Multiresolution Video

Adam Finkelstein *Charles E. Jacobs* *David H. Salesin*

Department of Computer Science and Engineering*
University of Washington

Abstract

We present a new representation for time-varying image data that allows for varying—and arbitrarily high—spatial and temporal resolutions in different parts of a video sequence. The representation, called *multiresolution video*, is based on a sparse, hierarchical encoding of the video data. We describe a number of operations for creating, viewing, and editing multiresolution sequences. These operations support a variety of applications: multiresolution playback, including motion-blurred "fast-forward" and "reverse"; constant-speed display; enhanced video scrubbing; and "video clip-art" editing and compositing. The multiresolution representation requires little storage overhead, and the algorithms using the representation are both simple and efficient.

CR Categories and Subject Descriptors:
H.5.1 [Information Interfaces]: Multimedia Information Systems—video
I.3.3 [Computer Graphics]: Picture/Image Generation—display algorithms
I.4.10 [Image Processing]: Image Representation—hierarchical

Additional Keywords: clip-art, compositing, image pyramids, multigrid methods, multimedia, scientific visualization, video editing

1 Introduction

Scientists often run physical simulations of time-varying data in which different parts of the simulation are performed at differing spatial and temporal resolutions. For example, in a simulation of the air flow about an airplane wing, it is useful to run the slowly-varying parts of the simulation—generally, the portion of space further from the wing—at a fairly coarse scale, both spatially and temporally, while running the more complex parts—say, the region of turbulence just aft of the wing—at a much higher resolution. The *multigrid techniques* used frequently for solving large-scale problems in physics [15], astronomy [16], meteorology [1], and applied mathematics [8] are a common example of this kind of computation.

In this paper, we present a new approach for representing the time-varying data produced by such algorithms, called *multiresolution video*. The multiresolution video representation provides a means of capturing time-varying image data produced at multiple scales, both spatially and temporally. In addition, we introduce efficient algorithms for viewing multiresolution video at arbitrary scales and speeds. For example, in a sequence depicting the flow of air about a wing, a user can interactively zoom in on an area of relative turbulence, computed at an enhanced *spatial resolution*. Analogously, fast-changing components in a scene can be represented and viewed at a higher *temporal resolution*, allowing, for example, a propeller blade to be represented and viewed in slow motion.

Moreover, we have found that multiresolution video has applications that are useful even for conventional *uniresolution* video. First, the representation facilitates a variety of viewing applications, such as multiresolution playback, including motion-blurred "fast-forward" and "reverse"; constant-speed viewing of video over a network with varying throughput; and an enhanced form of video "scrubbing." The representation also provides a controlled degree of lossy compression, particularly in areas of the video that change little from frame to frame. Finally, the representation supports the assembly of complex multiresolution videos from either uniresolution or multiresolution "video clip-art" elements.

1.1 Related work

The multiresolution video representation described in this paper generalizes some of the multiresolution representations that have previously been proposed for images, such as "image pyramids" [18] and "MIP maps" [19]. It is also similar in spirit to the wavelet-based representations for images described by Berman *et al.* [2] and Perlin and Velho [11]. In particular, like these latter works, our representation is sparse, and it supports efficient compositing operations [12] for assembling complex frames from simpler elements.

Several commercially available video editing systems support many of the operations described in this paper for uniresolution video. For example, Adobe After Effects allows the user to view video segments at low resolution and to construct an edit list that is later applied to the high-resolution frames off-line. Discrete Logic's Flame and Flint systems also provide digital video compositing and many other digital editing operations on videos of arbitrary resolution. Swartz and Smith [17] describe a language for manipulation of video segments in a resolution-independent fashion. However, the input and output from all of these systems is uniresolution video.

Multiresolution video also allows the user to pan and zoom to explore a flat video environment. This style of interaction is similar in spirit to two image-based environments: Apple Computer's QuickTime® VR [3] and the "plenoptic modeling" system of McMillan and Bishop [9]. These methods provide an image-based representation of an environment that surrounds the viewer. In section Section 4.5, we describe how such methods can be combined with multiresolution video to create a kind of "multiresolution video QuickTime VR," in which a viewer can investigate a panoramic environment by panning and zooming, with the environment changing in time and having different amounts of detail in different locations.

While not the emphasis of this work, we also describe a simple form of lossy compression suitable for multiresolution video. Video compression is a heavily-studied area, with too many papers to adequately survey here. MPEG [5] and QuickTime [13] are two industry standards. Other techniques based on multiscale transforms [7, 10] might be adapted to work for multiresolution video.

---

* For project updates, addresses, and email see our Web page:
`http://www.cs.washington.edu/research/graphics/mrvideo`

1.2 Overview

The rest of this paper is organized as follows. Section 2 describes our representation for multiresolution video, and Section 3 describes how it is created and displayed. Section 4 describes a variety of applications of the multiresolution video representation, and Section 5 provides some concrete examples. Finally, Section 6 outlines some areas for future work. The appendix provides additional low-level operations useful for editing multiresolution video.

2 Representation

Our goals in designing a multiresolution video representation were fivefold. We wanted it to:

- support varying spatial and temporal resolutions;
- require overall storage proportional only to the detail present (with a small constant of proportionality);
- efficiently support a variety of primitive operations for creating, viewing, and editing the video;
- permit lossy compression; and
- require only a small "working storage" overhead, so that video could be streamed in from disk as it is needed.

The rest of this section describes the multiresolution video format we chose and an analysis of the storage required.

2.1 The basic multiresolution video format

Perhaps the most obvious choice for a multiresolution video format would be a sparse octree [14], whose three dimensions were used to encode the two spatial directions and time. Indeed, such a representation was our first choice, but we found that it did not adequately address a number of the goals enumerated above. Put briefly, the problem with such a representation is that it couples the dimensions of space and time too tightly. In an octree structure, each node would correspond to a "cube" with a fixed extent in space and time. Thus, it would be efficient to rescale a video to, say, twice the spatial resolution only if it were equally rescaled in time—that is, played at half the speed. We therefore needed to develop a representation that, while still making it possible to take advantage of temporal and spatial coherence, could couple space and time more loosely.

The structure we ultimately chose is a sparse binary tree of sparse quadtrees. The binary tree encodes the flow of time, and each quadtree encodes the spatial decomposition of a frame (Figure 1).

In the binary tree, called the *Time Tree*, each node corresponds to a single image, or *frame*, of the video sequence at some temporal resolution. The leaves of the Time Tree correspond to the frames at the highest temporal resolution for which information is present in the video. Internal nodes of the Time Tree correspond to box-filtered averages of their two children frames. Visually, these frames appear as motion-blurred versions of their children. Note that this representation supports video sequences with varying degrees of temporal resolution simply by allowing the Time Tree to grow to different depths in different parts of the sequence. For convenience, we will call the child nodes of the Time Tree *child time nodes* and their parents *parent time nodes*. We will use capitalized names for any time node.

