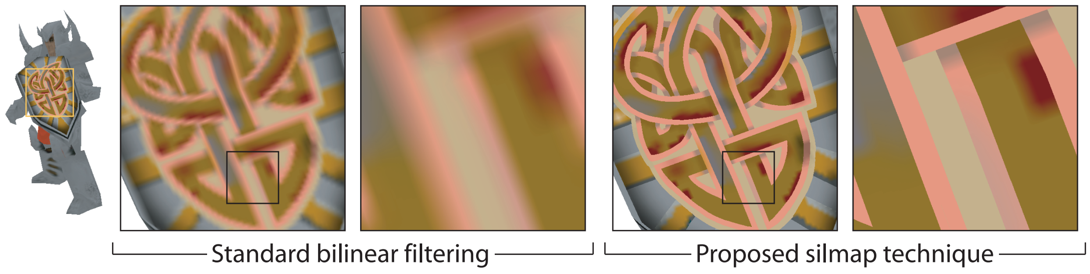
Graphics Hardware (2004)
T. Akenine-Möller, M. McCool (Editors)

Silhouette Maps for Improved Texture Magnification

Pradeep Sen

Stanford University



Abstract

Texture mapping is a simple way of increasing visual realism without adding geometrical complexity. Because it is a discrete process, it is important to properly filter samples when the sampling rate of the texture differs from that of the final image. This is particularly problematic when the texture is magnified or minified. While reasonable approaches exist to tackle the minified case, few options exist for improving the quality of magnified textures in real-time applications. Most simply bilinearly interpolate between samples, yielding exceedingly blurry textures. In this paper, we address the real-time magnification problem by extending the silhouette map algorithm to general texturing. In particular, we discuss the creation of these silmap textures as well as a simple filtering scheme that allows for viewing at all levels of magnification. The technique was implemented on current graphics hardware and our results show that we can achieve a level of visual quality comparable to that of a much larger texture.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture I.3.3 [Computer Graphics]: Picture/Image Generation

1. Introduction

Texture mapping was developed by Catmull as simple a way of improving realism without increasing the geometric complexity of a scene [Cat74]. Since then, texturing has become a ubiquitous primitive in computer graphics, employed with much success in both high-end rendering for feature film as well as real-time graphics applications such as video games.

The sampling process inherent in computer graphics mandates that one must sample these textures carefully or risk introducing unsightly artifacts, a fact noted by Catmull in his original work. These can occur when the rendering samples in screen space coincide poorly with the discrete texture samples, eg. when the texture is magnified or minified on the screen. Although there has been extensive work for ameliorating artifacts during texture minification, relatively little work has been done in the regime where the texture is magnified, particularly for interactive applications. In this paper, we extend the silhouette maps introduced by Sen et al. [SCH03] to reduce the artifacts of textures under magnification. Our solution targets real-time applications and is amenable to acceleration on graphics hardware.

1.1. Previous work

We will begin with a survey of previous work specifically relating to the antialiasing of textures, and thus refer readers who seek an introduction to texture mapping to Heckbert's survey of the subject [Hec86]. As stated in the introduction, most of the previous work deals with textures under minification. One of the seminal papers in this area is Williams's work on mipmaps [Wil83]. A mipmap is a pyramidal data structure that maintains pre-filtered versions of a texture at different resolutions. The base layer contains the texture at full resolution, and each successive layer in the pyramid is $1/2$ the resolution of the layer below it. Texels in upper layers are computed by applying a simple box filter to the $2{\times}2$ texel neighborhood in the layer below it. To query for a filtered texture value, the algorithm must compute the size d of the pixel's projection onto the texture and then use this value to find the two neighboring mipmap levels with texels closest to that size. Texels from these two levels are then each bilinearly interpolated using the u and v texture coordinates, and the two results are linearly interpolated using d. The linear interpolation between levels prevents popping of the texture as the object moves back and forth. The main advantage of mipmaps is that they allow for a constant-time lookup of a filtered texture value regardless of the size of the pixel's footprint. They are also straightforward to implement in graphics hardware and are thus popular in real-time applications.

There have been other proposed approaches for accelerating the filtering of minimized textures for real-time applications (eg. Crow's summed-area tables [Cro84]). However, these approaches have not gained as widespread use as the mipmap algorithm in today's interactive applications.

So far, we have discussed previous work dealing with the regime where the texture is minified, or in mipmapping parlance the region of $d > 0$. Unfortunately, the problem of antialiasing of magnified textures has not received as much attention. The naïve approach is to simply perform a nearest lookup on the texture, resulting in a jagged "pixelated" texture. An improvement of this is to bilinearly interpolate the data, which results in blurry textures but alleviates pixelation considerably. Since this interpolation is fast and simple to implement (and supported natively on modern graphics hardware), most interactive applications use this technique despite low quality results. Finally, there is also the possibility of a brute-force approach which would simply store a higher resolution texture to begin with so that the user is never able zoom in beyond the base level in the mipmap. To reduce the memory consumption, compression techniques such as rendering from compressed textures [BAC96] could be implemented. Our proposed solution yields results comparable to that of a larger texture but with a modest memory overhead.

An approach that specifically addresses the magnification problem is SGI's GL_SGIS_sharpen_texture which tries to generate a higher resolution version of the texture by extrapolating from lower resolution versions. Unfortunately, without any more information other than the pixel values this extrapolation is often prone to error.

