Pixar Animation Studios - Technical Memo #217

Texture On Demand

Darwyn Peachey

Pixar

San Rafael, California

Abstract

Texture On Demand (TOD) is a technique for organizing large amounts of stored texture data in disk files and accessing it efficiently. Simply reading entire texture images into memory is not a good solution for real memory systems or for virtual memory systems. Texture data should be read from disk files only *on demand*. In the TOD technique, each texture image is stored as a sequence of fixed-size rectangular regions called *tiles*, rather than in the conventional raster scan-line order. Tiles are an appropriate unit of texture data to read into memory on demand. As fixed-size units with good locality of reference in a variety of rendering schemes, tiles can be cached in main memory using the paging algorithms common in virtual memory systems. Good results have been obtained using an LRU tile replacement algorithm to select a tile to be deleted when main memory space is required.

Prefiltered textures are an important means of limiting bandwidth. TOD uses a set of prefiltered texture images called an *r-set*, a generalization of the texture pyramid (‘‘mip map’’). Texture filtering methods are reconsidered in terms of their performance in the TOD environment. Efficient filtering methods using the r-set are described.

The paper describes various implementations of TOD, including a virtual memory implementation and a distributed implementation on a 16-processor multicomputer.

CR Categories and Subject Headings: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism − color, shading, shadowing, and texture.

General Terms: Algorithms, Performance.

Additional Keywords and Phrases: antialiasing, filtering, mip map, multicomputer, paging, pyramid, texture, virtual memory.

1. Introduction

A computer graphics image is more realistic and pleasing to the eye when the objects in the image appear to be complex and detailed. Texturing is a way of adding apparent complexity without changing the underlying geometric model. The additional details are created by modulating shading parameters as a function of some texture coordinate system. It is common to use a digitized or precomputed 2D table of texture values to generate the modulating function. This texture table is accessed during rendering and the values in the table are filtered to obtain the texture values used to modulate shading parameters. Since the source of the data in a texture table is often a digitized photograph, the table may be thought of as a texture image.

Access to tabular texture data has been studied by several authors [1,2,5-11,18]. A primary concern of these authors has been to reduce the CPU time cost of texture filtering calculations. It is commonly assumed that texture data resides in main memory and can be accessed directly by an array reference.

We are interested in making pictures which use tens, hundreds, or thousands of high resolution texture images. A single texture image typically consists of one to sixteen megabytes of data. There are difficult problems in the design of a texture system that can efficiently access many such images during rendering. Textures of this size inevitably reside in disk files before rendering, and an accurate measure of the cost of accessing the textures must take into account the operating system overhead and disk I/O time involved in reading the texture data into memory.

This paper describes *texture on demand* (TOD), a technique for organizing and accessing texture data which attempts to minimize the I/O and CPU cost of access. Although the paper focuses on the case of 2D texture tables (images), the technique is easily extended to 1D tables and 3D tables (solid textures [13,14]).

2. Principles of Efficient Texture Access

Assuming that texture data initially resides in disk files, the total time cost of texture access during the rendering of an image is

$$\text{cost} = N\,L + \frac{D}{rate} + CPU$$

where N is the number of I/O operations (texture reads), L is the latency or time overhead of each read (including setup, seek time, rotational delay), D is the total amount of texture data read in bytes, *rate* is the I/O system transfer rate in bytes per unit time, and CPU is the CPU time taken to access and filter the texture data. We assume that L and *rate* are fixed by the hardware and operating system design. Thus to minimize the cost, a texture access algorithm should minimize N, D, and CPU.

2.1. Why Not Just Read It All?

There is a temptation, especially on computers with virtual memory systems, to ‘‘just read it all in.’’ Our cost function shows one reason why this is a poor strategy: reading texture data that might never be accessed increases D and adds to disk read time. It is generally difficult to predict which parts of a texture will be needed, since a textured surface can be partially or completely hidden by another surface.

The other reason not to read the entire texture into memory before rendering is to conserve scarce memory space. Unless the computer system has an infinite amount of real memory, there usually won’t be enough memory to hold all of the texture data which might be needed. A large virtual memory space simulated with a small real memory doesn’t solve the problem. Reading a texture into virtual memory can result in two I/O operations, one to read the data from a file and another to page it out to a disk paging area. A third I/O operation can take place when the texture is accessed (but only for the parts that are actually used in rendering the image).

2.2. Texture On Demand

In order to minimize D the access algorithm must avoid reading texture data that is never used. It is difficult to know in advance what portion of the texture data will be needed, since visibility of the surfaces must be considered. Visibility is properly the concern of the renderer. This suggests that the right thing to do is to read texture on demand, as it is required by the renderer.

The extreme implementation of this idea is to read the texture one pixel at a time. This minimizes D (assuming no pixel is ever read more than once), but makes N enormous, in fact, equal to D. Since L is four or five orders of magnitude larger than $1/rate$ in practice, the latency dominates the rendering time and the overall rendering time is greatly increased.

Obviously texture data must be read in units larger than single pixels. Choosing the best size for I/O operations is a complex decision involving the hardware and operating system characteristics; in most systems an I/O size of 1K to 16K bytes balances the effects of latency and transfer rate. Each texture read operation will include many ‘‘extra’’ pixels along with the desired texture pixel. It is important to ensure that most of the extra pixels are ones that will be needed later in the rendering process. Any unused texture data which is read will increase D and thus increase the total cost of texture access.

Most images are stored in raster scanline order, so a scanline can be read by a single I/O operation. But a scanline is not an ideal unit of I/O for texture access. For example, if only the left half of a texture is used, each scanline that is read will consist half of useful pixels and half of wasted pixels; in an extreme case, all but one pixel of each scanline could be wasted. Scanlines can also exhibit poor *locality of reference* [3], that is, a low probability that a pixel access will soon be followed by further accesses to the same scanline.

The optimum I/O unit for texture access is both renderer and image dependent, but we would prefer a unit which performs fairly well with a wide variety of renderers and scene descriptions. Such a unit should be *isotropic*, that is, it shouldn’t have a preferred access direction, since effective use of the preferred direction would require that the rendering algorithm and scene description be adapted to the texturing algorithm. A scanline is a poor unit because it is not isotropic; accesses along the scanline are far more efficient than accesses across scanlines.

