Digital Cubism, Part 2

Andrew Glassner

In my last column I discussed a problem faced by filmmakers who want to show simultaneous action happening in different places. If two or more locations are near one another, we might be able to catch them all with a single camera. But if the actions are happening far away from one another, or a single image doesn't capture everything the way we'd prefer, we have a challenge.

One common solution is to use a split-screen effect, where two or more images are shown simultaneously and separated by a black bar, as Figure 1c shows. Another common technique is to edit the piece together in such a way that the audience realizes that what they're viewing sequentially (or in alternation) represents action that should be interpreted as simultaneous in the world of the story.

I wanted to find another option, which would allow me to show disparate events in a single frame, imaged by different cameras in different places, yet all smoothly joined together visually. I developed a technique that I call the multicamera collage, or MCC. Figure 1d shows how MCC could handle the split-screen example.

Figure 2 recaps the basic idea behind creating Figure 1d. The following summary is a brief overview of the approach; you can find more details and discussion in my May/June 2004 column.

To make a single image using the MCC, you first render several views of your scene. You can make each rendering with any camera located in any position; the cameras can even be custom camera shaders if they're useful to you. Figures 2a through 2c show three such images.

After running a program that processes the images, load the resulting pictures up into any image-editing program that supports layers, such as Adobe Photoshop or Corel Painter. Then use any selection tool you like to choose the pieces of each image you want in the final frame. Place a black background behind your images to fill in the holes. You can choose from the images any way you like: pick one big blob from this camera, five little blobs from another, and a circle from the third. There's no restriction on how many image pieces you can use for each camera, or what the shapes of those pieces should be. The result is a collage. Figure 2d shows one possible collage for our example. At this point you can also create an optional region image, which I'll discuss later.

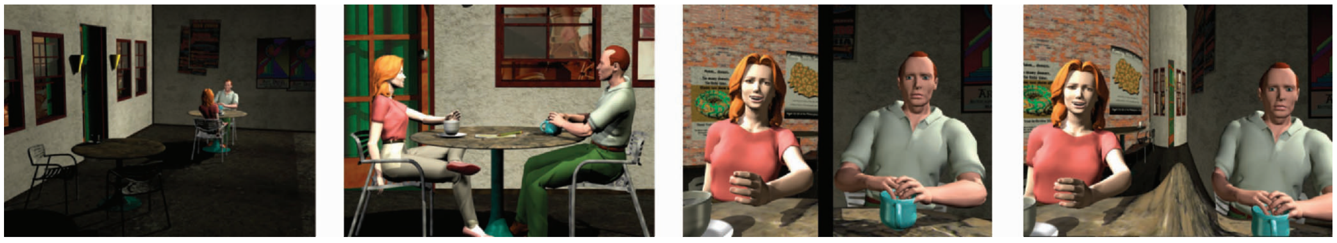
That's it for your involvement. Just run another program that processes the collage (as well as some other files automatically created when you made your rendered images) and then hand it back to your 3D modeling or rendering system to create a new, smoothly interpolated image. Figure 2e shows the results for our collage in Figure 2d.

If you're working in animation, you can create as many keyframes this way as you like. Each keyframe can have a different number of cameras, a different number of regions, and so on. In fact, keyframes need not even have any cameras in common. Just choose the cameras you want for each frame and draw the collages that you like, and the frames in between will interpolate smoothly just like the still images themselves.

Now that we've quickly reviewed the workflow, let's open the hood and look inside to see what programs are involved and how they work.

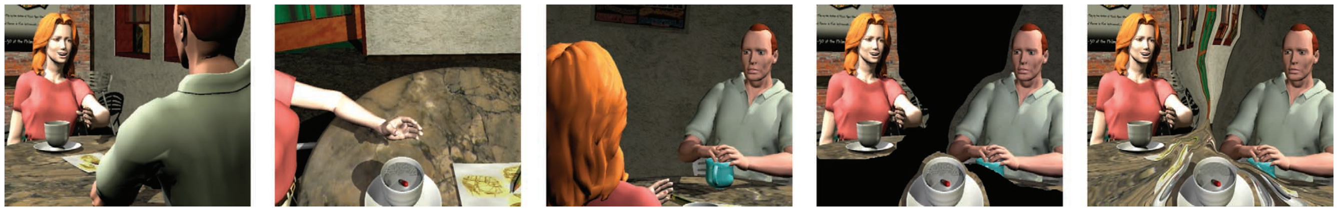
Under the hood

The MCC is a collection of five programs: two shaders and three standalone programs. The shaders capture



(a) (b) (c) (d)

1 (a, b) Two people having a chat in their local cafe. (c) Split-screen version of the scene. (d) Multicamera collage version of the same shot.



(a) (b) (c) (d) (e)

2 **Recapping the multicamera collage process. (a–c) Three shots of a couple at a cafe. (d) Collage made from these images. (e) Interpolated result from multicamera collage.**

data from the rendering program when making the initial images, and supply new data for rerendering. The standalone programs prepare the images for collaging, process individual collages, and build in-between frames for animation. Let's run through the general flow again, pointing out where each of these programs is involved. For simplicity, I'll assume we're just making a single frame, rather than an animated sequence.

Start off by rendering your scene from as many different points of view as you like, using your 3D system's built-in cameras. During this process, you tell the program to use a custom lens shader, which I call the *lens writer*. This writes an auxiliary data file during the rendering process, saving information about each screen ray fired by the renderer. When the frames are rendered, run them through the *camera tagger* program. This modifies the pixels in each rendered image so that we can later identify which camera was used to create each pixel.

Load up these modified images into any image-editing program that supports layers. Behind them, place a black background. Using a selection tool, cut away the parts of each image that you don't want in the final, and save the result as a collage. Now run another program—called the *lens builder*—which creates a new file that provides one ray per pixel to guide the rerendering process.

Now open up your 3D scene again, but tell the renderer to use a camera shader I call the *lens reader*, and render the image. It reads the file created by the lens builder to build rays that generate a new image that looks like your collage, but with the black regions smoothly filled in with scene views.

If you're making an animated sequence, you'll probably want to make several collage files at different points in the sequence. You'd then run the *in-betweener*, which creates one rerendering file per output frame. The program smoothly interpolates the keyframe collages, producing a smooth animation with your changing cameras.

Let's look at the details of these programs.

Taking shade

Shaders were invented by Rob Cook to break open the traditional rendering pipeline. Shaders are little nuggets of programming that tell the rendering system what to do at various steps along the way of creating an image. The MCC depends on a pair of custom (but easy to write) lens shaders.

Most ray tracers let you plug a lens shader into the pipeline. After the system has built an eye ray, but before it starts using it to find illumination from the environment, it calls your lens shader. This little piece of code can do anything: compute the square root of pi, simulate the flow of wind over a wing, or more commonly, adjust the ray parameters. If there are several lens shaders, they're simply called one after another in the order you specify.

Once a lens shader finishes its work, the system uses the potentially modified eye ray to find the color associated with the ray's corresponding spot on the film. If the system wants to perform antialiasing, motion blur, depth of field, or any other process that requires more eye rays, it simply makes and traces them as usual. The lens shader is called just before the ray goes into the world.