Schilling et al. describe detail maps in their Texram paper[SKS96], which provide a hierarchical resolution framework that allow certain regions of the texture to be stored at much higher resolutions than others. Unfortunately, because of the potentially large chain of indirect texture references, this method could be difficult to implement efficiently on consumer graphics hardware and can inherently only improve the quality of a few regions of the texture. Similar detail textures were implemented by SGI in their Infinite Reality system.

Another option is to use procedural textures [EMP*94]. Since these have a functional representation as opposed to a bitmap representation, they can be magnified arbitrarily. These textures tend to work well for some natural phenomena such as clouds, but are typically not well-suited for the textures normally found on signs, characters, etc. because it is often impossible to describe their artwork in functional form. They can also be computationally expensive, and all but the simplest are unsuitable for real-time.

This paper uses discontinuity information for improving filtering to reduce aliasing, something that is not new to graphics. One example of this was the successful use of discontinuity meshes for improved radiosity algorithms in the early 90's ([Hec92, LTG92]). On a more related topic, Salisbury et al [SALS96] described a piece-wise linear structure to allow magnification of pen-and-ink illustrations. Their technique is more complicated than the algorithm presented here and is not easily applied to current graphics hardware.

Bala et al. use an edge-and-point image, a representation that stores both samples and discontinuity information, to improve interpolation of sparse samples for high-end rendering [BWG03]. In work done concurrently to our own, Bala et al. extend their idea to texturing [RBW04]. Their technique, known as feature-based textures, approximates discontinuities in each cell with splines and can store several discontinuities in each cell. Feature-based textures target applications requiring high reconstruction quality, and it is not easy to see how the spline intersection and filtering parts of their algorithm could be mapped to graphics hardware.

Finally, in work done concurrently with our own, Tumblin et al. embed boundary information into a bitmap to create what they call bixels[TC04]. Bixels can be thought of as a silhouette map with two bits of extra information to indicate the boundary conditions. This is similar to our proposed extension of silhouette maps. The main difference in the two approaches lies in the reconstruction filter, with bixels using a more complex patch-based scheme. Our algorithm is simpler and has been implemented directly on the graphics hardware.

Our contribution is an algorithm that significantly improves the quality of magnified textures at comparable memory sizes and can be implemented on current graphics hardware. We do this by extending the silhouette map work of Sen et al. [SCH03] to general textures. While the standard bilinear interpolation attempts to reconstruct the original texture by applying a bilinear filter to all the bitmap samples, we improve the reconstruction by enhancing the point samples with a silhouette map that contains boundary information. We modify the original silhouette map algorithm to filter neighboring samples while preserving the discontinuities. Finally, we describe a simple way of antialiasing the texture when the textured object is minified. This results in a texture that can be viewed from up close or from far away without a degradation in quality and with a memory footprint com-

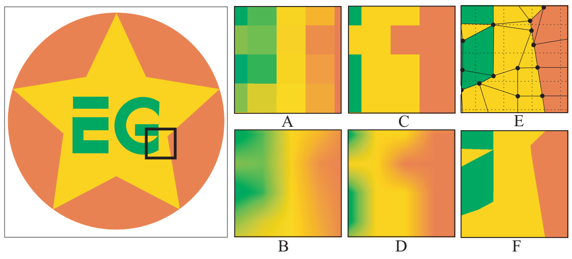


Figure 1: *This design was converted into a texture 32×32 pixels in size. The region indicated is a little over 3 pixels on a side. (A) Filtered bitmap (32×32) generated from the original vector representation. (B) Bilinear interpolation performed on the filtered bitmap during rendering. (C) Closeup of the bitmap used by the silmap algorithm, which is different than (A) because we do not pre-filter across discontinuities. (D) The same bitmap as (C), but this time bilinearly interpolated. (E) Positions of the silhouette points in the texels, with the deformed cells shown. Original silmap cells are in dashed lines. (F) Final result of our algorithm.*

parable to that of standard texturing. We call these silhouette map textures, or *silmap textures* for short.

2. Silhouette maps for textures

Silhouette maps were introduced by Sen et al. as a way to remove the jagged artifacts from the shadow mapping algorithm [SCH03]. Standard shadow maps store visibility from the light source in a depth texture which is then projected onto the scene during the final rendering pass. This projection often enlarges the depth texture in screen space, magnifying the depth texels and resulting in visible jagged contours along shadow boundaries.

The shadow silhouette map algorithm ameliorates these boundary artifacts by storing additional information about the position of the shadow edge in a data structure with fast, constant-time lookup. Specifically, a silhouette map is a uniform grid structure that stores the coordinates of single point in each cell. Every cell has a default point at the cell's center, (0.5, 0.5) in local coordinates, except when the silhouette boundary of an object passes through the cell. In this case, the point is placed on the silhouette boundary, and collectively the points form a piece-wise linear approximation to the shadow contour. This yields a better approximation of the true shadow contour than the piece-wise constant approximation achieved with standard shadow maps. To shade a point in the final rendering pass, it is first projected into the appropriate silhouette map texel. Lines connecting the cell's silhouette point to those of its four neighbors divide the cell into four *skewed quadrants*, and the point must be shaded according to the depth test result of its skewed quadrant.

