caches are better exploited when computing interpolations for rendering mipmaps. This improved performance reveals itself as higher resolution and frame rate. In other words, fewer *bits* per *texel* is countered by more *texels* per *frame* and more *frames* per *second*, increasing rather than decreasing their net product, bits per second.

Thus, 8-bit texels are a low (and possibly negative) price to pay for the corresponding advantage of quadrupled data storage.

4.2 Audio waveform

Even though a displayed waveform reveals little more than peak amplitude, its sheer familiarity justifies devoting part of the screen to it. To extract slightly more information from this part of the screen, then, the waveform's amplitude autoscales to match what is currently on screen. The unscaled display is bright, but possibly of near-zero amplitude. Behind that, the scaled display is dimmer; the dimness varies with scaling, to ensure that unusually large blobs are not excessively salient (Fig. 2). The auto-scaling is slew-rate limited, to reduce flicker artifacts when quickly zooming or panning. (Like an audio dynamic range compressor's attack and release controls, the rates for increasing and for decreasing the scaling differ.)

Recall that with HTK features, the agglomerative cache is used only to construct mipmaps. For waveform display, however, the browser uses the cache without any such indirection. This is because mipmaps cannot apply to such a two-dimensional display: a two-dimensional mipmap scales only in both dimensions, not just in the horizontal (temporal) dimension. (A possible alternative is the anisotropic mipmap called a ripmap [10, pp. 61–62], but in this specialized application it uses memory inefficiently [11]. At any rate, ripmaps are implemented by neither OpenGL nor commodity graphics hardware.)

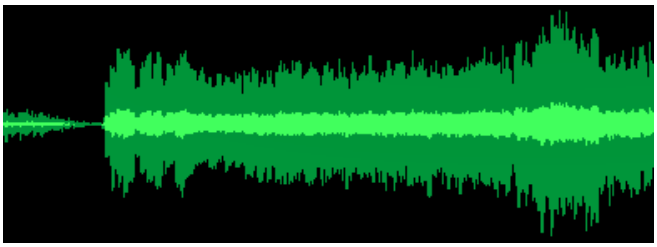

Figure 2: Detail of waveform display. The bright waveform's amplitude remains nominal, while the dim waveform behind it auto-scales to fill the allocated on-screen height.

4.3 Timeline and warnings

The bottom of the screen shows a temporal axis, but without distracting textual and numerical clutter (Fig. 3). Users are reminded of how far they are zoomed in by simple abbreviated units like `d` for days or `10s` for ten seconds. When zoom level changes, some units fade out and others fade in.

Behind these abbreviations, a faint white-noise texture provides optic flow during pan and zoom even if on-screen flow is momentarily absent elsewhere, as when displaying an interval of silence. This "infinite" texture is implemented as multiple layers of a single texture, which crossfade in opacity like a Shepard-Risset glissando's components crossfade in amplitude [15]. (With a Perlin noise source, such infinite noise textures work in higher dimensions [1]; even arbitrary exemplars such as the sky in Van Gogh's *The Starry Night* can produce infinitely zoomable textures [8].)

To represent the fraction of the whole recording that is on-screen, the noise texture is overlaid with an indicator

like the thumb of a scrollbar (the short bright-green patch in Fig. 3; also barely visible at bottom middle in Fig. 1).

If a pan or zoom-out tries to go past the start or end of the recording, a red flash at the start or end (or both) alerts the user of the limit being hit. Similarly, an attempt to zoom in beyond the data's resolution produces a yellow warning flash at the left and right edges of the screen.

Figure 3: Detail of timeline display.

5. OPERATION

Pan and zoom can be done with either the mouse or keyboard, whichever a user finds most familiar. However, the keyboard is faster for two reasons. First, panning by "pawing" the mouse involves wasted backstroke motion. Second, left-hand pan and zoom frees the right-hand mouse to have the sole task of aiming at particular positions to listen to sounds. The user's gaze too is freed from hunting for keys, because all the keys lie under the left hand without needing to aim the fingers, in the "WASD" layout that became dominant for mouse-keyboard real-time games in the mid-1990s. Users familiar with a mouse's scroll wheel for zooming in and out can use that to zoom, too.

To clarify optic flow despite noisy input gestures, both pan and zoom are slightly smoothed with a simple IIR filter, whose only sophistication is that it adapts to Timeliner's frame rate for identical temporal behavior independent of the hardware's capabilities.