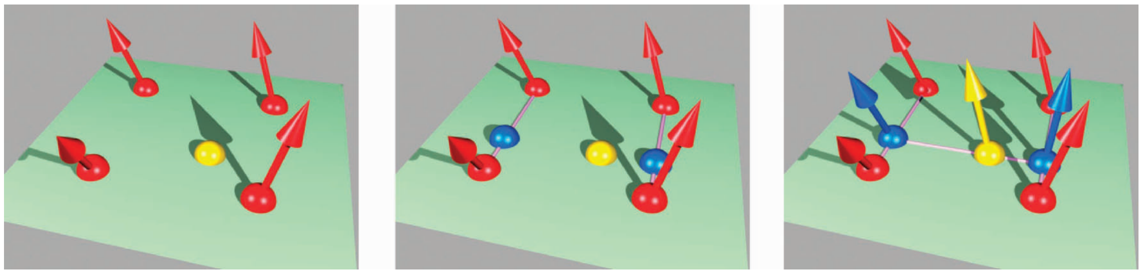
Len shaders typically have at least two sections: the initialization routine and the shader proper. The initialization procedure is usually called at the start of each new frame, and the shader itself once per ray.

The MCC uses two lens shaders. The *lens writer* is used when you're rendering the initial images of your scene, which you'll later use to create the collage. The initialization routine for the lens writer creates a new file, whose name is a combination of the camera number and the frame number. For example, if we're rendering frame 5 from camera 3, the file might be called frame0005camera03.txt. The initialization routine saves the name of this file internally, writes out the image's width and height, and closes the file. In this discussion, I'll assume that all the files are plain text. This is a good choice during development and debugging, since you can open the files and read them yourself. When everything is working, it's reasonable to use a binary format, which is typically smaller than text and faster to process.

Let's return to our pipeline. During the image's rendering, each time the lens writer is called, it simply opens up the file, appends to the end a plain-text line containing the ray's 2D screen location, 3D origin point, and 3D direction vector, and then closes the file. Then it returns and lets the system trace the ray as usual.

For convenience in later processing, I configure the renderer during this step to create exactly one ray per pixel while rendering these initial images. So if the image has dimensions of w by h pixels, there are $wh + 1$ lines in the text file (the first line declares the image width and height). Of course this means that the image can have jaggies and other aliasing artifacts, but these images are just for use in the collage and the artifacts never show up in the final, rerendered output.

Let's jump to the end of the MCC process, when we're



3 Schematic view of how camera rays are combined to create a new, interpolated ray for rerendering. Yellow rays at the corners are our input. I interpolate pairs to find the blue rays, and then interpolate them to find the red ray.

rendering the new, interpolated image. The *lens reader* carries out the lens writer's process roughly in reverse. It assumes that the lens builder program has been run already. That program creates one file per frame, in the same format as that of the lens writer: one line per ray, containing the ray's screen location, origin, and direction.

When the lens reader is initialized, it opens the file for that frame (for example, frame 51 might be rerender-frame0051.txt). It reads in all of the ray descriptions and saves them in local memory, then closes the file.

From now on, each time the lens reader is called, it uses its input ray's 2D screen location to access the database of rays that it read. It uses the screen coordinates to interpolate the input rays, as Figure 3 shows. I blend the ray positions using linear interpolation, and the directions using direction interpolation (which I'll discuss later). I overwrite the origin point and direction vector of the screen ray created by the system with this computed ray, and then return it to the rendering system, which follows it into the scene.

So although the renderer is generating eye rays and setting them up for rendering, the lens reader overwrites those rays before they head out into the environment. The renderer doesn't know that, of course, so it antialiases and does motion blur and everything else as usual. The result is a picture in which every ray has been independently placed and directed by the lens reader to create the smooth scene we desire.

Let's now look at the standalone programs I've mentioned.

Camera tagger

The job of the *camera tagger* is very simple: It overwrites a few bits in every pixel of your original rendered images with the number of the camera that created that image.

For example, suppose we open a rendered frame called frame0013camera09.tif, which tells us that the image was rendered by camera 9. The camera tagger reads each pixel in the image one at a time, and puts the number 9 into the color value of every pixel.

I think of the low-order 2 bits of each color component as together forming a single 6-bit binary number. The 2 low-order bits of blue correspond to the 2 highest-order bits in the number, the 2 low-order bits of green are in the middle, and the 2 low-order bits of red make up the lowest-order bits of the number. In our example, camera 9 corresponds to binary 001001. So I'd put 00 in the 2 low-order bits of the blue color, 10 in the 2 low-order bits of green, and 01 in the 2 low-order bits of red, and save the pixel. I do this for every pixel in the image.

Of course, this can introduce some visible artifacts into the image, such as color shifting and banding. But these modified pixels are only used in the collaging process, and don't appear in the final, rerendered image. As long as the image is still legible enough to be useful in making the collage, no harm is done.

Since I'm using 6 bits, I can represent 63 cameras (as we'll see later, the number 0 is reserved to mean *no camera*). I've never wanted that many cameras in a single frame, but if you wanted more, you could just use more bits. Taking 3 bits per channel, for example, would let you encode 511 different cameras.

Lens remapper

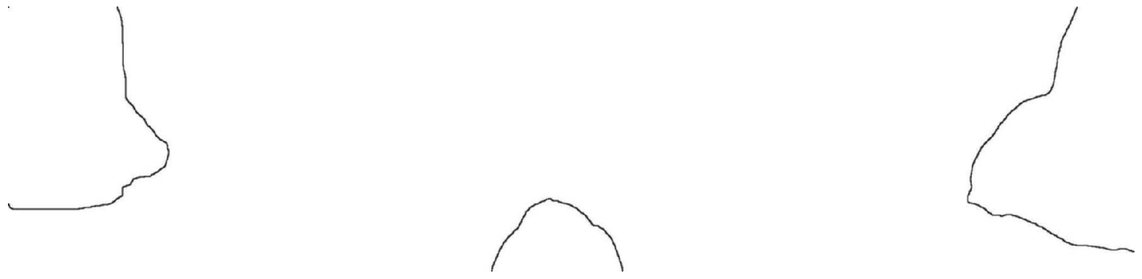
The collage file is an image that contains two types of pixels: those that are from the black background (that is, they have 0 in all three color channels), and those that are from a camera (so at least one of the lowest 2 bits in at least one of the color channels is a 1). Note that the background black is different from black in one of the camera images. For example, if camera 6 was looking at a pitch-black object and thus originally wrote (0, 0, 0) in the RGB channels for some pixels, those pixels would become (2, 1, 0) after the camera tagger went to work (red = 10, green = 01, blue = 00, which go together in the order 000110, forming the binary number 6). So pixels with the color (0, 0, 0) mean no camera, and those are the ones we'll fill in.

The lens remapper's job is to create a new input file for the lens reader, giving it the location and direction of one ray in the center of each pixel. The lens reader (our lens shader in the final step) will interpolate these rays as necessary for screen rays generated by the renderer.

The lens remapper is the largest program in the system, and it's where we create the visual look of the rerendered image.

The lens remapper starts by assuming that the pixels in the collage that have a camera number associated with them are pixels you want to see in the final image. That is, the regions you selected to keep in the collage should also appear in the output. So if a given pixel in the collage is labeled with camera 5, then in the output file the ray for that pixel will come from the text file stored with camera 5, which gives the exact origin and direction of the ray that was used for that pixel by that camera.

On the other hand, pixels that have no camera asso-


4 **Edge images from the cameras in Figure 2.**

ciated with them receive a computed ray. The computation is designed so that the rays sample the visual field smoothly over the entire image. I do this by finding the nearest pixel from each camera, and coming up with a weight for those cameras, so the nearer ones make more of a contribution than those far away. Then I go to the text files for each camera, weight the rays associated with those pixels by the given amount, and then add them. The key is to remember that the text files let us retrieve the ray information associated with every camera and every pixel.

The first step in the process is to find the edges for each camera, as Figure 4 shows. The edge-finder is pretty straightforward. It marches through the collage image, and calls a routine that returns true or false, identifying whether the pixel is considered on an edge.