At the end of the paper, the authors observe that a silhouette map can be thought to deform a regular grid to fit discontinuities. In this paper, we build on this idea and apply

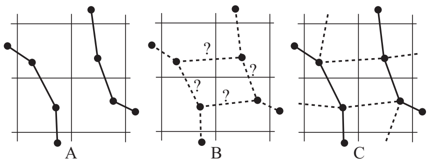


Figure 2: *(A) Situation in which two discontinuities are in close proximity to each other. (B) If only the discontinuity positions are stored in each cell, one cannot determine the true orientation of the boundaries. (C) Our solution is to store an extra bit for each edge to indicate whether the edge is valid. Here the valid edges are shown in a solid line.*

silhouette maps to general texturing. Without much modification, the silhouette map algorithm can perform nearest-neighbor lookups for textures, just like it did for the depth map. To do this, we think of the silhouette map as a more flexible structure than originally envisioned by Sen et al. Instead of storing scalar depth values in the cells of this deformed grid, we can easily store RGB color values (or any kind of values, for that matter). Thus, to render silmap textures we need two data structures at run-time: a regular bitmap, and a silhouette map that contains the edge information of the texture. As we will discuss later, these can be generated manually or automatically acquired from real photographs. The bitmap and the silhouette map are of the same resolution, and the bitmap is shifted 1/2 pixel in x,y with respect to the silhouette map so that the samples of the bitmap lie on the corners of the silhouette map cells.

During the rendering pass, the hardware projects the point to be shaded into the deformed bitmap and the deformed cell's sample is used as the final color for the sample. This yields results such as that shown in Fig. 1, where we can see that despite the low-resolution of the silmap texture we still get faithful reconstruction of the original texture. Since each deformed cell can only have one color, this algorithm is better suited for textures with regions of constant colors, such as decals and signs than for general textures. Besides RGB, one can also store alpha values for transparency or other variables so that shaders can be specified with high resolution using a relatively low-resolution data structure. Details on the implementation of this algorithm can be found in a later section.

2.1. Filtering

The nearest-neighbor algorithm is satisfactory for only a small number of applications. The problem with it is evident when we apply the algorithm to more complex textures (Fig. 5). If we do not filter neighboring samples together, we see that the interior regions appear pixelated. To reduce these artifacts, we must filter across a neighborhood of samples while being careful not to filter across established discontinuities. We must also keep the samples localized enough so that they can be accessed quickly on graphics hardware.

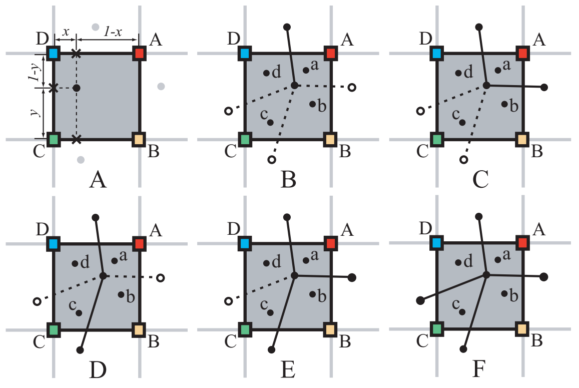


Figure 3: *Cases requiring different interpolation schemes. Corner samples representing bitmap entries are labeled A, B, C, D and random points are shown in each skewed quadrant to facilitate discussion. (A) Cell without boundaries, so the color is determined by bilinearly interpolating the four samples. (B) Cell contains one boundary segment. (C,D) Cell contains two boundary segments. (E) Three boundary segments, and finally (F) all four border segments.*

In order to avoid filtering across discontinuity boundaries, we must be able to define these boundaries in our silhouette map. The problem with the silmaps of Sen et al. is that they contain the position of potential discontinuities but do not indicate what is a real boundary [SCH03]. In their implementation, the depth tests at the corners are used to establish boundary information since shadow boundaries can only occur along an edge where the two depth tests give conflicting results. In our situation we do not have these tests so we must add additional information to the silhouette map structure to indicate these boundaries. One might be tempted to address this problem by only having silhouette points inside cells with valid discontinuities and empty cells elsewhere, with the hope of extrapolating boundary information by looking at neighboring cells. This approach does not work, however, since it can yield ambiguous situations (Fig. 2). The fundamental problem is that while the position of the discontinuity can be interpreted as a property of the cell, the connectivity of the discontinuity boundary of the cell with its neighbors is a property of the edges. Therefore, there can be up to 4 boundary edges in each cell.

Our solution is to add two extra bits per cell, which are then associated with two of the four boundary segments and indicate whether they represent a discontinuity or not. Only two of the four segments need to be labeled per cell because we can access the information on the other two from neighboring cells. This method specifies the boundaries of the texture explicitly and breaks the texture up into *regions*, sets of points located on the same side of the discontinuity that must be filtered together. In this paper, we will refer to the boundaries of a cell as the north, south, east, and west boundaries depending on the neighbor they connect to.

Initially we experimented with storing a region code at each corner of the cell to indicate its region instead of explicitly defining the boundaries. As in the shadow application, the boundaries could then be deduced from this information since corners with different region codes would have an implicit boundary in between them. While this allowed the re-use of the nearest neighbor lookup code to fetch the region information in the same way we fetched the color information in the previous section, problems were encountered when a region curved around and had a boundary with itself. This required arbitrarily breaking up a continuous region into two different regions in order to preserve the boundary. Because of this, we found it better to explicitly express the boundaries in each cell.

When rendering with a silmap texture, we must also determine how many samples from the bitmap to use in texture reconstruction. More complicated reconstruction schemes use samples from neighboring cells [RBW04, TC04], but this makes them more difficult to implement on graphics hardware. In this work, we decided to keep the samples localized and will work only with the four corner color samples of a silhouette cell. Not only is this realizable in current hardware, but it is fast and provides reasonable results. For the same reason, we chose bilinear interpolation as our filter basis. It works well for real-time applications and is supported natively in the hardware, a fact that we used to our advantage in the implementation.