Time Tree nodes are represented by the following data structure:

type *TimeNode* = **record**
    frame: **pointer to** *ImageNode*
    Half1, *Half2*: **pointer to** *TimeNode*
end record

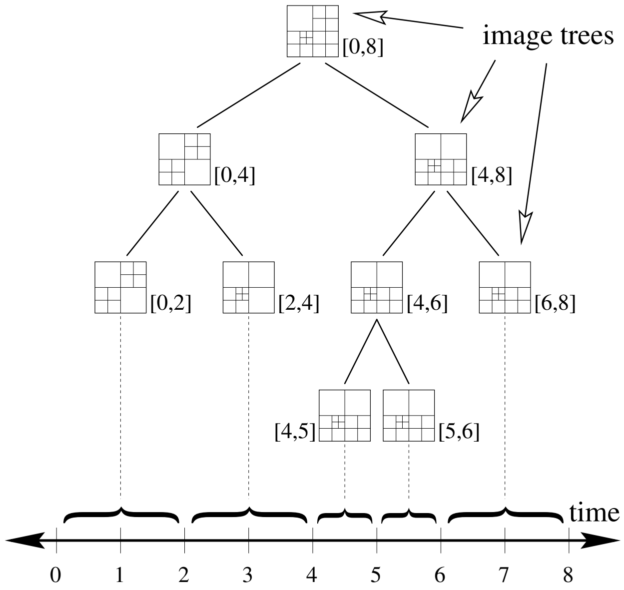


Figure 1 Binary tree of quadtrees.

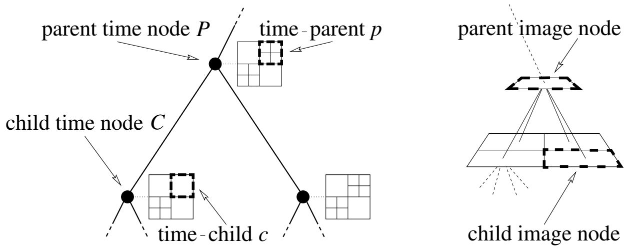
Each node of the Time Tree points to a sparse quadtree, called an *image tree*, which represents the multiresolution image content of a single frame of the video sequence. In analogy to the Time Tree, leaves of an image tree correspond to pixels at the highest spatial resolution for which information is present in the particular frame being represented. Internal nodes of an image tree correspond, once again, to box-filtered averages of their children—in this case, to a 2×2 block of higher-resolution pixels. Note that the image tree supports varying spatial resolution simply by allowing the quadtree to reach different depths in different parts of the frame. We will call the child nodes of an image tree *child image nodes* and their parents *parent image nodes*. In our pseudocode we will use lower-case names for any image node. Figure 4 shows a frame from a video clip, where leaf nodes of the image tree are boxed in yellow.

Specifically, here is how we encode each node in the image tree:

type *ImageNode* = **record**
    type: TREE | COLOR
    uplink: *UpLinkInfo*
    union
        tree: **pointer to** *ImageSubtree*
        color: *PixelRGBA*
    end union
end record

type *ImageSubtree* = **record**
    avgcolor: *PixelRGBA*
    child$[0..1, 0..1]$: **array of** *ImageNode*
end record

Each subtree contains both the average color for a region of the image, stored as an RGBA pixel, and also image nodes for the four quadrants of that region. We compute the average of the pixels as if each color channel were premultiplied by alpha—as prescribed by Porter and Duff [12]—but we do not actually represent the pixels that way in our image nodes, in order to preserve color fidelity in highly-transparent regions. Each image node generally contains a pointer to a subtree for each quadrant. However, if a given quadrant only has a single pixel's worth of data, then the color of the pixel is stored in the node directly, in place of the pointer. (This trick works nicely, since an RGBA pixel value is represented in our system with 4 bytes, the same amount of space as a pointer. Packing the pixel in-



(a) Time tree **(b)** Image tree

Figure 2 Parent-child relationships in the trees.

formation into the pointer space allows us to save a large amount of memory that we might otherwise waste on null pointers at the leaves.) There is also an *uplink* field, whose use we will discuss in the next section.

There is an additional relationship between image nodes that is not represented explicitly in the structure, but which is nevertheless crucial to our algorithms. As described already, there are many different image nodes that correspond to the same region of space, each hanging from a different time node. We will call any two such image nodes *time-relatives*. In particular, for a given image node c hanging from a time node C, we will call the time-relative p hanging from the parent time node P of C the *time-parent* of c. In this case, the image node c is also called the *time-child* of p. (See Figure 2.) Note that a given node does not necessarily have a time-parent or a time-child, as the quadtree structures hanging from P and C may differ.

2.2 Temporal coherence

Recall that the representation of each frame exploits spatial coherence by pruning the image tree at nodes for which the image content is nearly constant. We can take advantage of temporal coherence in a similar way, even in regions that are spatially complex.

Consider an image node p and its two time-children c_1 and c_2. Whenever the images in c_1 and c_2 are similar to each other, the image in p will be similar to these images as well. Rather than triplicating the pixels in all three places, we can instead just store the image in the time-parent p and allow c_1 and c_2 to point to this image directly. We call such pointers *up-links*. See Figure 3 for a schematic example. Figure 5 shows a frame from a multiresolution video clip in which all up-link regions (which cover most of the frame) are shaded red.

The up-links are described by the following structure:

type *UpLinkInfo* = **record**
  linked: *Boolean*
  type: FIRST | MIDDLE | LAST
end record

The *linked* field tells whether or not there is an up-link. There is also a *type* field, which we will describe in Section 3.2.