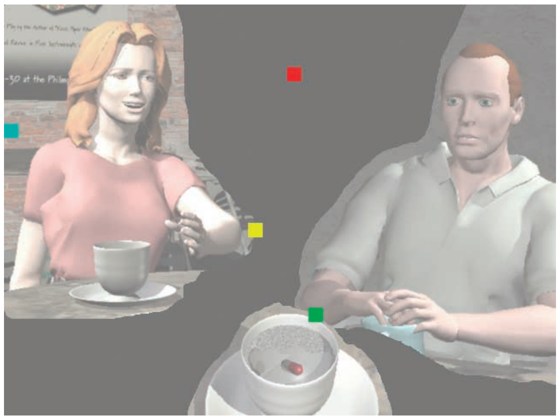
The identifying routine begins by extracting the pixel's camera number. If the camera is 0 (that is, the pixel is from the black background) the routine returns false. Continuing on, if the pixel is on the outermost border of the collage, the routine returns true. Otherwise, the routine compares the pixel's camera number to the camera number of its eight neighbors. If any of those eight cameras are different, the routine returns true; otherwise, it returns false. Figure 5 shows these tests visually.

When I find an edge pixel, I add it to a list of edge pixels for its associated camera. So later on, when I want to look through all the edge pixels for any camera, I can easily access them all without any additional processing.

With the edge lists completed, I now start computing weights. Let's say that $A_i(x,y)$ is the ray origin saved at pixel (x,y) for camera i, and $\mathbf{D}_i(x,y)$ is the ray direction saved at pixel (x,y) for camera i. Let's package this information together as a ray $R_i(x,y)$. The weighting process involves finding a set of scalar weights that I can apply to these points and vectors to create a new ray. For each camera i, I find a scalar weight α_i, and a particular pixel (x_i, y_i), so the new ray R drawn from C cameras in the collage is

$$R(x,y)=\sum_{i=1}^{C}\alpha_i R_i(x_i,y_i)$$

In other words, to find the ray at (x, y), I find a pixel (x_i, y_i) for each camera i, get the ray information R_i for that pixel from the text file for that camera, weight it by α_i, and add it into the running total for the new ray. Just how the rays are scaled and added up requires a little discussion, which I'll get back to in a bit. For now, let's concentrate on the problems of finding the pixel we want for each camera, and then the right weight for it.


5 **Finding the edges. Red pixel: If a pixel is in the background, it's not on an edge. Yellow pixel: If a pixel is in a camera but on the border, it's an edge. Cyan pixel: If a pixel is on the border of a camera region and black, it's an edge. Green pixel: If a pixel is on the border of two camera regions, it's an edge.**

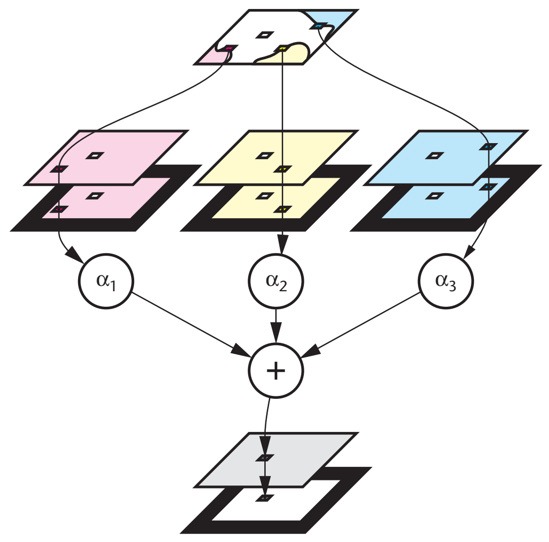
I compute all of this information across the whole picture before I actually build the new rays, because I process the weights after they've all been calculated.

One way to think of this is that if the input image has dimensions $w \times h$, then I build C 2D data structures (one for each camera). Each element of these data structures contains a floating-point number (the weight α) and two integers identifying the pixel (x, y) that should be used from that camera to contribute to the pixel where it's located.

Let's process a single pixel P from the collage, located at (x_p, y_p). I first retrieve the pixel from the collage and look for its camera value. If it has one (that is, the camera number encoded in those low bits is nonzero), then I know I want that output pixel to have exactly the same ray as the one used by its camera. Let's say it was made by camera c. Then I set $\alpha_c = 1$ and all the other α_i to 0. I also set the pixel coordinates in all data structures for all the cameras to (x_p, y_p). The result is that when I go to build the ray for this pixel later, the ray used by camera c at this pixel will be what I compute. Then I move on to the next pixel.

On the other hand, if the pixel has no camera, then I need to work harder. First, I scan through the edge list for each camera, and locate the edge pixel that's nearest to P. The coordinates of that pixel get written into the data structure for each camera. Figure 6 (next page) shows this idea.

Now I need to compute the weights. For this, I use the



6 **Schematic of the weighting process.**

multipoint weighting technique I described in my March/April 2001 column, "Tricks of the Trade" (see the "Further Reading" sidebar for pointers to book versions of this and other columns). To summarize this idea quickly, let's call each camera pixel c_i for $i = [1, C]$. I start by finding the distance from P to each of these pixels:

$$d_i = |P - c_i|$$

Now I normalize these so they sum to 1. I call the normalized distances g_i:

$$g_i = d_i / \sum_{j=1}^{C} d_j$$

Now I find a strength v_i that tells me how much each of the pixels is affecting P:

$$v_i = (1 - g_i) \sum_{\substack{j=1 \\ j \neq i}}^{C} g_j$$

To get our weights w_i, we just normalize the v_i:

$$w_i = v_i / \sum_{j=1}^{C} g_j$$

Although the weights produced by this method are continuous across the image, they're not necessarily continuous in their first derivative. Figure 7 shows an example of the weight applied to a camera as we move from a region where that camera is present into the black zone next to it. Any first-derivative discontinuities in the weights could show up as artifacts in the final image.

To smooth things, I apply a little bit of filtering. Each filter runs on the weights for each camera independently. It's important to double-buffer these filters, which means that I process all of the weights and compute new values for the entire camera before I write them back.

First I blur the weights using a box filter of radius 4. This just means that for each black pixel P, I add all of the weights applied to the 9×9 block of pixels centered around P, divide by 81, and use that for the new value at P. (For boundary pixels, I only divide by the number of pixels that are actually in the image and get added to the average.)

Then I apply an ease curve to the weights. This is simply an S-shaped curve that smoothes the edges a bit more. Returning again to my March/April 2001 column, I use the ease curve described there with a premapped control value of 0.75 (see Figure 8). After this I blur with another box filter of radius 3, which smoothes out any remaining rough edges.

I then pass through each pixel one at a time, and renormalize the weights so that they sum up to 1 again. To do this, I just add up the weights and divide each one by that sum.