2.3. Texture Tiles

A better unit for texture I/O is a square (or nearly square) region of an image called a *texture tile*[1]. A square is a convenient approximation to the most isotropic shape, a circle. Each texture image can be cut up into a grid of rectangular tiles (Figure 1). The dimensions of a tile are such that it

[1]Dungan, et. al. [6] use the term ‘‘texture tile’’ differently. Their concern is to build up large areas of texture by repeating a single texture tile designed to have opposite edges which match. Our textures consist of many different texture tiles. The edges of a given tile do not (necessarily) match.

can fit into a fixed I/O size, and therefore depend on the size of one pixel of the texture. The size and dimensions of a tile are restricted to powers of two to simplify and speed up access calculations. If tiles are 4K bytes in size and texture pixels consist of a single byte each, the tile dimensions are 64×64 pixels. If instead each pixel consists of two bytes, the tile dimensions are 64×32. Tiles are square if the number of pixels in a tile is an integral power of four; otherwise, tiles are twice as wide as they are high.

Texture images are stored in disk files as a set of tiles, rather than as a sequence of image scanlines. A tile can be read with a single I/O operation. Each tile is itself a small image stored in raster scanline order. The tiles may be stored in any order in the texture file, as long as there is some means to determine the file address of a tile given its location in the image. Usually the tiles of a particular image are stored one after another and are addressed as a 2D array of tiles in row-major order. Alternatively, tiles can be located through a 2D table of tile addresses. In this scheme, only one of a set of identical tiles needs to be stored and read; others may be represented by multiple references to the same tile address.

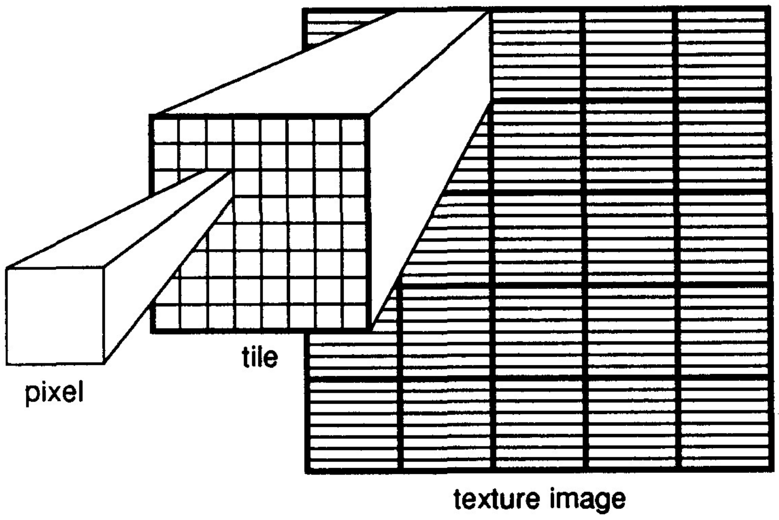

Figure 1: Tiled texture organization.

2.4. Prefiltering Texture Images

Reading texture tiles on demand helps reduce *D*, the total amount of texture data read, by not reading unused parts of the texture images. *D* can be reduced further by using texture images of an appropriate resolution. Much more data must be read from a high resolution texture image than from a low resolution one to filter over a given region in texture space. However, it is unreasonable to expect the texture user to select the appropriate resolutions before rendering.

Dungan, *et. al.* [6] and Williams [18] have proposed schemes in which a high-resolution source image is prefiltered to a variety of lower resolutions to form a *texture pyramid* as illustrated in Figure 2.

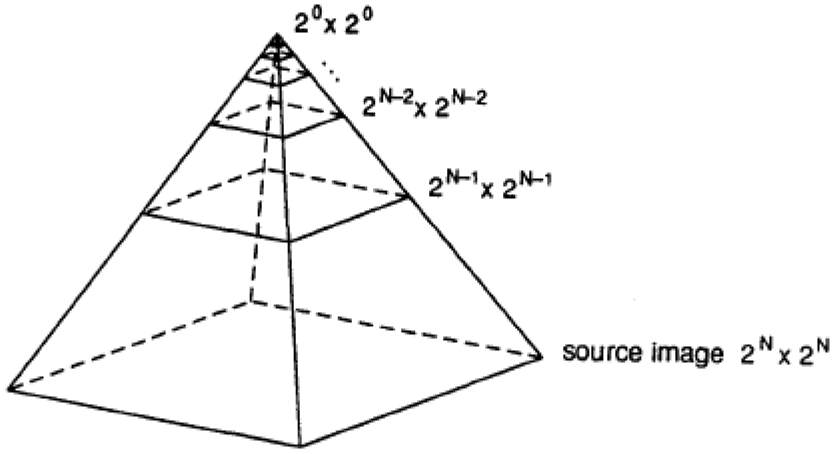

Figure 2: A texture pyramid.

Williams explains how approximate filtering calculations can be performed using a fixed number of texture pixels from the pyramid and a fixed number of CPU operations regardless of the size of the area being filtered. Prefiltering of texture offers a substantial reduction of the texture I/O volume D as well as a reduction of the CPU cost of access and filtering. In fact, we can make the following observation regarding the minimum amount of texture data required to render an image:

Principle of Texture Thrift

Given a scene consisting of textured 3D surfaces, the amount of texture information minimally required to render an image of the scene is proportional to the resolution of the image and is independent of the number of surfaces and the size of the textures.

This principle is a consequence of the signal processing limitation on the amount of information which can be conveyed by the rendered image: each pixel of the image can represent only one sample of each texture on a given surface. Imagine a scene consisting of many small textured objects. A wide angle view of the scene contains many objects, but little detail is visible in the texture on each object. A close up view of one of the objects shows the texture of that object at a much higher resolution, but does not show the other objects. The total number of texture pixels seen in the two images is about the same. Of course, this depends on being able to filter a texture to any desired resolution by accessing a fixed amount of stored texture data − exactly the result of using prefiltered textures.

If each surface in the scene uses more than one texture (*e.g.*, a color texture and a bump texture), the amount of texture necessary to render the scene is proportional to the number of textures per surface, which we call the *texture complexity* of the scene.

2.5. Resolution Sets

To simplify the following discussion, we use a powers-of-two notation for image resolutions. An image is a 2D rectangular array of pixels indexed by two coordinates called s and t. When the s-resolution of the image is 2^S and the t-resolution of the image is 2^T, the resolution is specified in powers-of-two notation as $[S, T]$. In this notation, a texture pyramid consists of a square source image of resolution $[R, R]$ and images of lower resolutions $[i, i]$, $i = 0, 1, \cdots, R-1$ obtained by repeatedly *minifying* (filtering an image to produce a lower resolution image).