At this point we can describe the full silhouette map texture algorithm with filtering. First, the *uv* coordinates of the point to be textured are computed to project it into a specific cell of the silmap. We must then decide which of the four corner samples are in the same region as the current sample by using our boundary information (Fig. 3). Once we have decided which corners to include in the region, we must apply the appropriate filter to the samples to get the final color value.

First, we determine which corner samples are available for filtering by following these rules:

1. A point is always in the same region as its respective corner. For example, in Figure 3, point *a* will always include corner sample *A* in its filter kernel.
2. Corner samples sharing a side with the point's corner will be included only if there is not a boundary in the way. For example, point *a* will use corner sample *B* only if there is no *east* boundary. Likewise it will include corner sample *D* if there is no *north* boundary.
3. The corner sample directly across the point's corner will be included only if there is a path there that does not intersect boundaries. For the case of point *a* we can write the decision to include corner sample *C* as the following boolean expression:

```
inc_C = ((!b_N && !b_W) || (!b_E && !b_S))
```

   where boolean *b_N* is a 1 if there is a *north* boundary, for example. Another way to say this is that corner sample *C* will be included if we can move counter-clockwise from A without hitting a boundary (*north* and *west* boundaries are both 0) *or* we can move clockwise without hitting a boundary (*east* and *south* boundaries are both 0).

To better understand this algorithm, we examine specific examples from Figure 3. In Fig. 3B, we can see that point d will include sample D by the first rule. In addition, it will include C and dismiss A by rule 2. Finally, since we can access B from D (because the *west* and *south* boundaries are both 0), B will be included as well. Thus, point d will filter between three corner samples, B, C, and D. In the same diagram, it can be seen that point c will include all four corner samples by following the rules stated. This means that in this example we could end up with a shading discontinuity along the non-existent *west* boundary because different filters are used on either side of the "boundary." While this is an artifact of the simple filtering algorithm, we note that in practice the results are reasonable and schemes that guarantee not to have these artifacts for any case are difficult to implement for real-time.

At this point we have determined how many corners (from one to all four) we will use in the interpolation. Because we are using a bilinear basis for filtering, several of the cases become straightforward to implement on graphics hardware. For the 1-corner case (eg. point b in Fig. 3E), we simply use the value of the appropriate corner. For the 2-corner case (eg. point c in Fig. 3D), we linearly interpolate between the two corner values, and with the 4-corner case we perform the full bilinear interpolation.

The 3-corner case requires special consideration, however. Suppose we wanted to find the value for point c in Fig. 3C using samples B, C, and D. We first compute the plane going through these samples by performing the cross products of the two vectors

$$\vec{CB} = (1, 0, C_B - C_C), \vec{CD} = (0, 1, C_D - C_C)$$

where C_B indicates the color at corner B. The cross product of them yields the normal

$$\begin{aligned} \vec{N} = \vec{CB} \times \vec{CD} &= \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ 1 & 0 & C_B - C_C \\ 0 & 1 & C_D - C_C \end{vmatrix} \\ &= -(C_B - C_C)\vec{i} - (C_D - C_C)\vec{j} + \vec{k} \end{aligned}$$

and so we have a plane equation which is equal to:

$$\vec{N} \begin{bmatrix} x - x_0 \\ y - y_0 \\ z - z_0 \end{bmatrix} = \vec{N} \begin{bmatrix} x \\ y \\ z - C_C \end{bmatrix} = 0$$

and which simplifies to

$$z = (1 - x - y)C_C + xC_B + yC_D$$

This is also the same result as barycentric coordinates for our triangle in the unit square.

We can now write down the rules for filtering each of the different cases in Fig. 3. Without loss of generality we write down the formulas for the cases specifically as shown. In this paper, we refer to (x, y) to be the local coordinates with respect to the origin of the texel, corner C. The global texture coordinates are (u, v).

Case	Equation
1 corner	C_C
2 corner	$(1-x)C_D + xD$
3 corner	$(1-x-y)C_C + xC_B + yC_D$
4 corner	$(1-y)((1-x)C_C + xC_B) + y((1-x)C_D + xC_A)$

Now we must come up with a way so that all of the different configurations can be handled efficiently in the hardware. In the implementation section, we discuss how we take advantage of the bilinear interpolator available on the hardware to perform this filtering in an efficient manner.

2.2. Mipmapping of silhouette map textures

Since silhouette maps introduce higher frequency components to the original texture, it is important to filter them properly when they are minimized or they will alias when the textured surface is animated.

Our solution to this problem takes advantage of the fast mipmapping capabilities of the current graphics hardware. After the silhouette map and the respective bitmap are created, a pre-processing step determines an average color for every cell in the silhouette map by weighting the corner samples by the area coverage of each of their respective quadrants. This generates a bitmap with filtered colors at the same resolution as the original silhouette map. This texture can then be mipmapped like a standard bitmap. The mipmap is generated from the silmap texture itself instead of simply using the original bitmap to ensure that the two are perfectly aligned.

Because the frequency content introduced by the silmap texture can be quite high, we switched over in some of our examples to the mipmapped version as soon as the screen/texture sample ratio became 1:1 to avoid aliasing. In the implementation section we describe how we use the mipmapping hardware to switch between the silmap and mipmapped textures.

2.3. Authoring of silhouette map textures

Silhouette map textures can be created in many ways. A drawing tool that combines bitmap and vector graphics would allow artists to manually create the silmap directly. Silmap textures can be generated from photographs by either extracting the discontinuity edges through standard computer vision techniques (eg. [Can86]) or manually annotating the edges. Once the edges are annotated, the original image can be filtered down to the appropriate size for the silhouette map while respecting the discontinuity edges. It is essential that one does not filter across discontinuities when generating the bitmap data for the silmap texture (Fig. 6). In a way, the process of generating a silmap texture from an original image is similar to the texturing process.