Now that I have everything I need, I just run through the pixels and apply the formula we saw at the start of

Further Reading

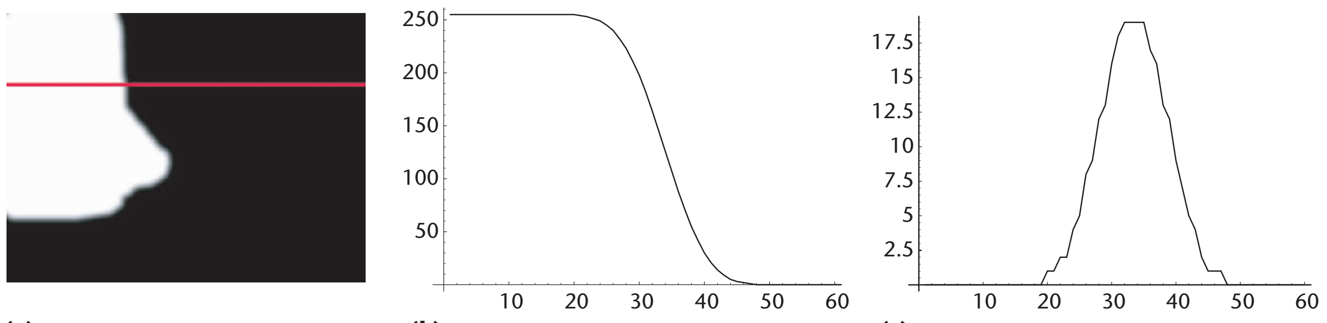
In my last column (see the May/June issue of *IEEE Computer Graphics and Applications*) I offered a variety of references on topics, from the optics of pinhole and complex cameras to the artistic school of Cubism. Please refer to that column for further details on any of those topics.

Shaders play a key role in this technique. The seminal paper on shaders is R.L. Cook's "Shade Trees," *Proc. Siggraph*, vol. 18, no. 3, 1984, pp. 223-231.

I've referred in the text to a number of my previous columns. My discussion of vector interpolation and the formula for computing circular interpolation appears in my article "Situation Normal," *IEEE Computer Graphics and Applications*, vol. 17, no. 2, Mar./Apr. 1997, pp. 83-87. My discussion of camera shutters appears in "An Open and Shut Case," *IEEE Computer Graphics and Applications*, vol. 19, no. 3, May/June 1999, pp. 82-92. Multipoint interpolation and the burp algorithm both appear in "Tricks of the Trade," *IEEE Computer Graphics and Applications*, vol. 21, no. 2, Mar./Apr. 2001, pp. 80-87. These columns are now all available in book form, where they have been revised and expanded. "Situation Normal" is chapter 5 of *Andrew Glassner's Notebook* (Morgan-Kaufmann, 1999), and the other two columns are chapter 1 and chapter 8, respectively, in *Andrew Glassner's Other Notebook* (AK Peters, 2002).

I wrote this implementation of the multicollage camera (MCC) with a variety of software tools. I wrote all of the programs in C# using Microsoft's .NET programming environment. To make the images, I used discreet's 3ds max 6 production system. I wrote my lens shaders for mental images' mental ray 4.3, which I used to create the rendered images. I created the collages in Adobe Photoshop CS.

I've been thinking about these ideas for a long time. I wrote my first implementation in 2000 when I worked at Microsoft Research. A summary of my work at that time is available under the following listing: A.S. Glassner, *Cubism and Cameras: Free-Form Optics for Computer Graphics*, tech. report MSR-TR-2000-05, Jan. 2000, available online at http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-2000-05).



7 **(a) Weight associated with this camera region. (b) Plot of the weight. (c) Derivative of Figure 7b.**

this section, weighting and combining the camera rays at each pixel to create a single new ray. I said that I'd return to how that's done later, and now the time has come.

Weighting and adding the ray's origins is a snap: I just scale the three coordinates by the weight and add them, just like interpolating any real numbers. Since the weights sum to 1, no further normalization is needed:

$$A(x,y)=\sum_{i=1}^{C}A_i(x_i,y_i)$$

The direction vectors are trickier. As I discussed in my March/April 1997 column, "Situation Normal," if we simply interpolate the vector coordinates independently and then renormalize the result (as we do in Phong shading), we don't quite get circular interpolation. What we want is to interpolate the direction vectors as vectors, not points. In that column I gave a formula for interpolating two unit vectors **P** and **R**:

$$\mathbf{Q}(\alpha)=\frac{\sin(\theta-\psi)}{\sin\theta}\mathbf{P}+\frac{\sin\psi}{\sin\theta}\mathbf{R}$$

where α is our interpolating variable that sweeps from 0 to 1, $\psi=\alpha\theta$, and θ is the angle between the two vectors: $\theta=\cos^{-1}(\mathbf{P}\cdot\mathbf{R})$. That's fine when only two vectors exist, but here we need to combine C vectors, which will usually be three or more.

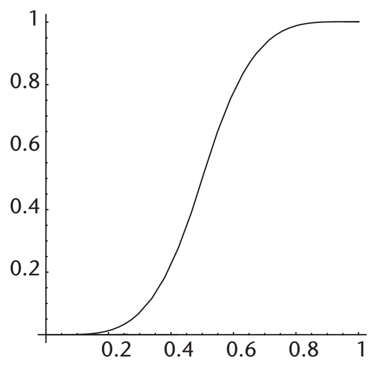
The solution comes from my March/April 1997 column, where I presented a method for bilinear uniform interpolation called *burp*. I'll summarize the essential details here. We want to find a linear sum P as

$$P=\sum_{i=0}^{n-1}\alpha_i p_i \quad \text{where} \quad \sum_{i=0}^{n-1}\alpha_i=1$$

using a series of two-point interpolations. We can decompose this result into several small pieces. We want to find $P = r_0$:

$$r_0=\sum_{i=0}^{n-1}\alpha_i p_i \quad \text{where} \quad \sum_{i=0}^{2}\alpha_i=1$$

We'll start by computing r_{n-1} and then working our way back up to r_0:



8 **Ease curve I apply to the weights.**

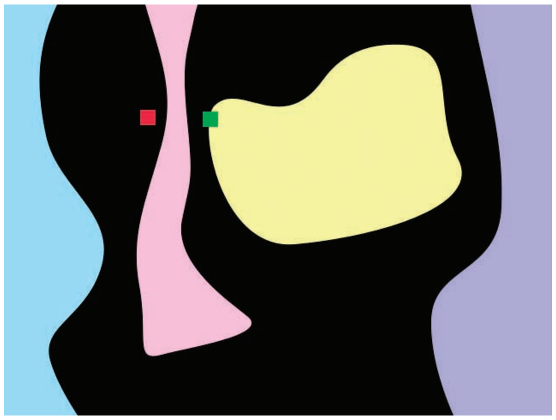
$$r_{n-1}=p_{n-1}$$
$$r_k=\frac{\alpha_k}{\beta_k}p_k+\frac{\beta_k-\alpha_k}{\beta_k}r_{k+1}$$
$$\beta_k=\sum_{j=k}^{n-1}\alpha_j$$

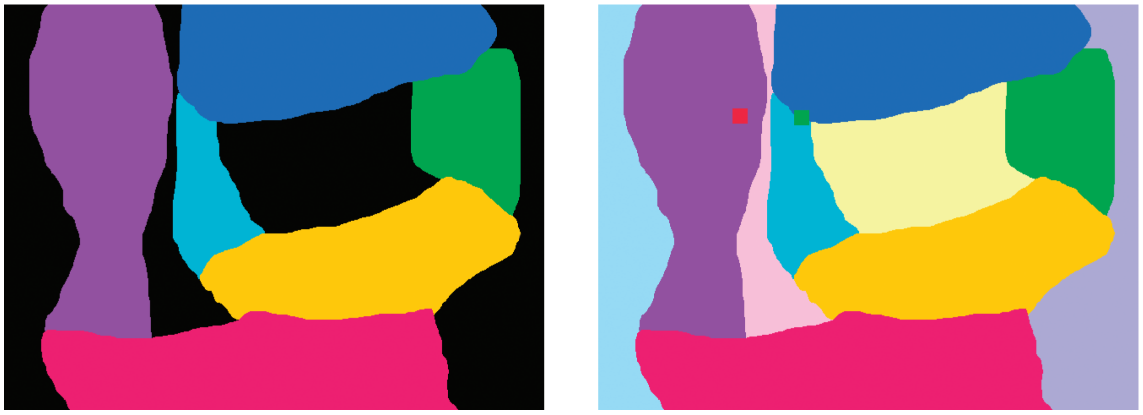
You can find more details on this technique, along with an example in the original article (see the "Further Reading" section). Now that I have both the ray origin and direction for this pixel, I simply write these six numbers out to a plain-text output file. The file has one line per ray, and one ray per pixel. This is the file that's read in by the lens reader shader to make the final image.