The texture pyramid is a special case of a more general set of minified texture images which we call a *resolution set* or *r-set*.. An r-set is built from a source image of resolution $[S_0, T_0]$. The images in the r-set have resolutions $[S, T]$ where S is an integer satisfying $S_0 \geq S \geq 0$ and $T_0 \geq T \geq 0$. A *complete r-set* is the set of *all* such minified versions of the source image. Any other r-set of a given source image is a proper subset of the complete r-set for that image.

For any r-set, we can construct a Boolean matrix B called the *pattern matrix* which tells us exactly which lower resolution versions of the source image are contained in the r-set (see Figure 3). Entry $B_{i,j}$ is true (indicated in the figure by •) if and only if the r-set contains an image of resolution $[i, j]$. Some common r-set resolution patterns are illustrated by the small matrices at the bottom of the figure. If the source image is square, the diagonal pattern is exactly the same as a resolution pyramid. In an r-set the diagonal pattern can be applied to non-square images as well

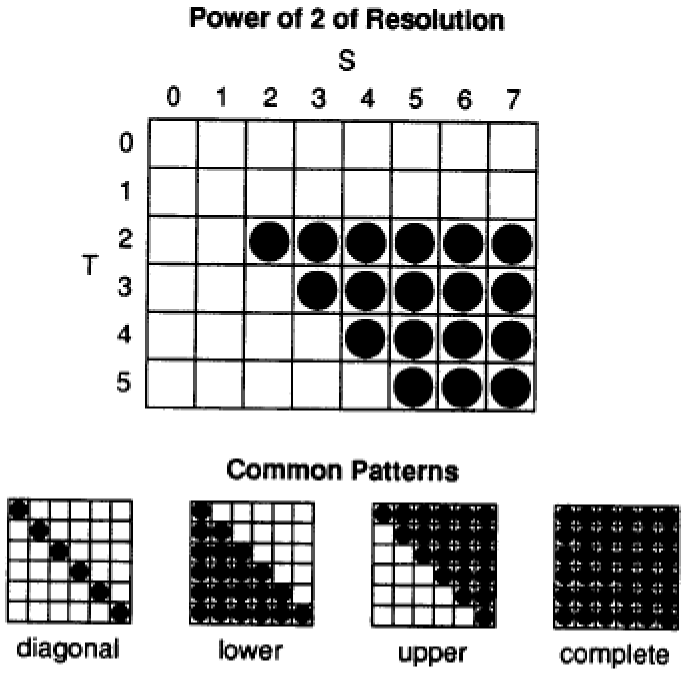

Figure 3: Pattern matrix of an r-set.

(using the diagonal which passes through the source image). The lower triangular pattern provides all values of the *s*-resolution for any image on the diagonal, and the upper triangular pattern provides all values of the *t*-resolution for any image on the diagonal. These patterns are useful when it is known in advance that the textured surface will be foreshortened by perspective or otherwise compressed so that low s or t resolutions will be needed in combination with higher resolutions in the other direction. A classic example is the case of wood texture on a floor which is seen from a low angle (as in Figure 7). The complete r-set provides all lower resolution versions of the source image and is useful when the texture may be foreshortened in either the *s* or *t* direction. Such r-sets provide a remedy for the inconvenience of filtering rectangular areas using a texture pyramid created with a square filter noted in [11].

2.6. Storage Cost of Resolution Sets

The storage cost of an r-set is the amount of disk space required to store the images whose resolutions are included in the pattern matrix. This section gives upper bounds on the storage cost for the most common resolution patterns in terms of the size of the source image. The bounds are most closely approached when the source image is square and of a very high resolution.

As shown by Williams [18], the diagonal pattern r-set (pyramid) takes no more than 4/3 as much storage as the source image, since each resolution is one-quarter as large as the next larger image and $1 + 1/4 + 1/16 + \cdots = 4/3$.

The complete r-set takes no more than four times as much space as the source image. Since $1 + 1/2 + 1/4 + \cdots = 2$, the images corresponding to each row of the pattern matrix take at most twice as much storage as the rightmost element of the row. The bottom row takes no more than twice as much storage as the source image. Each row takes half as much storage as the row below it, so the series summation can be applied again to determine that the complete r-set takes no more than twice as much storage as the bottom row, or four times as much as the source image.

Figure 4 shows how a complete r-set generated from a square source image can be arranged in a square whose area is less than four times as large as the source image. In the example the source image is 8×8 and the entire r-set fits inside a 15×15 square. This arrangement[2] is analogous to Williams' mip-map organization for a texture pyramid, and is a convenient organization for storing the images of a complete r-set in a framebuffer.

The upper and lower triangular r-sets take no more than $8/3 \approx 2.67$ times as much storage as the source image. This result can be obtained by a series expansion, or by the following argument: for a square source image, the upper and lower triangular r-sets have the same storage cost. Together they cost as much as the complete r-set plus the diagonal r-set, since the diagonal is included in both triangular r-sets. Therefore, the cost of each triangular r-set is $(4 + 4/3)/2 = 8/3$.

2.7. Tile Access and Cache Management

When a particular texture pixel is required, the TOD algorithm must determine whether the tile containing the pixel is already in memory. If so, the memory address of the tile must be determined. If the tile is not in memory, a *tile fault* is said to have occurred, and the tile must be read into some area of memory before the texture pixel can be used. The texture file identification, r-set member identification (resolution), and the tile location in the grid of tiles (tile number) together form a *tile key* which uniquely identifies a single tile among all the tiles which possibly could be accessed. The tile key is looked up in a hash table of in-memory tiles in order to find its

[2]This texture storage scheme was used by Charlie Gunn in Pixar's ChapReyes renderer, which stores texture data in framebuffer memory. Similar extensions to the mip-map structure have been implemented at NYIT.