2.3 Storage complexity

Now that we have defined the multiresolution video data structure, we can analyze its storage cost. The *type* and *uplink* fields require very few bits, and in practice these two fields for all four children may be bundled together in a single 4-byte field in the *ImageSubtree* structure. Thus, each *ImageSubtree* contains 4 bytes (for the average color), 4×4 bytes (for the children), and 4 bytes (for the flags), yielding a total of 24 bytes. Each leaf node of an image tree comprises 4 pixels, and there are 4/3 as many total nodes in these trees

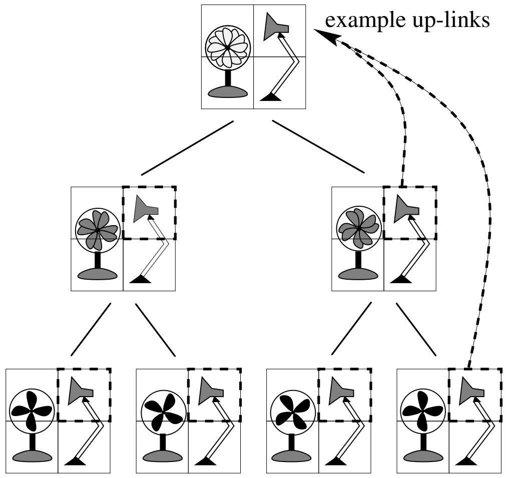


Figure 3 Exploiting temporal coherence. The quadrants containing the Luxo lamp are not duplicated in the lower six frames of the Time Tree. Instead, the right two quadrants in all six frames contain "up-links" to the corresponding quadrants in the Time Tree's root.

as there are leaves. Assuming P pixels per time node, we get:

$$\frac{24\text{ bytes}}{\text{node}} \times \frac{4\text{ nodes}}{3\text{ leaf}} \times \frac{1\text{ leaves}}{4\text{ pixels}} \times \frac{P\text{ pixels}}{\text{time node}} = \frac{8P\text{ bytes}}{\text{time node}}$$

Furthermore, there are twice as many time nodes as there are leaves (or frames) in the Time Tree, so the storage complexity is really $16P$ bytes/frame. In addition, each *TimeNode* contains $3 \times 4 = 12$ bytes, and there are twice as many nodes in this tree as there are leaves. Thus, the Time Tree needs an additional 24 bytes/frame. However, since $16P$ is generally much larger than 24, we can ignore the latter term in the analysis. The overall storage is therefore 16 bytes/pixel.

In the worst case—a complete tree with no up-links—we have as many pixels in the tree as in the original image. Thus, the tree takes 4 times as much space as required by just the highest-resolution pixel information alone. It is worthwhile to compare this overhead with the cost of directly storing the same set of time- and space-averaged frames, without allowing any space for pointers or flags. Such a structure would essentially involve storing all powers-of-two time and spatial scales of each image, requiring a storage overhead of $8/3$. Thus, our storage overhead of 4 is only slightly larger than the minimum overhead required. However, as will be described in Section 3.1, the very set of pointers that makes our worst-case overhead larger also permits both lossless and lossy compression by taking advantage of coherence in space and time.

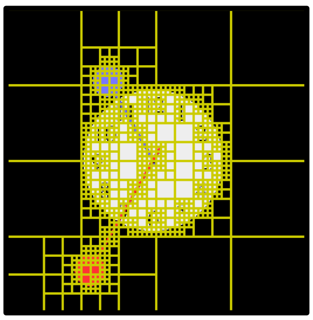


Figure 4 Quadtree.

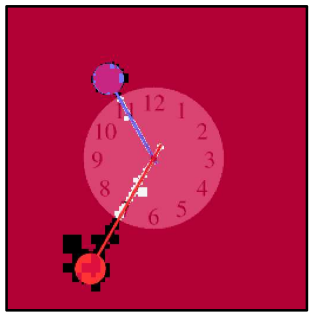


Figure 5 Up-links.

2.4 Working storage

One of the goals of our representation was to require a small "working storage" overhead, so that video could be streamed in from disk only as it is needed. This feature is crucial for viewing very large sequences, as well as for the editing operations we describe in the appendix. As we will see when we discuss these operations in detail, this goal is easily addressed by keeping resident in memory just the image trees that are currently being displayed or edited, along with all of their time-ancestors. For a video clip with 2^k frames, the number of time-ancestors required is at most k.

2.5 Comparison with wavelets

Wavelets have been successfully used to represent multiresolution functions in a variety of domains. Our first impulse—and, in fact, our first implementation—was to use a 3D wavelet representation for the video. However, we eventually moved to the data structure described in this section for several reasons. First, the box basis functions we use now are simpler, making it faster to render a frame of video. Second, wavelet coefficients require increasing numbers of bits at finer levels of detail, so we had to use floating-point numbers (rather than bytes) to store the color channels—a factor of four expansion. Third, the wavelets we were using (nonstandard tensor-product Haar wavelets) made it difficult to separate the spatial and temporal dimensions; thus, it was expensive to extract a frame in which the time and space dimensions were scaled differently. While it is possible to construct a wavelet basis that avoids this problem, we were not able to find an efficient compositing algorithm for it. Finally, the current representation takes advantage of areas of a video sequence that have temporal coherence but no spatial coherence; the wavelets we used were unable to compress such sequences.

3 Basic algorithms

In this section we describe algorithms for creating and displaying multiresolution video. Algorithms for translating, scaling, and compositing multiresolution video clips appear in the appendix.

3.1 Creating multiresolution video

We begin with the problem of creating multiresolution video from conventional uniresolution video. We break this process into two parts: creating the individual frames, and linking them together into a multiresolution video sequence.

3.1.1 Creating the individual frames

Given a $2^\ell \times 2^\ell$ source frame S we construct an image tree by calling the following function with arguments $(S, 0, 0, \ell)$:

```
function CreateFrame(S, x, y, ℓ): returns ImageNode
    if ℓ = 0 then return ImageNode(COLOR, S[x, y])
    for each i,j ∈ {0,1} do
        x′ ← 2x + i
        y′ ← 2y + j
        subtree.child[i,j] ← CreateFrame(S, x′, y′, ℓ − 1)
    end for
    subtree.avgcolor ← AverageChildren(subtree.child[0..1, 0..1])
    return ImageNode(TREE, subtree)
end function
```

Image trees built from images that are not of dimension $2^\ell \times 2^\ell$ are implicitly padded with transparent, black pixels.

The quadtree constructed by *CreateFrame*() is complete. The next step is to take advantage of spatial coherence by culling redundant information from the tree. The following function recursively traverses the image tree p and prunes any subtree whose colors differ from its average color a by less than a threshold δ:

```
function PruneTree(p, a, δ): returns Boolean
    if p.type = COLOR then return (ColorDiff(p.color, a) ≤ δ)
    prune ← TRUE
    for each i,j ∈ {0,1} do
        prune ← prune and PruneTree(p.child[i,j], p.avgcolor, δ)
    end for
    if prune = FALSE then return FALSE
    free(p.child[0..1, 0..1])
    p ← ImageNode(COLOR, p.avgcolor)
    return TRUE
end function
```

Choosing $\delta = 0$ yields lossless compression, whereas using $\delta > 0$ permits an arbitrary degree of lossy compression at the expense of image degradation. The function *ColorDiff*() measures the distance between two colors (r_1, g_1, b_1, a_1) and (r_2, g_2, b_2, a_2). We chose to measure the distance as the sum of the distances between color components, weighted by their luminance values:

$$0.299|r_1a_1 - r_2a_2| + 0.587|g_1a_1 - g_2a_2| + 0.114|b_1a_1 - b_2a_2|$$

In practice, the source material may be multiresolution in nature. For example, the results of some of the scientific simulations described in Section 5 were produced via adaptive refinement. It is easy to modify the function *CreateFrame*() to sample source material at different levels of detail in different parts of a frame. In this case, the recursive function descends to varying depths, depending on the amount of detail present in the source material.