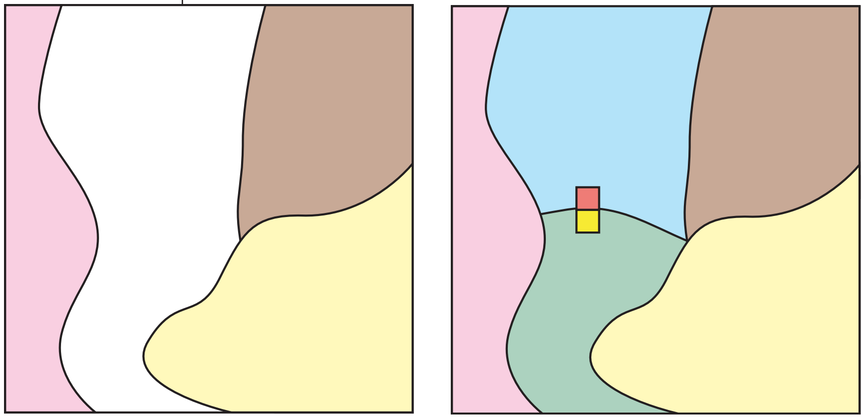
Regions

The previous algorithm does a pretty good job on some images, but it can make mistakes on others. The essential problem is that the algorithm finds the new ray by combining a ray from every camera. But there are times when we don't want all of the cameras to contribute to a pixel.

For example, consider Figure 9. This collage uses just a few simple colors to represent camera regions. I've denoted which pixel we're interested in computing weights for by coloring it red. A nearby camera, in yellow, would normally contribute its nearest pixel, shown in green. But in this collage, we probably don't want the


9 In this collage, we want to compute weights for the pixel marked in red. Although the pixel marked in green (belonging to the yellow camera) is near the red pixel, it probably shouldn't contribute to the red pixel at all.


10 Using hand-drawn regions to control which cameras contribute to which pixels in Figure 9.


11 Possible problem with regions. Camera 1 is on the left, camera 2 is in the upper right, and camera 3 is in the lower right. The upper and lower edges of this boundary join cameras 1 and 2 and 1 and 3, respectively. This would probably display a crease if the weights weren't smoothed.

yellow camera to contribute at all, since there's another camera between the red and green pixels. In this case, we'd like the yellow camera to make no contribution to the red pixel.

To bring this about, you can draw a *region image* when you make the collage image. Figure 10 shows a typical region image. It's just a bunch of areas of arbitrary shape, each filled with a different color.

When there's a region image available in the same directory as the collage image, I use it to guide the selection of pixels from each camera. When I start processing a pixel P, I first check to see what region it's in (I just use the color of that pixel in the region image). Now when I run through the list of edge pixels for each camera, I retrieve the region for each pixel, and if it's not the same as the region for P, I skip it. If I get to the end of the list and I haven't found any pixels from that camera in the same region as P, I just mark that camera as a nonparticipant for that pixel. If there's no region image, then implicitly every pixel in the picture is in the same region.

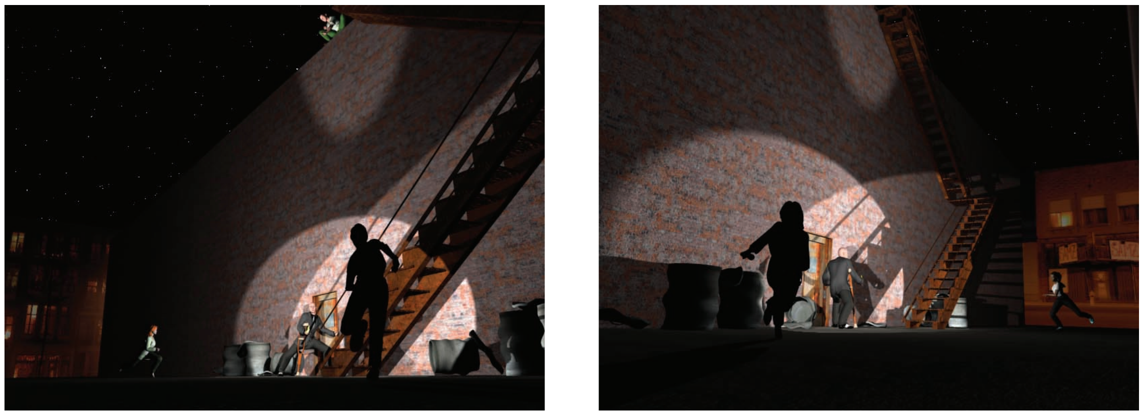
The shapes in the region image can have any shape or size, they can be filled with any color, and there can be any number of them. Just draw them as you like, but use a drawing tool that doesn't antialias your selection (otherwise those blended pixels will each be their own 1-pixel-large region).

Regions aren't a perfect solution, though, because they can introduce visible discontinuities in the final image. Figure 11 shows the boundary between two regions. In one region, we're interpolating cameras 1 and 2, and in the next, cameras 1 and 3. To minimize the effects of this problem, I use my eye when I draw regions, and try to avoid creating sharp edges. The filtering and blurring steps that I apply to the weights help smooth out this crease, but I'd rather not create it in the first place.

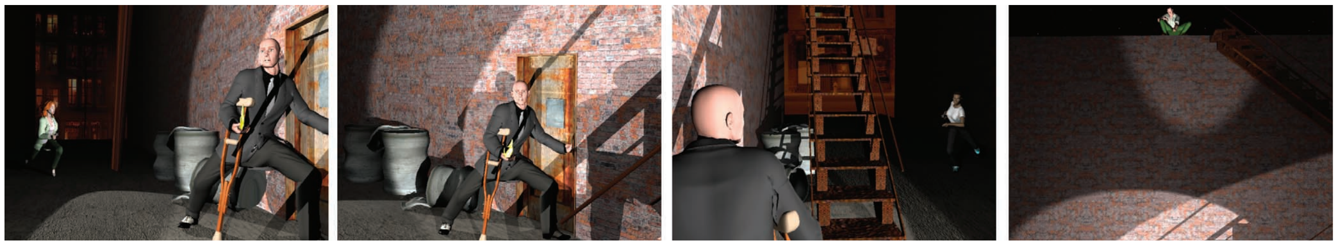
Details

I implemented this system using discreet's 3ds max 6 and mental images' mental ray 4.3 rendering system that ships with it. Mental ray is good at taking advantage of parallelism, and will make use of as many processors as it can find. On my home computer, I have two Pentium processors.

Running two processors at once caused me endless



12 Two views of a scene from my short film, *Rundown.*



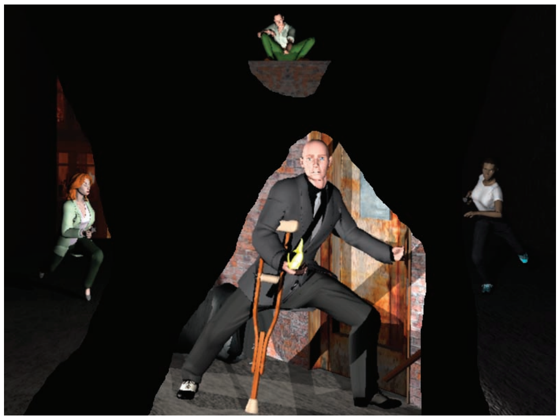
13 Four rendered images of the scene in Figure 12.

headaches while I was trying to get my lens writer shader to work, because different but parallel copies of the shader were trying to write to the same output file at the same time. Once I realized what was happening, I took the easy way out and put the code that writes to the file inside a locked region, which means that only one instance can be running at a time. The locked region opens the text file, writes a ray, then closes the file. It's inefficient and you can definitely see a slowdown, but because I'm only firing one ray per pixel, and this is just a testbed, this quick-and-dirty solution was okay.