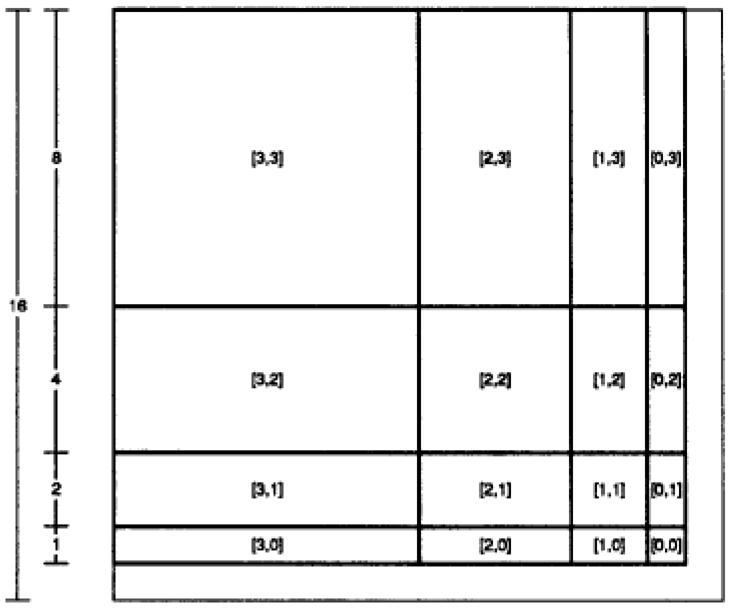

Figure 4: Framebuffer layout of complete r-set.

address or to determine that it is not in memory. The implementation of this procedure in various types of computer systems is discussed in section 3.

If memory is scarce, it is not possible to retain all of the tiles which are ever read. Even in a system with a huge (but finite) real memory, there is always some competing demand for memory, and it makes little sense to retain texture data which is unlikely to be reused. A virtual memory system cannot be treated as equivalent to a huge real memory, because careless use of virtual memory leads to enormous implicit use of disk I/O which can destroy the gains made by TOD.

Consequently, it is often necessary to make room for an incoming tile by removing another tile from the set of tiles in memory. Because tiles are never modified during texture access, removing a tile is simply a matter of deleting all references to it and reusing its memory space. The important question is how best to choose a tile to be removed. This is basically a page replacement problem and has been studied at length by operating system researchers interested in virtual memory algorithms [3]. TOD uses the LRU (least-recently used) page replacement algorithm to determine which tile to remove and replace. The in-memory tile which has least recently been accessed is removed and replaced with the tile being read. In multi-threaded renderers some tiles are not eligible to be removed because they are in use by other threads; such tiles are exempt from the LRU test.

High locality of reference is known to be the key to good performance in LRU-managed caches. The next section discusses locality of texture pixel accesses in various texture filtering techniques.

2.8. Texture Filtering

The demand for texture data comes from the texture filtering algorithm used in the renderer, and that algorithm has much to do with the amount of texture accessed and the locality of those accesses. Texture filtering is already a much-studied topic. Most of the work has focused on accurate filtering by direct convolution [1, 7, 10] and on approximate filtering methods that are less general and far less costly in CPU time [5, 8, 11, 15, 18]. Our concern for the *total* cost of texture access (I/O and CPU) leads us to reevaluate available methods of texture filtering in terms of their I/O demands and access locality.

Each filtering method can be characterized in terms of the number of texture pixel accesses it makes to filter a given area in texture space, and the locality of those accesses. A large number of texture pixel accesses is costly in both I/O and CPU time, because the texture pixels must be read from the texture file and then combined to obtain the filter result. Poor access locality leads to a sharp increase in the amount of I/O required to keep the necessary set of tiles in a limited amount of main memory (thrashing).

Direct convolution on the source image is the simplest and most accurate filtering technique. The number of texture pixel accesses is directly proportional to the filtered area, so the I/O and CPU costs are high for large filtered areas. However, the access locality is quite good, since the accesses are to adjacent pixels of a single image.

The summed area table [5] and other integrated array techniques [15, 11] make a small constant number of accesses to the texture table regardless of the size of the filtered area. A properly aligned rectangular area of the texture can be filtered with four accesses to the table. The I/O and CPU costs are increased somewhat due to the need for much higher numerical precision in the table entries than in the source image. Access locality is poor: the four accesses are at the corners of the rectangular area, and so may be very widely separated when the filtered area is large. Filtering over arbitrary quadrilaterals or other areas is difficult using these methods, so the filtering is performed on a bounding box of the true filtered area.

Williams' trilinear interpolation technique [18] also makes a small constant number of accesses to the texture table regardless of the size of the filtered area. Four texture pixels are accessed from each of two adjacent resolution levels of a texture pyramid and bilinearly interpolated to obtain a texture value from each level. The filter value is obtained by linear interpolation between the texture values of the two levels. This method exhibits good locality within each resolution level, since the pixels accessed are adjacent, but locality is reduced by accessing data from two different resolution levels. The minimum cache size required to ensure good cache hit rates is approximately doubled by the need to access two resolution levels rather than one. As with the integrated array techniques, filtering over arbitrary quadrilaterals is inconvenient, so a bounding box of the filtered area is used. Non-square areas are difficult to filter accurately because the prefiltering of the pyramid is based on a square filter.

As a compromise among CPU efficiency, I/O volume, locality of access, and filter shape control, TOD uses a direct filter convolution on an image of appropriate resolution selected from an r-set. Direct convolution has good locality of access. A region of any shape can be filtered using the convolution, but the accuracy with which the shape is approximated varies depending on the resolution of the selected image. An image must be selected from the r-set so that the width of the filtered area is at least one pixel in each direction or aliasing artifacts may be produced. If the r-set is complete, a suitable image can be found with no more than four pixels in the filtered area. A higher resolution can be selected to give better control of the filter shape and response. If the r-set is not complete, the resolution of the selected image may be unduly high, and the number of pixels accessed by the convolution can be large.

Given a complete r-set, the TOD filtering method accesses from one to nine adjacent texture pixels of a single image. A region of arbitrary shape may be filtered, and the filtered results are generally less blurred than those produced by Williams' trilinear interpolation. [Note to reviewers: an image to illustrate this will be included in the final paper.] The direct convolution approach is somewhat more costly in CPU time than trilinear interpolation, but results in better I/O performance; This seems to be a sensible compromise in view of the widening gap between CPU speed and I/O speed.

3. Implementation of TOD

This section discusses the structure of an implementation of the TOD technique and the issues raised by various computer system architectures. Section 4 discusses the results we have observed from extensive use of the TOD implementation on a variety of computers.

3.1. Overall Structure

The diagram in Figure 5 shows the hierarchical structure of our implementation of TOD.

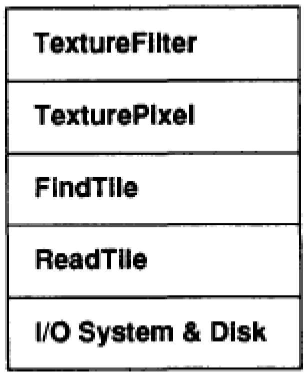

Figure 5: Block Structure of Access Mechanism

The top layer of the structure is `TextureFilter`, the texture filtering routine. Texture pixels are obtained by calling `TexturePixel`, the second level of the structure. `TexturePixel` calls the tile access routine `FindTile` as needed to locate the tile containing the desired pixel. `FindTile` calls the tile I/O routine `ReadTile` if the required tile is not in memory (a tile fault). The structure of the texture file is hidden in `ReadTile`, since only `ReadTile` needs to be able to locate a texture tile in a disk file.