2.4. Memory usage

The full silhouette map texturing algorithm proposed in this paper requires three pieces of information per texel: the color information from the bitmap, a silhouette point that encodes the position of the discontinuities, and the explicit identification of boundaries.

In order to reduce the memory overhead, the silhouette map and the region map can be packed into a single byte by using three bits for both x and y. This gives us 8 quantized locations in x and y, for a total of 64 possible locations inside the texel. The remaining two bits can be used for directly labeling the edges for discontinuities.

3. Implementation

3.1. Nearest-neighbor silhouette map textures

The nearest-neighbor silhouette map texture algorithm was implemented in ARB_fragment_program code. The algorithm is shown below (including instruction counts):

1. Fetch current silhouette point and 4 neighbors. [5 TEXs]
2. Translate neighboring points into the cell's local coordinate frame. [4 ADDs]
3. Fetch corner color samples. [4 TEXs]
4. Compute line equations for the 4 boundary segments and for 3 of the 4 line segments that connect the point to the corners of the cell. These corner line segments divide 3 quadrants in the cell into 2 slices each. The last quadrant is not tested since it is the default. [7 XPDs]
5. Test the point to shade against each line to see where it lies. [7 DP3s]
6. Move the correct color into the output depending on the quadrant we are in. [2 CMPs/slice $\times$ 2 slices/quadrant $\times$ 3 quadrants (last one is default)]

The explanation as to why we also test against the corner line segments is given in Figure 4.

3.2. Filtered silhouette map textures

For the hardware implementation of the filtered silmaps, we must first determine which corners are in the same region as the sample point. Since we do not have conditional execution, we must assume that our point can be any of the generic points a, b, c, or d in Fig. 3. A *reachability* vector (to use Bala's terminology [BWG03]) V is computed whose components indicate which corner samples should be included in the order $\{A,B,C,D\}$. This vector is generated for each of the generic points, and then the nearest neighbor algorithm is used to determine which skewed quadrant the point is in

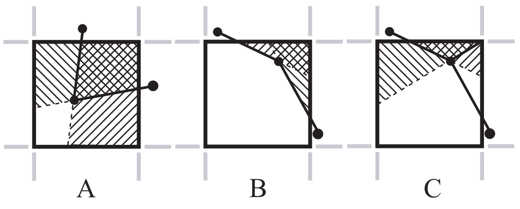


Figure 4: *Taking the dot product of the point to shade with the boundary lines to identify the quadrant only works when the angle formed by these lines is less than 180°. In this diagram, the hatched patterns indicate the regions where the dot product is of the same sign for each of the two boundary lines. (A) When the angle is less than 180°, the entire skewed quadrant is covered by the two tests, therefore we can simply AND the result of the two tests to determine if the point is in this quadrant. (B) If the angle becomes greater than 180°, however, the cross-hatch does not cover the entire quadrant and our test would give false negatives in regions with only one hatched pattern. To avoid these problems, we also test against the lines to the corners of the cell, turning each quadrant into two pie-shaped slices. (C) Here we show the results of testing against the first slice in the NE quadrant. This slice is guaranteed to be covered by the two tests.*

in order to move the correct V into the proper variable to be used in the filtering portion of the algorithm. As an example, the reachability vector for point a would be:

```
Va.x = 1 //A is always reachable
Va.y = !b_E // include B if no east boundary
// the C corner is more complicated, as we discussed...
Va.z = (!b_N && !b_W) || (!b_E && !b_S)
Va.w = !b_N // include D if no north boundary
```

Similar vectors can be written for the other quadrants. We must now implement the filtering equations $f(x,y)$ for each of the cases in an efficient manner for all the possible reachability vectors. We observe that it is possible to accelerate the computation in every one of these cases by using the bilinear interpolation already available in the hardware:

$$g(x,y) = (1-y)((1-x)C_C + xC_B) + y((1-x)C_D + xC_A)$$

In the 1-corner case we simply want to use the color sample at the corner. Suppose that this is corner C. We can access this value with $g(0,0)$. If we have a 2-corner situation, say corners B and C, we can access the result of the linear interpolation through $g(x,0)$. Finally, the 4-corner computation is the same as simply accessing $g(x,y)$ directly.

The 3-corner case requires more work. Take as an example point c in Fig. 3C. We have determined that for this case, $f(x,y) = (1-x-y)C_C + xC_B + yC_D$. We note that:

$$g(x,0) = (1-x)C_C + xC_B$$

$$g(0,y) = (1-y)C_C + yC_D$$

Configuration				
A	B	C	D	f(x,y)
0	0	0	0	invalid
0	0	0	1	g(0,1)
0	0	1	0	g(0,0)
0	0	1	1	g(0,y)
0	1	0	0	g(1,0)
0	1	0	1	invalid
0	1	1	0	g(x,0)
0	1	1	1	g(x,0) + g(0,y) - g(0,0)
1	0	0	0	g(1,1)
1	0	0	1	g(x,1)
1	0	1	0	invalid
1	0	1	1	g(x,1) + g(0,y) - g(0,1)
1	1	0	0	g(1,y)
1	1	0	1	g(x,1) + g(1,y) - g(1,1)
1	1	1	0	g(x,0) + g(1,y) - g(1,0)
1	1	1	1	g(x,y)