3.1.2 Linking the frames together

The next step is to link all the frames together into the Time Tree. We first insert all the image trees at the leaves of the Time Tree, and then compute all of the internal nodes by averaging pairs of frames in a depth-first recursion. Now that the complete Time Tree is built, the following two procedures discover and create all the up-links:

```
procedure MakeMRVideo(Timetree, δ):
    for each Half ∈ {Half1, Half2} of Timetree do
        if Half ≠ NULL then
            MakeUpLinks(Half.frame, Timetree.frame, δ)
            MakeMRVideo(Half, δ)
        end if
    end for
end procedure

function MakeUpLinks(p, c, δ): returns Boolean
    c.uplink.linked ← FALSE
    if p = NULL or p.type ≠ c.type then
        return FALSE
    else if c.type = COLOR then
        c.uplink.linked ← (ColorDiff(p.color, c.color) ≤ δ)
        return c.uplink.linked
    end if
    link ← TRUE
    for each i,j ∈ {0,1} do
        link ← (link and MakeUpLinks(p.child[i,j], c.child[i,j]), δ)
    end for
    if link = FALSE then return FALSE
    free(c.tree)
    c.tree ← p.tree
    c.uplink.linked ← TRUE
    return TRUE
end function
```

The *MakeMRVideo*() routine works by finding all of the up-links between the root of the Time Tree and its two child time nodes. The routine then calls itself recursively to find up-links between these children and their descendents in time. Because of the preorder recursion, up-links may actually point to any time-ancestor, not just a time-parent. (See Figure 3.)

The *MakeUpLinks*() function attempts to create an up-link from a time-child c to its time-parent p. An up-link is created if the two nodes are both subtrees with identical structure, and all of their descendent nodes are sufficiently close in color. The function returns TRUE if such an up-link is created. It also returns TRUE if the two nodes are colors and the two colors are sufficiently close; it furthermore sets the child node's *uplink* flag, which is used to optimize the display operation in the following section.

After executing *MakeMRVideo*(), we traverse the entire Time Tree in a separate pass that sets the *type* field of the *uplink* structure, whose use is explained in the following section.

3.2 Display

Drawing a frame at an arbitrary power-of-two spatial or temporal resolution is simple. Displaying at a particular temporal resolution involves drawing frames at the corresponding level in the Time Tree. Displaying at a particular spatial resolution involves drawing the pixels situated at the corresponding level in the image trees.

The up-links that were used in the previous section to optimize storage can also play a role in optimizing the performance of the display routine when playing successive frames. We would like to avoid refreshing any portion of a frame that is not changing in time; the up-links provide exactly the information we need. In particular, if we have just displayed frame t, then we do not need to render portions of frame $t+1$ (at the same time level) that share the same up-links. We will use the *type* field in the *UpLinkInfo* structure to specify the first and last up-links of a sequence of frames that all share the same parent data. When playing video forward, we do not need to render any region that is pointed to by an up-link, unless it is a FIRST up-link. Conversely, if we are playing backward, we only need to render LAST up-links.

To render the image content c of a single multiresolution video frame at a spatial resolution $2^\ell \times 2^\ell$, we can call the following recursive routine, passing it the root c of an image tree and other parameters $(0, 0, \ell)$:

```
procedure DrawImage(c, x, y, ℓ):
    if c.uplink.linked and c.uplink.type ≠ FIRST then return
    if c.type = COLOR then
        DrawSquare(x, y, 2^ℓ, c.color)
    else if ℓ = 0 then
        DrawPixel(x, y, c.avgcolor)
    else
        for each i, j ∈ {0, 1} do
            DrawImage(c.child[i, j], 2x + i, 2y + j, ℓ − 1)
        end for
    end if
end procedure
```

The routine *DrawSquare*() renders a square at the given location and size in our application window, while *DrawPixel*() renders a single pixel. Note that *DrawImage*() assumes that the video is being played in the forward direction from beginning to end. A routine to play the video in reverse would have to use LAST in place of FIRST in the first line. A routine to display a single frame that does not immediately follow the previously displayed frame (for example, the first frame to be played) would have to omit the first line of code entirely.

One further optimization is that we actually keep track of the bounding box of non-transparent pixels in each frame. We intersect this bounding box with the rectangle containing the visible portion of the frame on the screen, and only draw this intersection. Thus, if only a small portion of the frame is visible, we only draw that portion.

The *DrawImage*() routine takes time proportional to the number of squares that are drawn, assuming that the time to draw a square is constant.

Fractional-level zoom

The *DrawImage*() routine as described displays multiresolution video at any power-of-two spatial resolution. Berman *et al.* [2] describe a simple method to allow users to view multiresolution images at any arbitrary scale. We have adapted their method to work for multiresolution video. The basic idea is that if we want to display a frame of video at a fractional level between integer levels $\ell-1$ and ℓ, we select pixels from the image tree as though we were drawing a $2^\ell \times 2^\ell$ image, and then display those pixels at locations appropriate to the fractional level. So if a pixel would be drawn at location (x, y) in a $2^\ell \times 2^\ell$ image, then it would be drawn at location (x', y') in an $M \times M$ image, where

$$x' = \lfloor xM/2^\ell \rfloor \qquad\qquad y' = \lfloor yM/2^\ell \rfloor$$

Furthermore, as with MIP maps [19], we interpolate between the colors appearing at levels ℓ and $\ell - 1$ in the image tree in order to reduce point-sampling artifacts. Drawing at this fractional level is only slightly more expensive than drawing pixels at level ℓ.

Similarly, even though we are selecting frames from the Time Tree corresponding to power-of-two intervals of time, we can achieve "fractional rates" through the video, as will be described in Section 4.2.

4 Applications

We now describe several applications of the primitive operations presented in the last section. These applications include multiresolution playback, with motion-blurred "fast-forward" and "reverse"; constant perceived-speed playback; enhanced video scrubbing; "video clip-art" editing and compositing; and "multiresolution video QuickTime VR."

These applications of multiresolution video serve as "tools" that can be assembled in various combinations into higher-level applications. We describe our prototype multiresolution video editing and viewing application in Section 5.

4.1 Multiresolution playback

The primary application of multiresolution video is to support playback at different temporal and spatial resolutions. To play a video clip at any temporal resolution 2^k and spatial resolution $2^\ell \times 2^\ell$ we simply make successive calls to the procedure *DrawImage*(), passing it a series of nodes from level k of the Time Tree, as well as the spatial level ℓ. We can zoom in or out of the video by changing the level ℓ.