3.2. The `TextureFilter` Routine

A texturing operation begins with the need to filter a texture over some region in texture space. The texture coordinate system ranges from 0 to 1 in s and t over the source image. The renderer requests the filtered value of the texture over a region with a specified center (s, t) and widths $(swidth, twidth)$ in the texture coordinate system[3].

[3]This information is sufficient to specify a rectangular or elliptical region in texture space. This is not always an exact representation of the area being textured. However, the focus of this work is on efficiency rather than precision, and the filtering scheme is admittedly an approximation.

The first task of the `TextureFilter` routine is to determine the *desired* resolution in s and t. This is simply $1/swidth$ and $1/twidth$ respectively. The widths are clamped at a minimum of one source image pixel; this ensures that extremely magnified views of a texture show blurred pixels instead of sharp-edged rectangular pixels. The next step is to find a member of the r-set which has an equal or higher resolution in each direction. Once a member is selected, `TextureFilter` calls `TexturePixel` repeatedly to obtain the pixels of that texture image which are contained in the filtered region. A direct filter convolution is performed to obtain the filtered value of the texture and this value is returned.

Finding the most suitable member of an r-set to filter a given region of a texture involves a search operation on the r-set pattern matrix. The search begins with the resolution $[S, T]$ where $S = \lceil \log_2(1/swidth) \rceil$ and $T = \lceil \log_2(1/twidth) \rceil$. The search examines the pattern matrix along diagonal lines of decreasing S and increasing T, that is, from upper right to lower left in the matrix (Figure 6). Successive diagonal lines are one step further to the lower right. The final resolution considered is that of the source image. The search stops when it finds a • in the matrix indicating that the corresponding texture image is a member of the r-set. The rationale for this search method is:

- it sweeps out the entire area in which a suitable resolution might be found.
- each diagonal search line has resolutions of the same total size, with each successive search line doubling this size.

A suitable member is guaranteed to be found and is guaranteed to have a minimal storage size.

3.3. The `TexturePixel` Routine

The existence of texture tiles is hidden from the texture filtering code. The filtering code simply requests texture pixels in terms of their integer coordinates $(\bar{s}, \bar{t})$ which range from 0 to $resolution - 1$ in each direction based on the resolution of the image selected from the r-set. `TexturePixel` determines the tile dimensions in pixels, and divides the coordinates by the tile dimensions to obtain the tile number. It then constructs a tile key and calls `FindTile` to obtain the memory address of the tile. `TexturePixel` indexes the tile as a small image array using the coordinates ($\bar{s}$ mod *s-resolution*, $\bar{t}$ mod *t-resolution*). The size of the pixels in the texture is taken into account in the indexing, and the pixel value, which may be an 8-bit, 16-bit, or 32-bit integer or a 32-bit IEEE floating point number, is converted to a common floating point

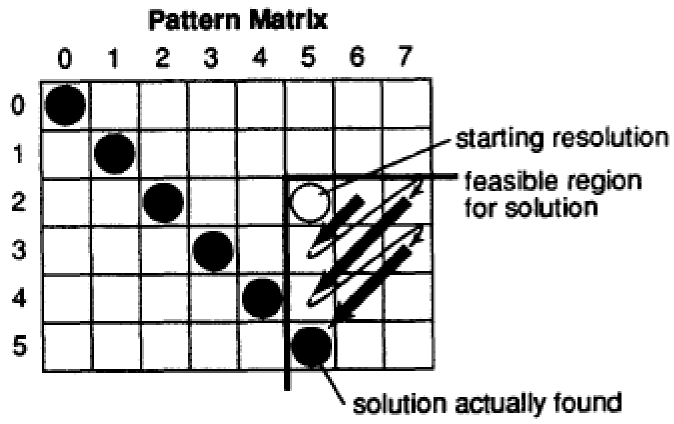

Figure 6: Resolution search in pattern matrix.

representation and returned to `TextureFilter`.

It would be very expensive to call `FindTile` for each texture pixel, so `TexturePixel` maintains a pointer to the last tile referenced in each texture. When a new pixel is required, a quick check is made to see whether the pixel is from the same tile as the previous pixel referenced in that texture. If so, the tile address is already known; otherwise, `FindTile` is called. In our experience, between 90% and 99% of texture accesses are to the same tile as the previous pixel.

3.4. The `FindTile` Routine

`FindTile` must implement an efficient mapping from a tile key to a memory address. This is done by hashing the tile key to obtain a table index. Collisions are rare in small caches (a few hundred tiles) if the hashing function uses the tile number and file ID from the tile key. If a tile is not found in the hash table, it is not currently resident in memory; a tile fault has occurred. Memory space must be found for the desired tile, possibly by removing a tile, and then `ReadTile` can be called to actually read the tile into memory. Each tile in the cache has a timestamp from its last access. The least-recently accessed tile is selected for removal when space is required. Note that the LRU tile removal procedure can be used to satisfy other demands for memory space which arise outside the texture mechanism.

3.5. Virtual-Memory Uniprocessor Implementation

The preceding sections have described the implementation of TOD for a real-memory uniprocessor. Adding virtual memory to the uniprocessor system adds the complication of interactions between the operating system's paging system and the tile cache mechanism.

If the tile cache size is made quite small, the texture tiles in the cache will be referenced frequently enough to remain in real memory. Some virtual memory systems make it possible to ask the operating system to force a piece of the address space to reside in real memory, and this can be applied to the tile cache. In such cases, TOD functions much as it would in a real-memory environment.

An alternative strategy is to make the tile cache very large, so that there is never any need to remove a tile from the cache. Instead the cache grows as necessary to hold every texture tile that is ever accessed. The tile access mechanism provides the mapping from the tile key to the virtual memory address where the tile resides, and the operating system pages the tile cache in and out of memory as necessary. There are some disadvantages: the hash table becomes so large that the key to address mapping can be expensive. At least one disk write operation is needed to copy each paged-out tile into the operating system's page area.