5.1 Audio playback

After positioning a cursor (the thin purple vertical line in Fig. 1) with a mouse click, hitting the spacebar starts playback from that cursor. A subsequent tap of the spacebar stops playback. But if the user repositions the cursor during playback, that subsequent tap immediately skips playback to that position, without an intervening pause. Not having to explicitly stop before each start halves the number of taps, when briefly listening at many places as Timeliner's search strategy encourages.

During playback, a special cursor appears and progresses rightwards. Its movement is synchronized with the playback through a POSIX Threads mutex.

Audio is loaded as a memory-mapped file, for similar benefits as when the preprocessor uses this technique.

5.2 Annotations

To annotate the position of an interesting sound, the user moves the mouse to the sound's position and hits a key. The keystroke is confirmed, at that screen position, by a brief flash fading to a line. (To guide children playing the open-house game from Sec. 7, the flash's color indicated whether the annotation was a hit or a miss.) Another keystroke lets the user undo the last annotation.

Upon exit, Timeliner logs the annotations to a text file.

6. HANDHELD COMPUTERS

Timeliner's mouse-and-keyboard interface adapts well to multitouch screens on handheld and tablet PCs. Panning, zooming and playback fit the familiar flick, pinch, and tap gestures. However, because this scheme lacks an independently positionable cursor indicating where playback will start, the "skip ahead" shortcut requires a separate gesture, for instance a double-tap or two-finger tap.

Reasonable performance (again in terms of bits per second or texels per second) is possible on handheld computers running OpenGL ES or its derivative WebGL because the CPU and relatively slow main memory bus are only lightly loaded, even at full-screen resolution. This is thanks to the mipmaps already being stored in graphics memory. Examples of such OpenGL ES handhelds include Asus's Transformer Prime and Google's Nexus 7 tablet (both 1280×800, Tegra 3 graphics), Apple's iPad 3 (2048×1536, A5X graphics), and numerous smartphones that run the Android operating system.

7. EVALUATION

A central premise of Timeliner is that visual search for interesting sounds is faster and more accurate than aural search. Two scenarios have measurably demonstrated that Timeliner's smooth, deep zooming guided by helpful features results in effective browsing and annotation of long recordings.

The first scenario was at an uncontrolled, untutored laboratory open house. Here, many young children used Timeliner with a traditional spectrogram in the context of a game, racing the clock to find and annotate brief amusing sound effects (mooing, cuckoo clocks, motorcycles, *etc.*) hidden in orchestral music. This exhibit aimed to teach the novel concept of time-frequency representations to children aged about 6 to 13. Had the children ignored or misunderstood the spectrogram, they would have found about as many sound effects as they would have from real-time listening. In fact, most found about three times as many, and some found upwards of seven times as many [9]. This confirmed that their searching was primarily visual, and validated the effectiveness of Timeliner's two-handed input and audiovisual output.

The second scenario was a controlled study using adult subjects given a task of finding and annotating rare, quiet, anomalous sounds that had been injected into long recordings of business meetings. Compared to conventional spectrograms, saliency-maximized spectrograms doubled subjects' F-score (a combination of precision and accuracy) [12]. Again, visual search was critical to finding any anomalous sounds at all. Brief aural confirmation of anomalies early in a session quickly trained subjects how to interpret the visualization—that is, how to predict anomalousness without slow aural confirmation—for the remainder of the session.

8. MAKING LONG RECORDINGS

Handheld audio recorders that store data on memory cards originally marketed for cameras have become common, but they need some persuading to produce multi-hour recordings. First, automatic power-down may need to be overridden. Second, recording may need to be restarted when the data file reaches 2 GB or 4 GB on a FAT16-formatted card (4 GB for FAT32, 2 TB for FAT64). Finally, short battery life may demand an external power supply.

9. HIGHER DIMENSIONAL EXTENSIONS

By generalizing the agglomerative cache's binary tree to a quadtree or an octree, and then computing more conventional two- or three-dimensional mipmaps, similar smooth panning and zooming of two- or three-dimensional data is possible. Others have suggested this for the special case of purely visual two-dimensional data. For example, Silverlight's DeepZoomImageTileSource class implements this in the context of serving images to a web browser plugin [5]; this has been adapted to databases of scientific imagery [14, 17].

10. CONCLUSIONS

11. ACKNOWLEDGMENTS

This work is funded by National Science Foundation grant 0807329.