Similarly, for "motion-blurred" fast-forward and reverse, we use a smaller time level k. In our implementation the motion-blur effect comes from simple box filtering of adjacent frames. Though box-filtering may not be ideal for creating high-quality animations, it does appear to be adequate for searching through video.

Sometimes it may be desirable to have a limited degree of motion blur, which might, for example, blur the action in just the first half of the frame's time interval. This kind of limited motion blur can be implemented by descending one level deeper in the Time Tree, displaying the first child time node rather than the fully motion-blurred frame.

4.2 Constant perceived-speed playback

During video playback, it is useful to be able to maintain a constant perceived speed, despite variations in the network throughput or CPU availability. Multiresolution video provides two ways of adjusting the speed of play, which can be used to compensate for any such variations in load. First, by rendering individual frames at a finer or coarser spatial resolution, the application can adjust the rendering time up or down. Second, by moving to higher or lower levels in the Time Tree, the application can also adjust the perceived rate at which each rendered frame advances through the video.

These two mechanisms can be traded off in order to achieve a constant perceived speed. One possibility is to simply adjust the spatial resolution to maintain a sufficiently high frame rate, say 30 frames/second. If, however, at some point the degradation in spatial resolution becomes too objectionable (for instance, on account of a large reduction in network bandwidth), then the application can drop to a lower frame rate, say, 15 frames/second, and at the same time move to the next higher level of motion-blurred frames in the Time Tree. At this lower frame rate, the application has the liberty to render more spatial detail, albeit at the cost of more blurred temporal detail.

Note that by alternating between the display of frames at two adjacent levels in the Time Tree, we can play at arbitrary speeds, not just those related by powers of two.

4.3 Scrubbing

Conventional broadcast-quality video editing systems allow a user to search through a video interactively by using a slider or a knob, in a process known as "scrubbing." In such systems, frames are simply dropped to achieve faster speeds through the video.

Multiresolution video supports a new kind of scrubbing that shows all of the motion-blurred video as the user searches through it, rather than dropping frames. In our implementation, the user interface provides a slider whose position corresponds to a position in the video sequence. As the user moves the slider, frames from the video are displayed. The temporal resolution of these frames is related to the speed at which the slider is pulled: if the slider moves slowly, frames of high temporal detail are displayed; if the slider moves quickly, blurred frames are displayed.

The benefits of this approach are similar to those of the constant perceived-speed playback mechanism described above. If the slider is pulled quickly, then the application does not have an opportunity to display many frames; instead, it can use the motion-blurred frames, which move faster through the video sequence. In addition, the motion blur may provide a useful visual cue to the speed at which the video is being viewed.

4.4 Clip-art

In our multiresolution video editor, the user may load video fragments, scale them, arrange them spatially with respect to each other, and preview how they will look together. These input fragments may be thought of as "video clip-art" in the sense that the user constructs the final product as a composite of these elements.

Since the final composition can take a long time to construct, our application provides a preview mode, which shows roughly how the final product will appear. The preview may differ from the final composite in that it performs compositing on the images currently being displayed rather than on the underlying video, which is potentially represented at a much higher resolution. (The degree to which the preview differs from the final composite corresponds exactly to the degree to which the "compositing assumption" [12] is violated.) When viewing the motion-blurred result of compositing two video sequences, there is a similar difference between the preview provided in our editor and the actual result of the compositing operation.

Figure 6 The application.

Once the desired effect is achieved, the user can press a button that translates, scales, and composites the various clip-art elements into a single multiresolution video, employing the operations described in the appendix. This video may be saved for subsequent viewing, or it may be combined with other elements as clip-art to form an even more elaborate multiresolution video.

4.5 Multiresolution video QuickTime VR

Apple Computer's QuickTime VR (QTVR) allows a user to explore an environment by looking from a fixed camera position out into a virtual world in any direction. Chen [3] proposes a potential augmentation of QTVR based on quadtrees that would provide two benefits. First, it would allow users to zoom into areas where there is more detail than in other areas. Second, it would reduce aliasing when the user zooms out. We implemented this idea, and extended it in the time dimension as well. Two simple modifications to multiresolution video were all that were required to achieve this "multiresolution video QuickTime VR" (MRVQTVR?!). First, we treat the video frames as panoramic images, periodic in the x direction. Second, we warp the displayed frames into cylindrical projections based on the view direction.

5 Results

We have implemented all of the operations of the previous section as part of a single prototype multiresolution video editing and viewing application, shown in Figure 6. Using the application, a user can zoom in and out of a video either spatially or temporally, pan across a scene, grab different video clips and move them around with respect to each other, play forward or backward, and use several sliders and dials to adjust the zoom factor, the speed of play through the video, the desired frame rate, and the current position in time.

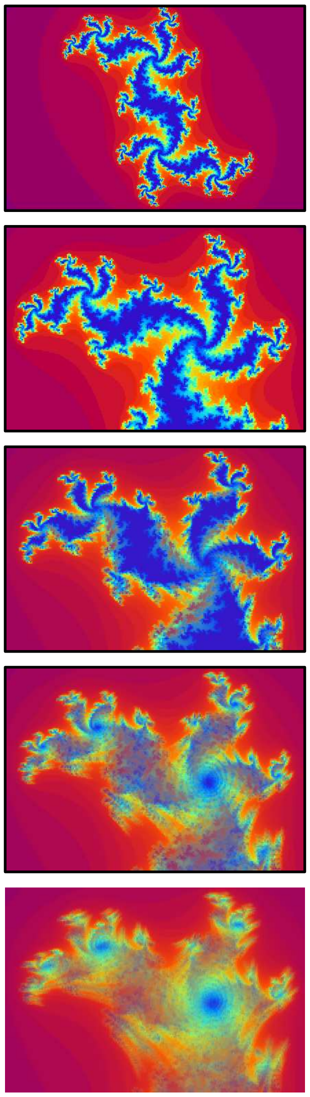


Figure 7 Julia set.

Figure 8 Van Gogh room.

Figure 7 illustrates how multiresolution video can be used for visualization of multiresolution data, in this case, an animation of the Julia set [4]. The data were generated procedurally, with higher spatial resolution in places of higher detail, as described in Section 3.1. The top three cells show increasing spatial detail, and the lower two cells show increasing "motion blur."

Figure 8 shows the result of arranging and compositing the many "clip-art" elements from the work area of the application shown in Figure 6 into a single multiresolution video, and then viewing this video at different spatial and temporal resolutions. (Apologies to Vincent Van Gogh.)