Some virtual memory systems (*e.g.*, the Sun-4 with SunOS 4.0.3) provide a way to map a disk file into the address space of a process. A texture file can be made directly accessible as part of the renderer's virtual address space. Tiles are read from the file automatically when they are referenced, and there is no need to copy tiles into a paging area. Tile keys still need to be mapped to virtual memory addresses, but this can be particularly simple: if the tiles of a single texture image are stored in sequence in the file, the hash table mapping can be limited to one entry per r-set member. The file ID and member ID map to the starting address of the member image in virtual memory. A particular tile address is computed by adding an offset that depends on the tile number. Unfortunately, this method can consume virtual address space very rapidly.

3.6. Real-memory Multicomputer

TOD has been implemented on an experimental rendering accelerator built at Pixar using 16 Inmos T800 Transputers with four megabytes of RAM per processor. The transputers are connected together by 20 megabit per second serial lines (transputer links). The transputer architecture does not provide virtual memory support, and there are no disks attached directly to the transputers. A host computer (Sun-4) is connected to one of the transputers by a transputer link. All texture data used by the rendering process which runs on each of the transputers must ultimately be read from the disk attached to the host computer and transmitted to the transputers via serial links.

To support remote texture access, the `ReadTile` routine in the transputer-based renderer is modified to transmit a message to a remote *texture server* process. The texture server may be on another transputer or on the host computer system which is attached to the multicomputer board. The texture server handles the tile request from the remote client just as it would handle a tile request generated by local rendering; that is, the requested tile is sought by `FindTile` in the server's texture cache. If the tile is not in the cache, a tile fault is generated. On the host computer this results in a disk file read operation. If the texture server is running on a transputer, the fault results in another network read request being sent to the texture server on another transputer or on the host.

We have experimented with various strategies for determining which texture server should be sent a given read request. Since the transputer communications network is usually configured as a ternary tree, one strategy is to send a request to the texture server on the processor which is the parent of the current processor. At the root of the tree, all requests are sent to the host computer. A tile request works its way up the tree, and is answered by the lowest ancestor in the tree which has the tile, or by the host computer if none of the ancestors has the tile.

A better strategy is the ‘‘location server’’ algorithm. All tile requests are sent to a location server running on one of the transputers (usually the root node). The location server keeps a list of the most recent requests it has received, and if it receives a second request for the same tile, it forwards the request to the texture server on the transputer which previously requested the tile. This texture server is likely to still have the desired tile in its tile cache. If not, it sends the tile request on to the host computer. Of course, the location server sends a request directly to the host computer if the desired tile has not been requested recently by another processor.

4. Results and Experience

A version of TOD is implemented in Pixar's *PhotoRealistic RenderMan 3.0* rendering package, which is based on the Reyes architecture [4]. Numerous still pictures and animated films have been produced using this renderer on a variety of machine types: real-memory uniprocessors such as the Compaq 386 MSDOS computer, virtual-memory uniprocessors such as the Sun-4 and SGI

4D/20, and the experimental 16-transputer multicomputers described in section 3.6. This section describes some of the texture-intensive uses of the renderer and presents quantitative results concerning the performance of TOD.

Figure 8 shows the image *Textbook Strike*, produced by Thomas Porter of Pixar. Each bowling pin is shaded using four textures to create the graphic designs on the pin and the dents and gouges in the pin's surface. Separate textures are used to simulate wood on the floor, and to produce the realistic reflections and shadows seen on the floor. The image was computed at a resolution of 3072×2304 and the texture images had correspondingly high resolutions.

Figure 9 is a 1024×768 pixel image of a living room in which the paintings, shadows, and reflections are produced by texture mapping. The shadow algorithm was developed by Reeves, *et. al.* [17]. The reflections in the chrome teapot are simulated using the cube-face environment texture [9, 16] shown in Figure 10. The image uses a total of thirteen textures. Reading in the entire set of textures needed to render Figure 9 would require 8.4 megabytes of disk I/O. Since this image was rendered on a 16-processor multicomputer, the data would then be copied 16 times using a total of 134 megabytes of transputer link I/O. Using the TOD method, only the necessary parts of the textures are accessed; this reduces the volume of transputer link I/O to 32.8 megabytes and the amount of disk I/O to 6.3 megabytes. The reduction in transputer I/O to 25 percent of the naive strategy is particularly large, because each transputer used only a portion of the texture data and only that part of the data contributed to the transputer I/O volume. The complete image made use of most of the texture data, so the total volume of disk I/O was reduced less dramatically from 8.4 to 6.3 megabytes.

In rendering Figure 9 from 94 to 99 percent of texture accesses were made to the same tile as the previous access of the same texture (allowing the use of the optimization described at the end of section 3.3). This statistic supports the claim that a nearly square tile is a good unit of texture data to maximize access locality. The small number of tile faults further indicates the high degree of texture access locality. With a cache size of only 64 tiles (4K bytes each) on each processor, the average tile fault rates in rendering Figure 9 range from 0.004 to 0.07 percent; 99.93 to 99.996 percent of texture accesses were made to tiles which were already in the cache. This high degree of locality in the access pattern of the texture filtering process makes it possible to render such images on scarce-memory systems such as the transputers for which the total size of the textures far exceeds the size of main memory.

Our experience has been that the distributed cache mechanism in the multicomputer implementation has a modest but substantial rate of return when the rendering workload is divided up among processors by having different processors render different parts of the screen. Typically 15 to 30% of tile requests sent to the network are satisfied by other transputers rather than by the host. This relatively low ‘‘hit rate’’ is not surprising, given that the rest of the texture system is functioning properly. Since only the required parts of the texture data are requested by each processor, there is little overlap in the texture data needed by different processors which are working on different parts of the image.

The short animated films *Tin Toy*, *Luxo Jr. 3D*, and *knickknack* were produced by Pixar’s animation group using several of the 16-transputer multicomputers described in section 3.6. Textures were used extensively to create painted patterns, text, photos, textured materials, reflections, and shadows.

The renderer has also been used by several Pixar customers, notably by Industrial Light and Magic to produce the ‘‘water pseudopod’’ special effect for the theatrical motion picture *The Abyss*. Mark Dippe gave the pseudopod realistic optical properties by using several textures to represent the environment and tracing rays through the water to the textures.

5. Conclusions

Cost measures for texture access must include terms for disk file access cost. These terms may dominate the cost when large amounts of texture are used. The texture-on-demand technique provides efficient access to texture by reading only the required parts of a texture image at only the appropriate resolutions. Prefiltered r-sets provide texture images of appropriate resolutions. A tile-based texture image organization facilitates I/O and gives good texture access locality (confirmed by high tile hit rates). An LRU tile replacement algorithm is effective in managing the set of tiles kept in memory in a scarce-memory environment.