It did mean that I had to precede each line with the (x, y) coordinates of the pixel being written, since they could be written in any sequence. The lens remapper uses those coordinates to make sure the correct data goes into the correct pixel.

Similarly, the lens reader initialization routine is locked, because I want to make sure that it finishes reading in the data before any rendering occurs. After that, there's no locking, since the reader is only looking up data from its database of per-pixel ray information, which remains constant.

It's important to use selection tools that don't antialias when building the collage and region images. The reason is that antialiasing versions of these tools typically blend nearby pixels to create a smooth edge. In the collage, such blending will mess up the camera values that we've written to our pixels, which will create extraneous cameras and misidentify pixels on the edge. In the region image, antialiased pixels will become tiny 1-pixel-large regions, each of their own color. The collage and region boundaries should be hard edged, and very likely will show ugly jaggies. As with the other image artifacts we've seen, this has no effect on the final image.



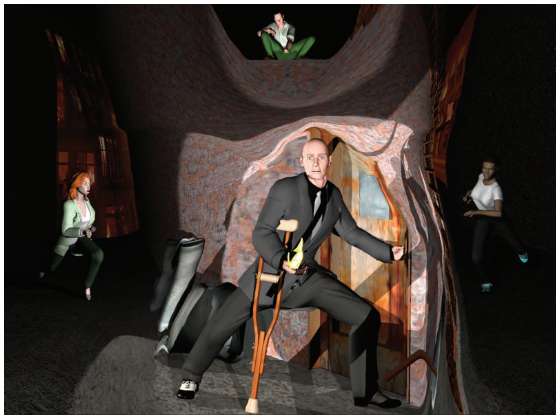
14 Collage built from the images in Figure 13.

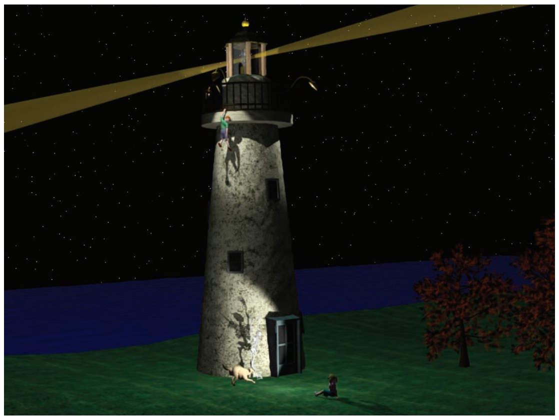
Examples

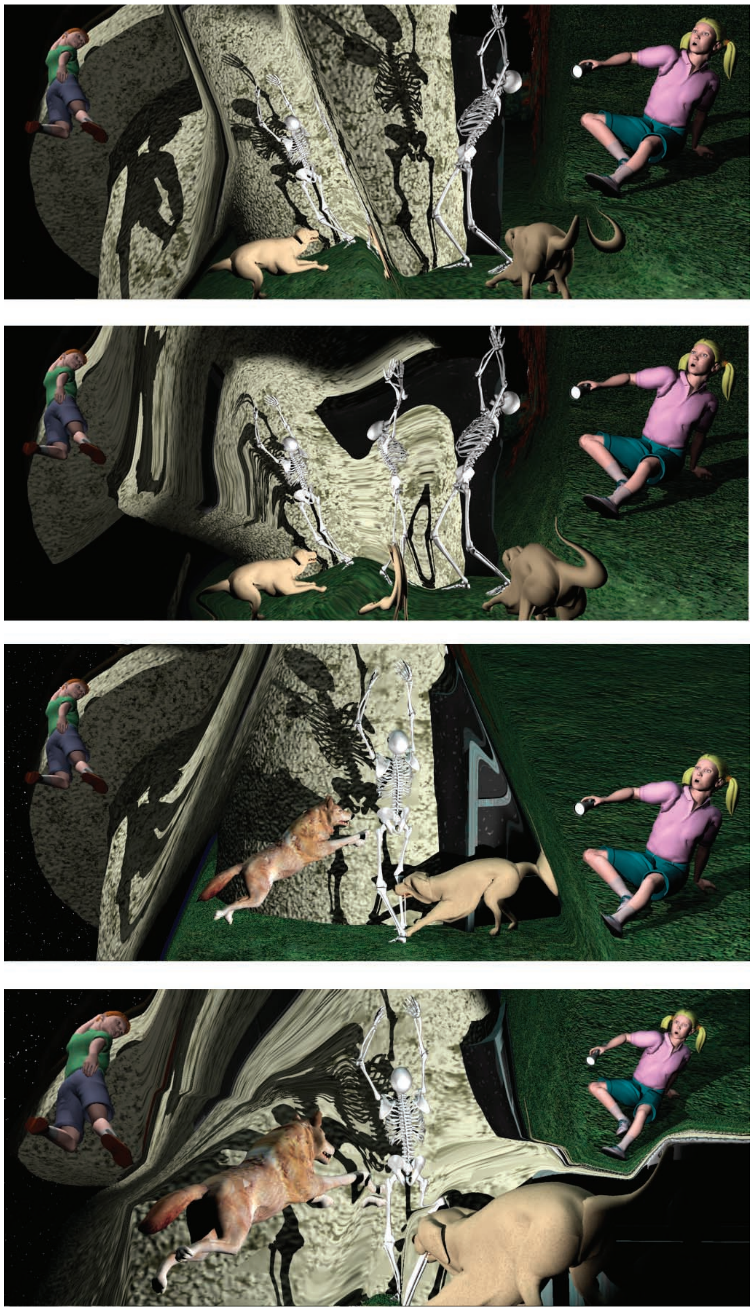
Let's look at some examples of the MCC system in action. Figure 12 shows two images from a chase in an alley. The man with the crutch is being run down by agents from both ends of the alley, while someone watches from high above the building he's trying to enter. Figure 13 shows four images of this scene, and Figure 14 shows a collage built from them. Figure 15 shows the rerendered output.

Figure 16 shows a scene from *A Bad Night at Big Light*. Poor Timmy is hanging onto the lighthouse by one hand while a skeleton reaches for him from below. Hobbled by a broken ankle, Sally watches from the side as her two dogs attack the skeleton. Figure 17 shows several different MCC collages for this scene.

Finally, Figure 18 shows a scientist working late in her lab one night. Figure 19 shows two different MCC collages for this scene.


15 Rerendered MCC version of the collage in Figure 14.


16 Scene from *A Bad Night at Big Light.*


17 Four collages of the scene in Figure 15.

Animation

This algorithm is frame based, which makes it easy to use for animation. Think of the creation of collage images like the creation of key points: You create one where something interesting is happening, and let interpolation handle everything in between.

Suppose that you want to create a 500-frame animated sequence using eight cameras. Let's say you create collages at frames 0, 150, 400, and 500. The system processes those four collages and creates four lens reader files named, for example, rerender-frame000.txt, rerender-frame150.txt, and so on.

Then the in-between program uses the ray information at those frames to compute lens reader files for all the other frames, saving one text file per frame. Now you simply render the animation using the lens reader shader. When it's initialized at each frame's start, it reads in the data from the corresponding file.