Table 1: *Computing the interpolated value using only g(x,y)*

which when added give us

$$\begin{aligned} g(x,0)+g(0,y) &= (1-x)C_C+xC_B+(1-y)C_C+yC_D \\ &= 2C_C-xC_C-yC_C+xC_B+yC_D \end{aligned}$$

so to get f(x,y) for this configuration, we need to simply subtract out the extra C_C term:

$$\begin{aligned} f(x,y) &= g(x,0)+g(0,y)-g(0,0) \\ &= C_C-xC_C-yC_C+xC_B+yC_D \\ &= (1-x-y)C_C+xC_B+yC_D. \end{aligned}$$

Thus it is possible to preform the 3-corner interpolation by performing a linear interpolation of the two sides and subtracting out the sample at the right-angle corner. This allows us to derive Table 1 which shows how to compute $f(x,y)$ using only $g(x,y)$ for all possible configurations. The 1's and 0's in the table indicate which corners are in the same region as the current sample. There are three invalid entries in this table. The corner the sample is in is automatically included, so we can ignore the all-zero case. The other two invalid configurations cannot occur in the silhouette map.

The table shows that the interpolated value can be accessed with one texture lookup for the 1, 2, and 4-corner cases and with 3 texture lookups for the 3-corner case. In the 3-corner case, we must be careful since it is possible that $f(x,y)$ could be bigger than any of the three colors involved (this can happen when the sample is outside the triangle formed by the three corners). In this case we perform a per-component clamp to the maximum component of the two corners on the diagonal of the triangle. For example, if we are dealing with the 3-corner situation we have looked at before, we would clamp to $max(C_B,C_D)$, where the *max* is done per-component. We found in initial experimentation that if we did not clamp in this manner, we would have problems in the cases where there was a large gradient between two of the corner samples of the triangle. This would yield a color that was noticeably incorrect yet still in the still in the range 0, 1.

To implement this in an ARB_fragment_program, we constructed the Karnaugh map of the function to compute the coordinates of each $g(x,y)$ term. For example, the x coordinate of the first $g(x,y)$ term is sometimes 0, sometimes 1 and sometimes x. By simplifying the Boolean logic that take the inputs *ABCD* and maps them to one of these values, we were able to come up with expressions that mapped well to the conditional move statements available in the hardware. This yielded a fragment program that was 45 instructions long.

3.3. Mipmapping

The mipmapped versions of the silhouette maps were computed as a pre-process. During rendering, certain portions of the texture might need to be mipmapped while others are being magnified at the same time. In order to switch between the silhouette map texture and the mipmapped version, we first rendered the mipmap levels into a texture in solid colors. This is similar to the way mipmap levels are visualized, but in our case only the base level was of a different color (Fig. 7). To prevent popping as the program switched between the silmap texture and the mipmapped version, we blended between levels to get a smooth transition. Our tests suggested that even without the smooth blending between levels the popping was not noticeable. Once we had established which regions of the screen we wanted to use with which algorithm, we used the this texture to combine the results of a silmap texture pass and a mipmap pass. The final result was a scene that would smoothly switch between the mipmap and the silmap texture as needed.

4. Results

Using a silhouette map editor we developed, we created several silmap textures and scenes to test our algorithm. Our implementation runs on both NVIDIA and ATI hardware at real-time rates. To give some concrete numbers, the knight scene with the full algorithm including blending and mipmapping runs at 70 fps on a pre-release ATI X800XT (500/500MHz). The knight is an animated md2 character and is textured by a silmap 256×256 texels in size. Images from this and other tests are shown throughout the paper. In places where we compare to standard bilinear interpolation, we use a prefiltered version of the bitmap (not the bitmap we incorporate into the silhouette map), since this is what you would normally use in practice and it makes the quantization artifacts less visible. This causes the images to look different even in regions where full bilinear interpolation occurs in both. When we substitute the bitmap from the silhouette map into the standard algorithm, the fully-blended regions are identical as one would expect.

Texture	Resolution	Original size	Mipmapped size	Silmap texture algorithm
Knight	256×256	196.6KB	262.1KB	524.3KB
Teddy Bear	64×64	12.3KB	16.4KB	20.5KB
EG logo	32×32	3.1KB	4.1K	8.2K
Pedestrian X-ing	128×128	49.1KB	65.5K	131.1K

Table 2: *Survey of memory usage for some of the textures we tested. Our algorithm yielded results equivalent to using a standard texture many times larger in size while typically only doubling the amount of memory required.*

Since our algorithm can be considered a form of texture compression, it is of particular interest to study the memory consumption of our new technique. A comparison with standard texture maps is shown in Table 2 for a few textures.

Our results show that our algorithm roughly doubles the size of the memory needed while at the same time providing an image quality of a texture many times larger. Increasing the size of the texture to improve quality is expensive since you can only increase the texture in $\sqrt{2}$ times in each direction before it doubles in size.

Note that the memory consumption of our silmap technique is based on theoretical calculations. In practice we had to use a larger texture because we cannot pack everything into a single byte (fragment programs do not have bitwise operations). If the algorithm had native hardware support then these numbers could be achieved.

5. Discussion

5.1. Artifacts

The artifacts associated with silhouette maps have been discussed by Sen et al [SCH03]. These artifacts occur when the discontinuity has a region of high curvature that cannot be represented by a single point or when multiple discontinuity curves intersect inside a cell. These artifacts can also occur in silmap textures. In addition, the simple interpolation scheme we propose can sometimes cause banding artifacts, especially when two cells with a different number of valid corners are next to each other. Our experimentation demonstrated that these artifacts are not objectionable.