Figure 9 shows *wind stress*, the force exerted by wind over the earth's surface, measured for 2000 days over the Pacific Ocean by the National Oceanographic and Atmospheric Administration (NOAA). Wind stress is a vector quantity, which we encoded in multiresolution video using hue for direction and value for magnitude. The left image shows a leaf time node (reflecting a single day's measurements), while the right image shows the root time node (reflecting the average wind stress over the 2000-day period). Note the emergence of the dark continents in the right image, which reveals the generally smaller magnitude of wind stress over land.

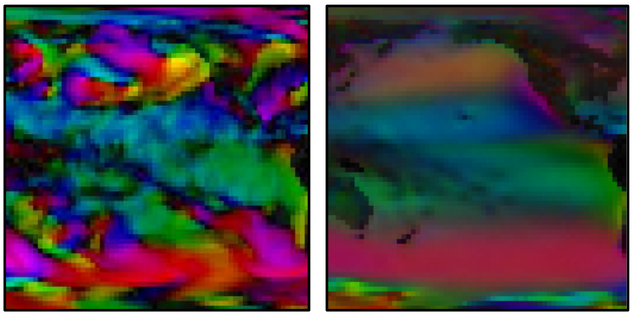


Figure 9 Wind stress over the Pacific Ocean.

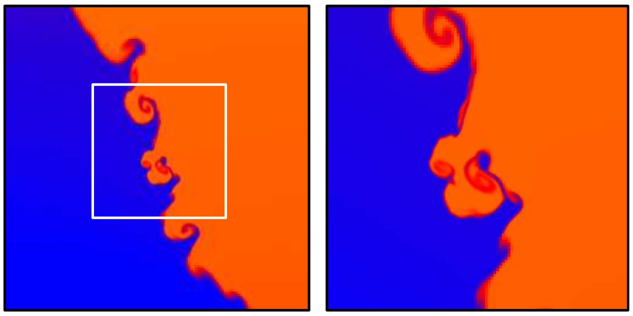


Figure 10 Fluid dynamics simulation.

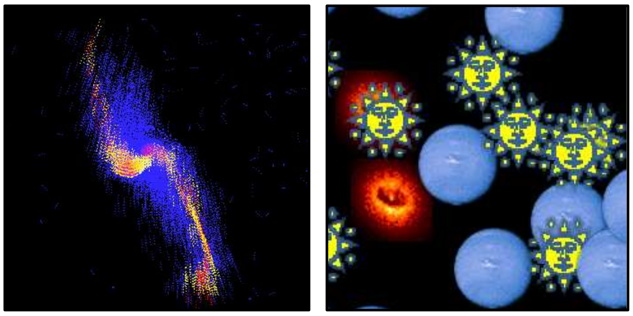


Figure 11 Astrophysical simulation of a galaxy.

The left side of Figure 10 shows a frame from a computational fluid dynamics simulation in which two fluids (one heavy, one light) interact in a closed tank. The simulation method [6] adaptively refines its sample grid in regions where the function is spatially complex, so the resolution of the data is higher at the interface between the two fluids than it is in the large, constant regions containing just one fluid. This refinement also occurs in time, providing higher temporal resolution in areas that are changing rapidly. The right image shows a close-up of the boxed region in the left image.

One more scientific visualization is shown in Figure 11. In this simulation, a galaxy is swept about a cluster of other astronomical bodies and is eventually ripped apart by their gravitational forces. The left image shows a close-up—late in the entire simulation— focused on the galaxy. The right image shows an even closer view of a single frame in which some whimsical high-resolution detail has been added. (However, creating the entire video sequence at this level of detail would be prohibitively expensive.)

Finally, Figure 12 shows a QTVR panoramic image that we have adapted for use with multiresolution video QuickTime VR. Over the picture frame on the wall we have composited the entire Van Gogh room video from Figure 8.

Figure 12 Panoramic image for multiresolution video QuickTime VR, and two views in the scene.

Figure	*Video*	*Disk Size*	*Memory Size*	*Unires Size*
7	Julia set	23,049	58,926	67,109
8	Van Gogh	46,738	98,798	34,359,730
9	Wind stress	68,566	134,201	33,554
10	Fluids	40,091	106,745	536,870
11	Galaxy	37,222	315,098	137,438,953
12	Panorama	47,723	100,804	2,199,023,256

Table 1 Sizes (in Kb) of some example multiresolution video clips.

Table 1 reports information about the storage space for the examples in Figures 7–12. The "Disk Size" column gives the total amount of space required to store the entire structure on disk, with averages and pointers included, after it has been compressed without loss using a Lempel-Ziv compressor [20]. The next column, "Memory Size" gives the total space required in memory, including all averages, pointers, and flags. The "Unires Size" column reports the total space that would be required to store the raw RGBA pixel values, assuming the entire video had been expanded to its highest spatial and temporal resolution present anywhere in the multiresolution video, but not including spatial or temporal averages. With the exception of the wind stress data, all of the video clips were smaller (in several cases much, much smaller) in the multiresolution video format than they would be in a uniresolution format, despite the overhead of the spatial and temporal averages. The wind stress data was difficult to consolidate because it has very little spatial or temporal coherence. The galaxy data compressed very well on disk because all of the colors stored in our structure (most of which were black) were selected from a small palette of very few colors.

6 Future work

This investigation of multiresolution video suggests a number of areas for future work:

User-interface paradigms. As in the multiresolution image work of Berman *et al.* [2], there is an important user-interface issue to be addressed: How does the user know when there is more spatial or temporal detail present in some part of the video? We have considered changing the cursor in areas where there is more spatial detail present than is currently being displayed. Perhaps a timeline showing a graph of the amount of temporal detail present in different parts of the video would address the corresponding temporal problem.

Environment mapping. Multiresolution video could be used for environment maps that change in time, allowing, for example, the rendering of a crystal glass, with animated objects in the environment reflecting in the glass. One benefit of using a multiresolution representation is that as the viewpoint and curvature of the glass surface vary, an accurate rendering may require more or less information from the surrounding environment.

Better compression. We currently require the up-links in our representation to point to a time-ancestor, primarily because coherence is fairly easy to discover this way. However, by relaxing this restriction—that is, by allowing up-links to point to any other place in the structure—we might be able to achieve much better compression, particularly for areas that have spatially repeating patterns. Unfortunately, finding the optimal set of up-links in this more general setting could be very expensive.

Spatial and temporal antialiasing. So far we only have used box-basis functions to represent the colors in multiresolution video. When the user zooms in to view a region at a higher spatial resolution than is present in the frame, large blocky pixels are displayed. Furthermore, if the user zooms in in time to view frames at higher temporal detail than is present in the video sequence, the motion is choppy. It would be interesting to explore the use of higher-order filters to produce smoother interpolations when the user views regions at higher resolution than is represented.