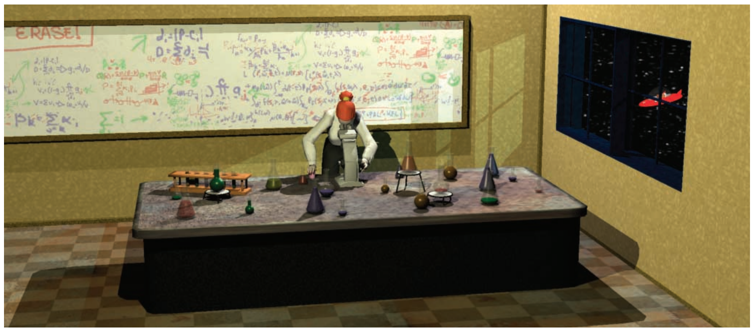
If you want to include motion blur, you can read in pairs of frame files for each rerendered image, and use each ray's time value to interpolate between the data in the two frames.

Figure 20 shows two keyframes for a 90-frame shot starring our couple from the cafe. These keys correspond to the first and last shot frames. Figure 21 shows several of the intermediate frames created to complete the shot.

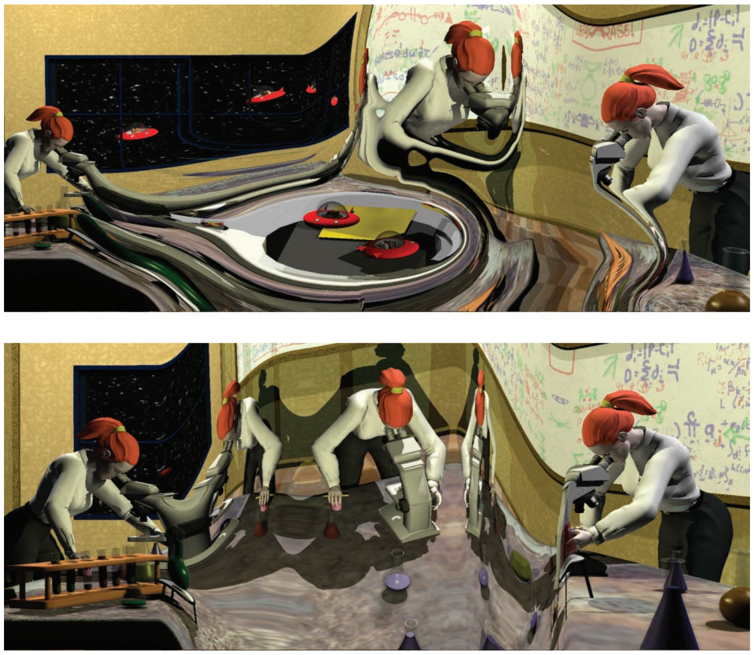
Figure 22 (see p. 94) shows the five keys used to create an 800-frame shot from *Rundown*. The agents are running toward Sharky (the man with the crutch) as he tries to get the door open. Figure 23 shows several of the intermediate frames created from these keys.

Discussion

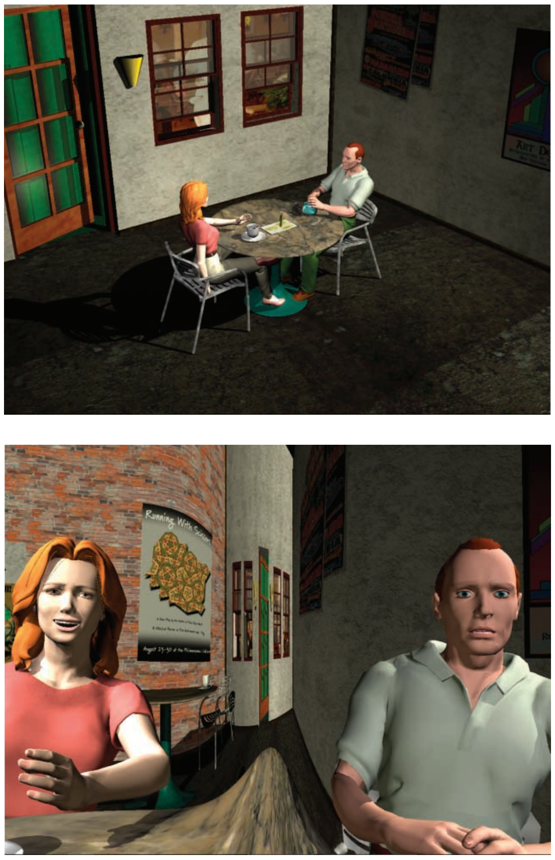
Note that the MCC technique doesn't slow down the rerendering step when we produce the final output frames. There's a short pause at the start of each frame to read in the database of rays, and then there's a bit of work done by the lens reader to compute each new eye ray. But this small amount of computation is insignificant compared to the overall task of rendering an image. In my nonoptimized, testbed implementation, I measured a quarter-second pause at the frame's start to read in the database; the processing performed by the lens



18 **Scientist working late into the night.**



19 **Two MCC versions of Figure 17.**

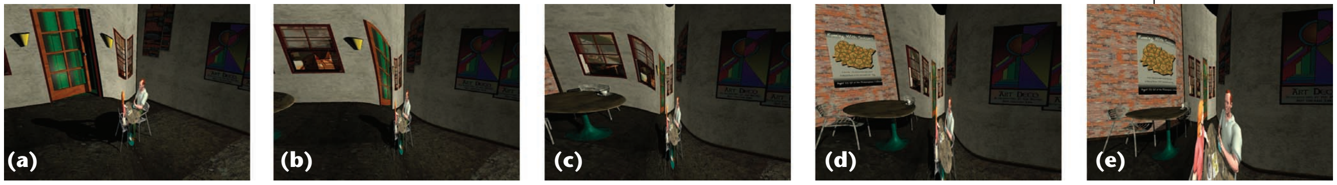


20 **Two key frames for the first and last frame of a 90-frame sequence.**

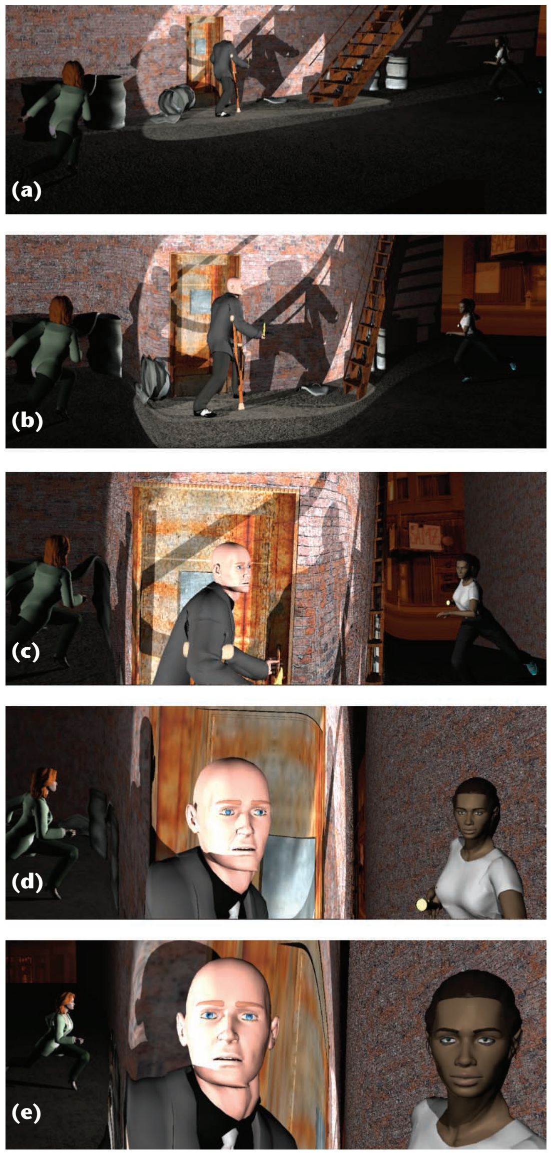
reader during the actual rendering process was too small to detect.