The TOD technique can be implemented for real and virtual memory uniprocessors and can also be effective in distributed systems based on loosely-coupled multicomputers where the secondary storage is a communications network with distributed texture tile servers. TOD has been used with good results on several computer systems spanning a range of performance levels and architectures.

Further work is needed on tile cache management algorithms. Demand fetching with LRU tile replacement works well without taking the rendering algorithm into account, but better heuristics might be developed for use with a specific rendering algorithm whose texture reference characteristics are known. In this case, it might be possible to predict near-future accesses with a high degree of success, so that it is possible to prefetch tiles which will soon be accessed by the renderer.

Acknowledgements

Many people at Pixar helped to make this work possible. The principle of texture thrift in section 2.4 was suggested by Pat Hanrahan. A starting point for this work was an earlier texture system for Reyes developed by Sam Leffler with help from Eben Ostby and Bill Reeves. Eben Ostby and Tom Porter provided the images in Figures 7 and 8. Several people read and commented on early drafts of the paper, and Tom Deering converted my crude sketches into polished diagrams. Ed Catmull and Mickey Mantle deserve credit for creating an environment in which research and experimentation are possible despite commercial pressures.

References

1. BLINN, J.F. AND NEWELL, M.E. Texture and reflection in computer generated images. *Comm. ACM 19,* 10 (Oct. 1976), 542-547.

2. CATMULL, E. *A subdivision algorithm for computer display of curved surfaces,* Ph.D. thesis, University of Utah, Dec. 1974.

3. COFFMAN, E.G. AND DENNING, P.J. *Operating Systems Theory,* Prentice-Hall, 1973, 241-312.

4. COOK R.L., CARPENTER, L., AND CATMULL, E. The Reyes image rendering architecture, *Computer Graphics 21,* 4 (July 1987), 95-102.

5. CROW, F.C. Summed-area tables for texture mapping. *Computer Graphics 18,* 3 (July 1984), 207-212.

6. DUNGAN, W., STENGER, A., AND SUTTY, G. Texture tile considerations for raster graphics. *Computer Graphics 12,* 3 (Aug. 1978), 130-134.

7. FEIBUSH, E.A., LEVOY, M., AND COOK, R.L. Synthetic texturing using digital filters. *Computer Graphics 14,* 3 (July 1980), 294-301.

8. GLASSNER, A. Adaptive precision in texture mapping. *Computer Graphics 20,* 4 (August 1986), 297-306.

9. GREENE, N. Environment mapping and other applications of world projections. *IEEE CG&A 6,* 11 (Nov. 1986), 21-29.

10. GREENE, N. AND HECKBERT, P.S. Creating raster omnimax images from multiple perspective views using the elliptical weighted filter. *IEEE CG&A 6,* 6 (June 1986), 21-27.

11. HECKBERT, P.S. Filtering by repeated integration. *Computer Graphics 20,* 4 (August 1986), 315-321.

12. HECKBERT, P.S. Survey of texture mapping. *IEEE CG&A 6,* 11 (Nov. 1986), 56-67.

13. PEACHEY, D.R. Solid texturing of complex surfaces. *Computer Graphics 19,* 3 (July 1985), 279-286.

14. PERLIN, K. An image synthesizer. *Computer Graphics 19,* 3 (July 1985), 287-296.

15. PERLIN, K. *SIGGRAPH '85 Course Notes: state of the art in image synthesis,* July 1985, 297-300.

16. PIXAR. *The RenderMan® Interface Version 3.1,* Sept. 1989, 87-92 and 128-131.

17. REEVES, W.T., SALESIN, D.H., AND COOK, R.L. Rendering antialiased shadows with depth maps. *Computer Graphics 21,* 4 (July 1987), 283-291.

18. WILLIAMS, L. Pyramidal parametrics. *Computer Graphics 17,* 3 (July 1983), 1-11.

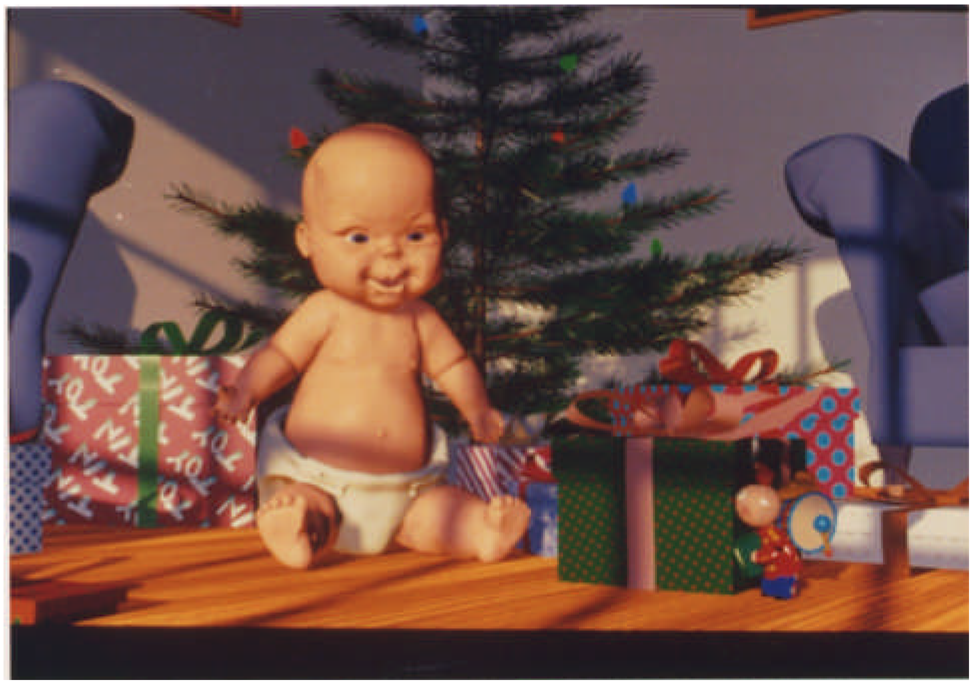

Figure 7: 1988 Pixar Christmas Card by Eben Ostby based on *Tin Toy***.**
Copyright 1988 Pixar Animation Studios