Acknowledgements

We would like to thank David Herbstman and David Simons for useful discussions during the initial phase of the project; Richard Anderson, Ronen Barzel, and Richard Ladner for valuable advice along the way; Neal Katz and Tom Quinn for the astronomical simulation; Randy LeVeque for the computational fluid dynamics; Xuri Yu for the wind stress data; Sean Anderson for the bee model; and Lightscape Technologies for the hotel lobby panorama.

This work was supported by an Alfred P. Sloan Research Fellowship (BR-3495), an NSF Presidential Faculty Fellow award (CCR-9553199), an ONR Young Investigator award (N00014-95-1-0728), an Intel Graduate Research Fellowship, and industrial gifts from Interval, Microsoft, and Xerox.

A Algorithms for combining video clips

This appendix describes a set of linear-time algorithms for translating, scaling and compositing multiresolution video sequences. Such operations are useful, for example, in the video clip-art application described in Section 4.4.

A.1 Translating a video clip

When combining video sequences, the various elements may need to be registered with respect to one another, requiring that they be translated and scaled within their own coordinate frames.

The basic operations of translation and scaling are well-understood for quadtrees [14]. However, as with drawing frames, we want these operations to take advantage of the temporal coherence encoded in the up-links of our data structure. For example, suppose we wanted to translate the fan and lamp video of Figure 3 a bit to the left. The regions of the video that contain the lamp should only be translated in the root node of the Time Tree, and all the time-children must somehow inherit that translation.

The following routine translates a multiresolution video clip, rooted at time node C, by an amount (dx, dy) at level ℓ_{tran} to produce a resulting Time Tree C'. In order to handle up-links, the routine is also passed the parent time node P of C, as well as the result P' of (previously) translating P by the given amount. In the top-level call to the procedure, the parameters P and P' are passed as NULL, and the Time Tree C' initially points to an image node containing just a single clear pixel. As the procedure writes its result into C', the translated image tree is developed (and padded with clear pixels as it is extended).

```
procedure TranslateTimeTree(C, C', P, P', dx, dy, ℓ_tran):
    TranslateFrame(C.frame, C'.frame, P.frame, P'.frame, dx, dy, ℓ_tran, 0, 0, 0)
    ComputeSpatialAverages(C'.frame)
    for each Half ∈ {Half1, Half2} of Timetree do
        if C.Half ≠ NULL then
            TranslateTimeTree(C.Half, C'.Half, C, C', dx, dy, ℓ_tran)
        end if
    end for
end procedure
```

The call to *ComputeSpatialAverages*() in the above procedure calculates average colors in the internal nodes of the image tree, using code similar to the *CreateFrame*() routine from Section 3.

The *TranslateFrame*() routine translates a single image tree c by an amount (dx, dy) at level ℓ_{tran}. In general the translation can cause large regions of constant color (leaves high in c) to be broken up across many nodes in the resulting tree c'. To handle the up-links, we must pass into the procedure the time-parent p of c, as well as the result p' of (previously) translating p. We also pass into the procedure arguments x, y and ℓ (initially all 0), which keep track of the location and level of node c.

```
procedure TranslateFrame(c, c', p, p', dx, dy, ℓ_tran, x, y, ℓ):
    if c.Type = COLOR or c.uplink.linked or ℓ_tran = ℓ then
        w ← 2^(ℓ_tran − ℓ)
        r ← Rectangle(w · x + dx, w · y + dy, w, w, ℓ_tran)
        PutRectInTree(c, c', p', r, 0, 0, 0)
    else
        for each i, j ∈ {0, 1} do
            TranslateFrame(c.child[i, j], c', p.child[i, j], p', dx, dy, ℓ_tran,
                                                    2x + i, 2y + j, ℓ + 1)
        end for
    end if
end procedure
```

The above procedure recursively descends image tree c, pausing to copy any "terminal" squares that it encounters as it goes. There are three kinds of terminal squares: large regions of constant color, subtrees that hang from level ℓ_{tran}, and up-links. In the first two cases, we copy the source from the original tree, whereas in the latter case we copy from the time-parent. A square's size and position are combined in a single structure *Rectangle*$(x, y, width, height, \ell_r)$, the coordinates of which are relative to the level ℓ_r. When the procedure finds one of these squares, it copies it into the resulting tree using the following procedure:

```
procedure PutRectInTree(c, c', p', r, x, y, ℓ):
    coverage ← CoverageType(r, x, y, ℓ)
    if coverage = COMPLETE then
        if c.type = COLOR or not c.uplink.linked then
            c' ← c
        else
            c' ← p'
            c'.uplink.linked ← TRUE
        end if
    else if coverage = PARTIAL then
        for each i, j ∈ {0, 1} do
            PutRectInTree(c, c'.child[i, j], p'.child[i, j], r, 2x + i, 2y + j, ℓ + 1)
        end for
    end if
end procedure
```

The above procedure recursively descends the result tree c' to find those nodes that are completely covered by the given rectangle, an approach reminiscent of Warnock's algorithm [4]. The function *CoverageType*(r, x, y, ℓ) returns a code indicating whether rectangle r completely covers, partially covers, or does not cover pixel (x, y) at level ℓ. For those nodes that are completely covered, *PutRectInTree*() copies either a color or a pointer, depending on the type of node being copied. If the node is a color, then the color is simply copied to its new position. If the node is a pointer but not an up-link, the routine copies the pointer, which essentially moves an entire subtree from the original tree. Finally, if the node is an up-link, the routine copies the corresponding pointer from the (already translated) time-parent p'. Thus, we have to descend the result tree c' and its time-parent p' in lock-step in the recursive call.

As with *DrawImage*(), the complexity of *TranslateFrame*() is related to the number of nodes it copies using *PutRectInTree*(). The latter procedure is dependent on the number of nodes it encounters when copying a rectangle. Since the former call makes a single pass over the source quadtree c, and the collective calls to the latter procedure make a single pass over the resulting image tree c', the overall complexity is proportional to the sum of the complexities of c and c'.

A.2 Scaling a video clip

Here, we consider scaling a Time Tree by some integer factors s_x in the x direction and s_y in y. Note that to shrink a video frame by any power of two in *both* x and y we simply insert more image parent nodes above the existing image root, filling in any new siblings with "clear." Conversely, to magnify a video frame by any power of two, we simply scale all other videos down by that factor, since we are only interested in their *relative* scales. Thus, scaling both x and y by any power of two is essentially free, and we can really think of the scales as being $s_x/2^\ell$ and $s_y/2^\ell$ for any (positive or negative) ℓ. For efficiency, it is best to divide both s_x and s_y by their greatest common power-of-two divisor.