I typically build my collages at a smaller resolution than the final output. Because the lens reader interpolates the input rays, everything is still smooth even when the output resolution is larger than it is in the collages and regions. I usually work at one quarter of the final output size, which means my files are one-sixteenth the size of the final output images. This means the MCC files consume less disk space, and processing is faster. I only work at higher resolutions when there are fine details in the collage or region images that I want to preserve in the output.

The MCC technique raises some interesting artistic and technical ideas. Suppose we have a collage where the top half is one rendering, and a small blob appears somewhere on the bottom edge, as in Figure 24a. What should happen in the lower corners? The general problem is that outside the convex hull of the renderings, we don't have data to interpolate. One reasonable solution might be to use some kind of parametric extrapolation. An alternative would use an iterative approach that "grows" extrapolated data beyond the interpolated convex hull using incremental techniques. My general solution to this is to render images slightly larger than I need,



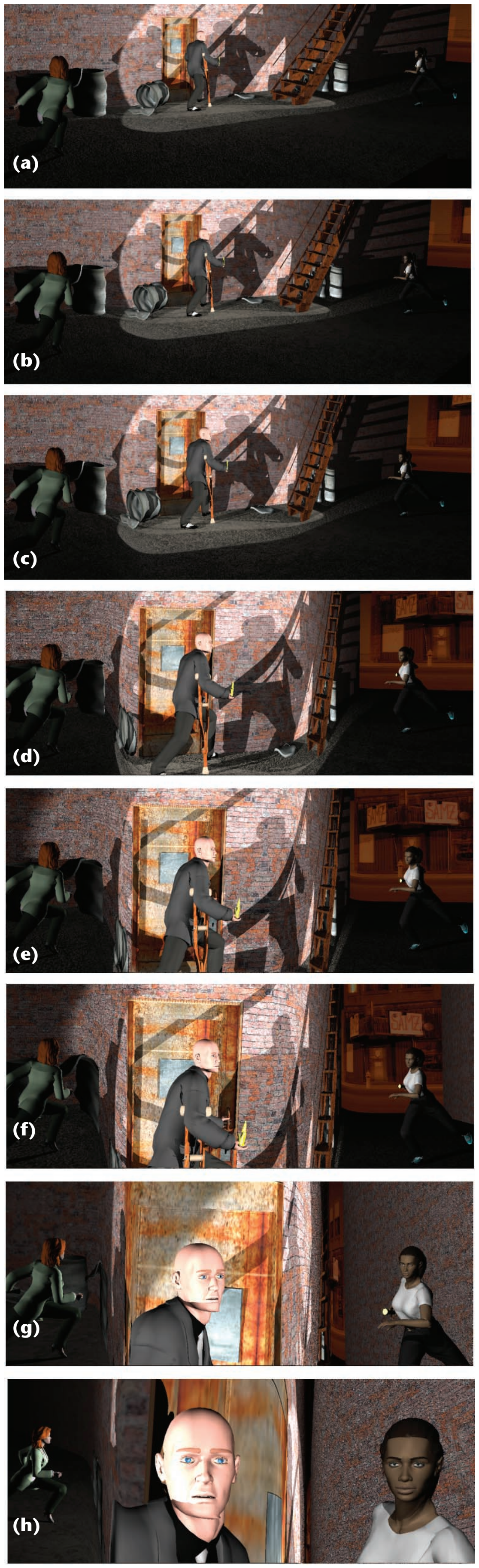
21 **Five intermediate frames from the sequence created by the two keys in Figure 20. These are frames (a) 15, (b) 30, (c) 45, (d) 60, and (e) 75.**



22 Five key frames for an 800-frame shot from *Rundown*. These are frames (a) 0, (b) 200, (c) 400, (d) 600, and (e) 800.

so that I can always include a frame of in-camera pixels along the outermost border of my collages, as in Figure 24b. After my new frames are computed, I crop them and discard the border.

I've thought about building a painting interface on top of the rendering system. Imagine starting with a blank image, selecting a camera, and then painting over the image. In this approach, the camera is rendered and immediately displayed where paint is placed. The director can work with these image regions like camera viewports in the rendering system: They can be moved around, cut, pasted, and changed in shape. The director can select controls like those in the rendering interface, and directly change the camera's location, direction, field of view, and so on. The system could produce low-reso-



23 Eight intermediate frames from the sequence created by the five keys in Figure 22. These are frames (a) 50, (b) 100, (c) 150, (d) 250, (e) 300, (f) 350, (g) 500, and (h) 700.

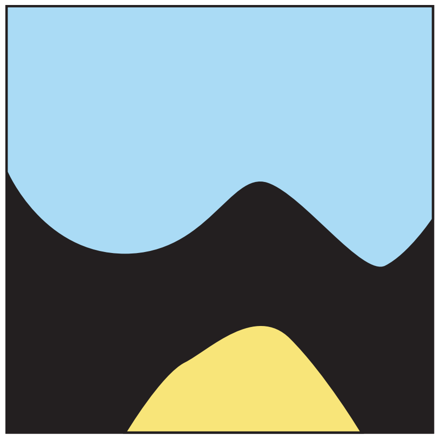
lution interpolation results in as close to real time as possible; when the director ceases to paint for a moment, the system takes the idle time to compute an ever-denser smattering of interpolated pixels, giving an ever-better impression of the final result. The director could then respond immediately and adjust the cameras and renderings as desired. This might be more convenient than building a collage from prerendered images.

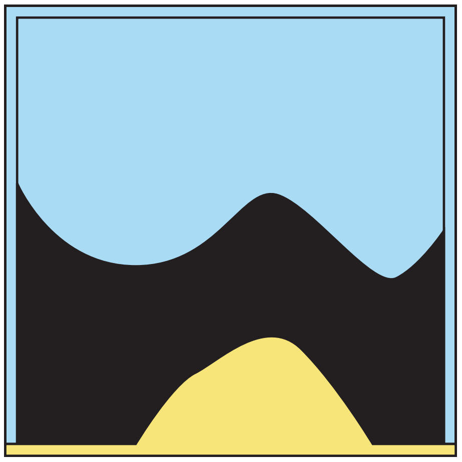
The MCC approach computes the pixels missing in the collage by ray tracing a synthetic environment. An alternative approach would use image-based rendering (IBR) to retrieve environment data from scene images. The director could start the process with rendered images built from the IBR data, or with real photographs taken on a scene, where another program takes the camera calibration information and writes the ray data that corresponds to the camera's image. If during the interpolation phase the MCC algorithm needs pixels that aren't available from the IBR data set, it could tell the user where to place and direct a camera to gather the missing imagery. I think we could integrate both IBR data and synthetic 3D renderings in the same scene, either mixing them seamlessly for a single continuous image, or deliberately making it clear which is which for artistic reasons. ■

Acknowledgments

Thanks to Steven Drucker, Ned Greene, Tom McClure, and Claude Robillard for discussions, ideas, and help.

Readers may contact Andrew Glassner at andrew@glassner.com.


(a)


(b)

24 (a) Blob problem. How do we find weights for the pixels in the lower right corner? (b) My solution is to always include a frame of camera pixels in the collage.