Despite these constraints, it should be emphasized that silmap textures are created as a pre-process and, unlike the silhouette map shadows, they can be modified and adjusted to eliminate all artifacts. Once the silmap texture is artifact-free it is guaranteed to be correct at run-time. When creating the examples of this paper, we found that after some practice it became straightforward to avoid problematic situations.

5.2. Real photographs

The algorithm was tested on real photographs like those in Figures 5 and 6. While the results look reasonable, the sharp discontinuity boundaries tend to give the photographs a cartoon-like quality, especially when viewed under large magnification. The explanation of this phenomenon is not complicated. Natural photographs are composed of wide range of different frequencies. Our algorithm explicitly preserves discontinuities, which are stored as high frequency components in the frequency domain. Because we aggressively downsample the image, we filter away all of the middle frequencies and end up with a representation that preserves the low and high frequencies of the original photograph but lacks the mid-range. Our minds automatically interpret these images as being cartoon-like, since cartoons often consist of low-frequency regions of fairly constant color carefully separated from each other through high-frequency outlines. Thus this is not something particular to our algorithm, but inherent to all the algorithms that attempt to explicitly store high-frequency discontinuity information while filtering down the color information (eg. bixels and feature-based textures). This suggests that simply avoiding filtering across discontinuity boundaries is not enough for high quality rendering of enlarged photographs.

5.3. Other applications

The silhouette map textures we propose in this paper can be used for more than just simple texturing. In Figure 9, we use a texture to modulate a procedural noise shader [EMP*94] on a surface. Because the silmap texture gives us a piecewise linear approximation of the discontinuities of the design pattern, even a low resolution texture can give us precise control of the shader on the surface of the object. Another potential application of silmap textures are for billboarding, where the alpha channel is typically a boolean indicating transparency. Using current techniques, billboarded trees appear jagged when viewed from up close. Silmap textures would greatly reduce this problem.

5.4. Hardware acceleration

The silhouette map texture algorithm could benefit tremendously from hardware acceleration, even if only part of it is performed by the hardware. One can imagine a texture unit that automatically performs a nearest-neighbor lookup on a texture that has been enhanced with a silhouette map. Such a unit could be used to automatically generate the images in Figs. 1 and 8 without the need of a fragment program. It could also be accessed by the shader in Fig. 9 as easily as one does a texture fetch. To filter more complex textures, one can write a fragment program that relies on this unit to help compute the reachability vector to decide which corners to include in the filtering operation. Finally, this unit would allow the silhouette map shadow algorithm to access the depth map through the same mechanism, thus eliminating the need for a fragment program in the final pass altogether.

6. Further work and conclusion

Any future work on silhouette map textures should probably begin by exploring the application of the technique to natural photographs. As presented in this paper, our algorithm stores RGB values in each cell which we then interpolate together to get the final image. While RGB is one useful basis, other bases (eg. DCT coefficients or indices to texture patches) might be better suited for reproducing photographs. This could open the door to interesting hybrid techniques for image representation. On the more practical side, there needs to be hardware and software support for silhouette maps if the technique is to be adopted by the game development community. Therefore, a good editor that merges seamlessly into current production pipelines is a must.

This paper presented a simple algorithm that considerably improves the quality of textures under magnification while modestly increasing the memory required. The silmap textures are fairly easy to create and the algorithm runs on current graphics hardware.

7. Acknowledgments

The initial ideas that led to this work were conceived through discussions with Mike Cammarano and Pat Hanrahan while working on the paper on silhouette map shadows. Yasamin Mostofi, Mike Houston, and Tim Foley proofread the paper and gave valuable feedback during writing. Special thanks to Anup Lobo who beta-tested the silmap editor and generated the silmap for Figure 9. Thanks also to Fluff for willing to be photographed. The model for the knight shown throughout the paper is available at www.polycount.com and the engine to render it was based on the NVIDIA shadow volumes demo. We would also like to thank NVIDIA and ATI for providing the hardware and support needed to run our experiments. Pradeep Sen was supported by a research grant from ATI, NVIDIA, and SONY.

References

[BAC96] Beers A. C., Agrawala M., Chaddha N.: Rendering from compressed textures. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), ACM Press, pp. 373–378.

[BWG03] Bala K., Walter B., Greenberg D. P.: Combining edges and points for interactive high-quality rendering. *ACM Trans. Graph. 22*, 3 (2003), 631–640.

[Can86] Canny F. J.: A computational approach to edge detection. *IEEE Trans PAMI 8*, 6 (1986), 679–698.

[Cat74] Catmull E. E.: *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, University of Utah, 1974.

[Cro84] Crow F. C.: Summed-area tables for texture mapping. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (1984), ACM Press, pp. 207–212.

[EMP*94] Ebert D., Musgrave K., Peachey D., Perlin K., Worley: *Texturing and Modeling: A Procedural Approach*. Academic Press, Oct. 1994.

[Hec86] Heckbert P. S.: Survey of texture mapping. *IEEE Comput. Graph. Appl. 6*, 11 (1986), 56–67.

[Hec92] Heckbert P. S.: Discontinuity meshing for radiosity. In *Rendering Techniques '92* (1992), Chalmers D. P. A., Sillion F., (Eds.), Eurographics, Consolidation Express Bristol, pp. 203–216.

[LTG92] Lischinski D., Tampieri F., Greenberg D. P.: Discontinuity meshing for accurate radiosity. *IEEE Computer Graphics and Applications 12(6)* (Nov. 1992), 25–39.

[RBW04] Ramanarayanan G., Bala K., Walter B.: Feature-based textures. In *Proceedings of the Eurographics Symposium on Rendering* (2004), Eurographics Association.