The algorithms for scaling multiresolution video are structurally very similar to those for translation. The two main differences are that they copy scaled (rather than translated) versions of the source tree into the destination tree, and that they must descend down to the leaves of the image trees. We omit the specific pseudocode for scaling a video clip for lack of space. The time complexity of scaling is the same as translation: linear in the size of the input and output.

A.3 Compositing two video clips

The final operation addressed in this appendix is compositing two Time Trees A and B using the compositing operation op [12]:

```
function CompositeTimeTrees(A, B, op): returns TimeTree
    for each Half ∈ {Half1, Half2} do
        if A.Half = NULL and B.Half = NULL then
            Result.Half ← NULL
        else
            Ahalf ← A.Half
            Bhalf ← B.Half
            if Ahalf = NULL then Ahalf ← NewUplinkNode(A) end if
            if Bhalf = NULL then Bhalf ← NewUplinkNode(B) end if
            Result.Half ← CompositeTimeTrees(Ahalf, Bhalf, op)
        end if
    end for
    Result.frame ← CompositeFrames(A.frame, B.frame, FALSE, FALSE,
                                    Result.Half1.frame, Result.Half2.frame, op)
    return Result
end function
```

This function recursively traverses A and B in a bottom-up fashion, compositing child time nodes first, then their parents. If one of A or B has more temporal resolution than the other, then a temporary node is created by the function *NewUplinkNode*(). Invoking this function with the argument A creates a new *TimeNode* containing a single *ImageNode*, each of whose four children is an up-link pointing to its "time-parent" in A.

The following function works on two image trees a and b, taking a pair of arguments aUp and bUp that are set to FALSE in the top-level call; these flags are used to keep track of whether a and b are really parts of a time-parent. The function also takes a pair of arguments c_1 and c_2 that are the time-children of the resulting tree. In order to pass c_1 and c_2, the *CompositeTimeTrees*() function must have already computed these time-children, which is why it makes a bottom-up traversal of the Time Tree.

```
function CompositeFrames(a, b, aUp, bUp, c1, c2, op): returns ImageNode
  if a.uplink.linked then aUp ← TRUE end if
  if b.uplink.linked then bUp ← TRUE end if
  if aUp and bUp then return NULL end if
  if a.Type = COLOR and b.Type = COLOR then
    if c1 = NULL or c2 = NULL then
      return ImageNode(COLOR, CompositePixels(a, b, op))
    else
      return ImageNode(COLOR, Average(c1.avgcolor, c2.avgcolor))
    end if
  end if
  for each i, j ∈ {0, 1} do
    result.child[i, j] ← CompositeFrames(GC(a, i, j), GC(b, i, j),
                                         aUp, bUp, GC(c1, i, j), GC(c2, i, j), op)
  end for
  result.avgcolor ← AverageChildColors(result)
  return result
end function
```

We composite two image trees by traversing them recursively, in lock-step, compositing any leaf nodes. Child colors are propagated up to parents to construct internal averages. The helper function *GC*() (for "*GetChild*" or "*GetColor*") simply returns its argument node if it is a color, or the requested child if it is a subtree.

There are two subtleties to this algorithm. The first is that when the routine finds some region of the result for which both a and b are up-links (or subtrees of up-links), then it can assume that the result will be an up-link as well; in this case it simply returns NULL. Later, after all of the frames in the Time Tree have been composited, we invoke a simple function that traverses the Time Tree once, replacing all NULL pointers with the appropriate up-link. (This assignment cannot occur in *CompositeFrames*() because the nodes to which the up-links will point have not been computed yet.) The second subtlety is that if time-child c_1 or c_2 is NULL it means that the resulting image node has no time-children: either the node is part of an image tree that hangs from a leaf of the Time Tree, or its children are up-links. In either case we perform the compositing operation. If, on the other hand, c_1 and c_2 exist, then we are working on an internal node in the Time Tree and we can simply average c_1 and c_2.

The compositing operation described in this section creates a new Time Tree that uses up-links to take advantage of any temporal coherence in the resulting video. Since this resulting Time Tree is built using two bottom-up traversals, the complexity of creating it is linear in the size of the input trees.

References

[1] J. Adams, R. Garcia, B. Gross, J. Hack, D. Haidvogel, and V. Pizzo. Applications of multigrid software in the atmospheric sciences. *Monthly Weather Review*, 120(7):1447–1458, July 1992.

[2] Deborah F. Berman, Jason T. Bartell, and David H. Salesin. Multiresolution painting and compositing. In *Proceedings of SIGGRAPH '94*, Computer Graphics Proceedings, Annual Conference Series, pages 85–90, July 1994.

[3] Shenchang Eric Chen. Quicktime VR—an image-based approach to virtual environment navigation. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 29–38, August 1995.

[4] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Prentice-Hall, 1990.

[5] D. Le Gall. MPEG: A video compression standard for multimedia applications. *CACM*, 34(4):46–58, April 1991.

[6] Randy LeVeque and Marsha Berger. AMRCLAW: adaptive mesh refinement + CLAWPACK. `http://www.amath.washington.edu/~rjl/amrclaw/`

[7] A. S. Lewis and G. Knowles. Video compression using 3D wavelet transforms. *Electronics Letters*, 26(6):396–398, 15 March 1990.

[8] S. McCormick and U. Rude. A finite volume convergence theory for the fast adaptive composite grid methods. *Applied Numerical Mathematics*, 14(1–3):91–103, May 1994.

[9] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 39–46, August 1995.

[10] Arun N. Netravali and Barry G. Haskell. *Digital Pictures*. Plenum Press, New York, 1988.

[11] Ken Perlin and Luiz Velho. Live paint: Painting with procedural multiscale textures. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 153–160, August 1995.

[12] Thomas Porter and Tom Duff. Compositing digital images. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259, July 1984.

[13] Steven Radecki. *Multimedia With Quicktime*. Academic Press, 1993. ISBN 0-12-574750-0.

[14] Hanan Samet. *Applications of Spatial Data Structures*. Addison-Wesley, Reading, Massachusetts, 1990.

[15] P. S. Sathyamurthy and S. V. Patankar. Block-correction-based multigrid method for fluid flow problems. *Numerical Heat Transfer, Part B (Fundamentals)*, 25(4):375–94, June 1994.

[16] I. Suisalu and E. Saar. An adaptive multigrid solver for high-resolution cosmological simulations. *Monthly Notices of the Royal Astronomical Society*, 274(1):287–299, May 1995.

[17] Jonathan Swartz and Brian C. Smith. A resolution independent video language. In *ACM Multimedia 95*, pages 179–188. ACM, Addison-Wesley, November 1995.

[18] S. L. Tanimoto and Theo Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104–119, June 1975.

[19] Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 1–11, July 1983.

[20] L. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform.Theory, Vol.IT-23*, (3), May 1977.