The author thanks Mark Hasegawa-Johnson, Sarah King, Kai-Hsiang Lin, and Xiaodan Zhuang for their technical observations and insights, and Audrey Fisher for meticulous proofreading.

12. REFERENCES

[1] P. Bénard, A. Bousseau, and J. Thollot. Dynamic solid textures for real-time coherent stylization. In *Symposium on Interactive 3D Graphics and Games (I3D)*, pages 121–127. ACM, 2009.

[2] J. Blow. Mipmapping, part 1. *Game Developer Magazine*, 8(12):13–17, Dec. 2001.

[3] J. Blow. Mipmapping, part 2. *Game Developer Magazine*, 9(1):16–19, Jan. 2002.

[4] D. Cohen, C. Goudeseune, and M. Hasegawa-Johnson. Efficient simultaneous multi-scale computation of FFTs. Technical Report FODAVA-09-01, NSF/DHS FODAVA-Lead: Foundations of Data and Visual Analytics, 2009.

[5] M. Corp. Silverlight 5. http://www.silverlight.net/, 2012.

[6] D. Ellis. The SPRACH project. http://www.icsi.berkeley.edu/ dpwe/projects/sprach/, 1999.

[7] D. Ellis, C. Oei, C. Wooters, and P. Faerber. Quicknet. http://www.icsi.berkeley.edu/Speech/qn.html, 2012.

[8] C. Han, E. Risser, R. Ramamoorthi, and E. Grinspun. Multiscale texture synthesis. *ACM Trans. Graphics*, 27(3), Aug. 2008.

[9] M. Hasegawa-Johnson, C. Goudeseune, J. Cole, H. Kaczmarski, H. Kim, S. King, T. Mahrt, J.-T. Huang, X. Zhuang, K.-H. Lin, H. V. Sharma, Z. Li, and T. S. Huang. Multimodal speech and audio user interfaces for K-12 outreach. In *Proc. Asia-Pacific Signal and Information Processing Assn.*, 2011.

[10] P. S. Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, University of California, Berkeley, June 1989.

[11] C.-F. Hollemeersch, B. Pieters, P. Lambert, and R. Van de Walle. A new approach to combine texture compression and filtering. *The Visual Computer*, 28(4):371–385, 2012.

[12] K.-H. Lin, X. Zhuang, C. Goudeseune, S. King, M. Hasegawa-Johnson, and T. S. Huang. Improving faster-than-real-time human acoustic event detection by saliency-maximized audio visualization. In *Proc. ICASSP*, pages 1–4, 2012.

[13] P. Mermelstein. Distance measures for speech recognition: Psychological and instrumental. In C. H. Chen, editor, *Pattern Recognition and Artificial Intelligence*, pages 374–388. Academic Press, New York, 1976.

[14] B. Reitinger, M. Hoefler, A. Lengauer, R. Tomasi, M. Lamperter, and M. Gruber. Dragonfly: interactive visualization of huge aerial image datasets. In *Proc. 21st ISPRS Congress*, volume 37, pages 491–494, 2008.

[15] R. N. Shepard. Circularity in judgements of relative pitch. *J. Acoust. Soc. Am.*, 36(12):2346–2353, 1964.

[16] L. Williams. Pyramidal parametrics. *SIGGRAPH Computer Graphics*, 17(3):1–11, July 1983.

[17] R. Williams, L. Yan, X. Zhou, L. Lu, A. Centeno, L. Kuan, M. Hawrylycz, and G. Rosen. Global exploratory analysis of massive neuroimaging collections using Microsoft Live Labs Pivot and Silverlight. In *Front. Neurosci. Conference Abstract: Neuroinformatics*, 2010.

[18] C. Xu and S. A. Boppart. Comparative performance analysis of time-frequency distributions for spectroscopic optical coherence tomography. In *Biomedical Topical Meeting*, page FH9. Optical Society of America, 2004.

[19] S. Young, G. Evermann, T. Hain, D. Kershaw, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. Woodland. *The HTK Book*. Cambridge University Engineering Dept., Cambridge, UK, 2002.

[20] X. Zhuang, X. Zhou, M. A. Hasegawa-Johnson, and T. S. Huang. Real-world acoustic event detection. *Pattern Recognition Letters*, 31(2):1543–1551, Sept. 2010.

[21] X. Zhuang, X. Zhou, T. S. Huang, and M. Hasegawa-Johnson. Feature analysis and selection for acoustic event detection. In *Proc. ICASSP*, pages 17–20, 2008.