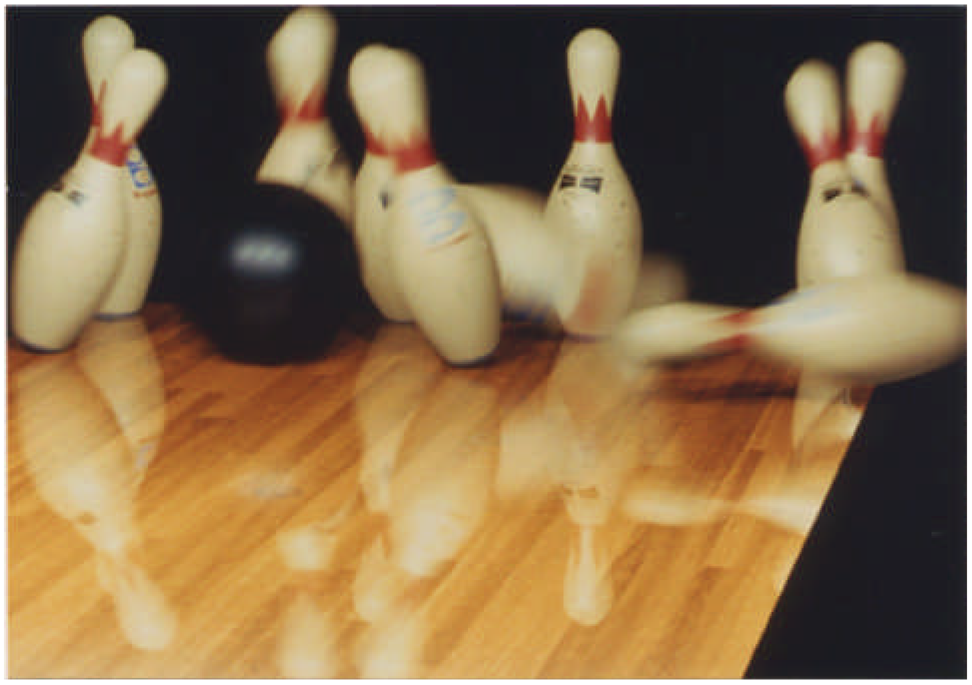

Figure 8: *Textbook Strike* **by Thomas Porter.**
Copyright 1989 Pixar Animation Studios

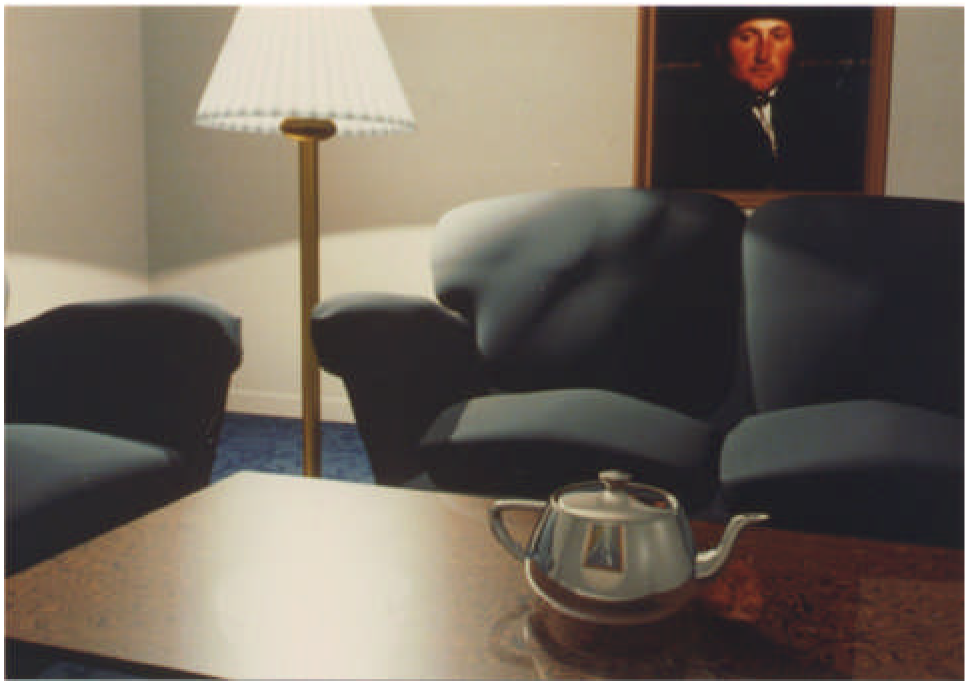

Figure 9: Living room with chrome teapot.
Copyright 1990 Pixar Animation Studios

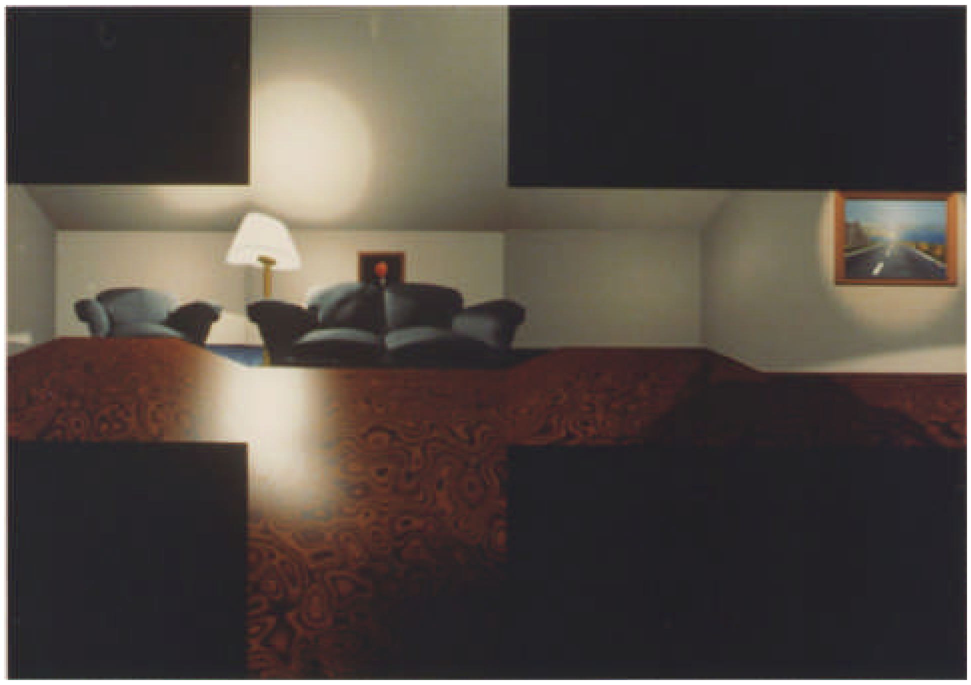

Figure 10: Environment texture used in Figure 9.
Copyright 1990 Pixar Animation Studios