[SALS96] Salisbury M., Anderson C., Lischinski D., Salesin D. H.: Scale-dependent reproduction of pen-and-ink illustrations. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), ACM Press, pp. 461–468.

[SCH03] Sen P., Cammarano M., Hanrahan P.: Shadow silhouette maps. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003) 22*, 3 (July 2003), 521–526.

[SKS96] Schilling A., Knittel G., Strasser W.: Texram: A smart memory for texturing. *IEEE Comput. Graph. Appl. 16*, 3 (1996), 32–41.

[TC04] Tumblin J., Choudhury P.: Bixels: Picture samples with sharp embedded boundaries. In *Proceedings of the Eurographics Symposium on Rendering* (2004), Eurographics Association.

[Wil83] Williams L.: Pyramidal parametrics. *SIGGRAPH Comput. Graph. 17*, 3 (1983), 1–11.

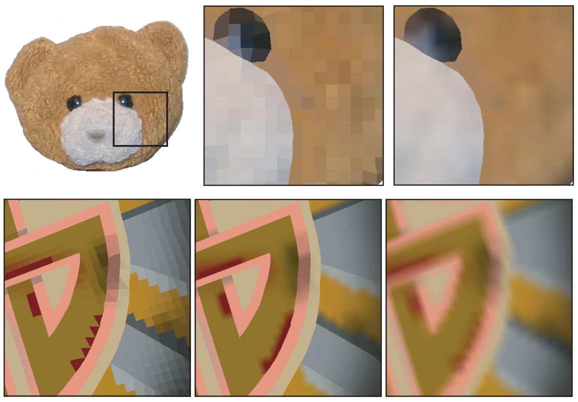


Figure 5: *Performing our algorithm on more complex textures highlights the need for proper filtering. In the top row, a photograph has been turned into a 64×64 silmap and the area indicated magnified to 512×512. The nearest neighbor silmap algorithm is shown on the left, the filtered version of the algorithm on the right. The bottom row shows unfiltered (left) and filtered (middle) closeups of the knight's shield. For reference, standard bilinear filtering disregarding discontinuities is shown on the right.*

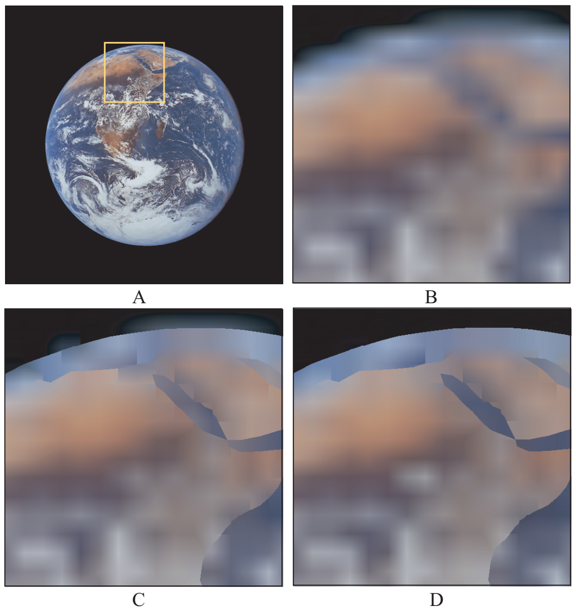


Figure 6: *Filtering a photograph to generate bitmap data for silmap texture. (A) Original photograph (1024×1024). (B) 64×64 pre-filtered bitmap magnified using standard bilinear interpolation. (C) Silmap technique, using a bitmap incorrectly pre-filtered ignoring discontinuities. (D) Results of our algorithm using a 64×64 silmap and properly filtering bitmap data. The improperly filtered image in (C) has a noticeable bluish tint in the space above the Earth and a brownish tint in the Red Sea.*

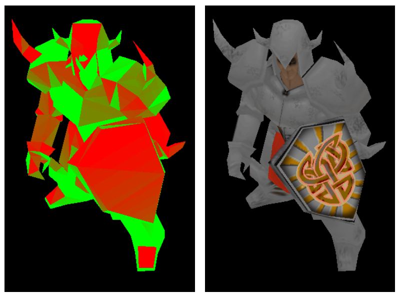


Figure 7: *(Left) The mipmap levels were rendered to switch to the silmap texture when necessary. The red regions indicate the areas that are being magnified. (Right) The result is a smooth blend of the silmap texture with mipmaps.*

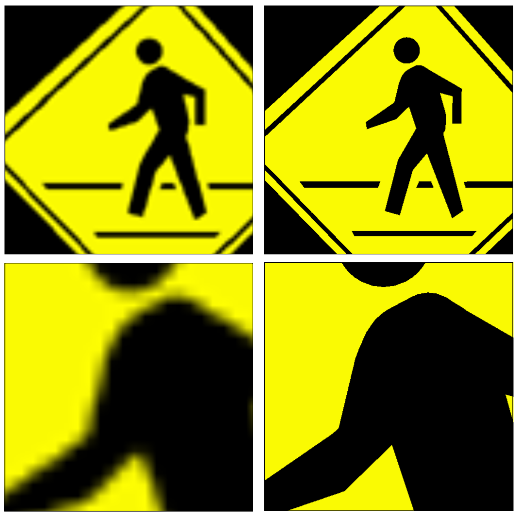


Figure 8: *An example of a sign that might appear in a racing video game, with resolution 128×128. The parts shown are magnified to 512×512. (Left) Standard texturing with bilinear interpolation. (Right) Our technique.*

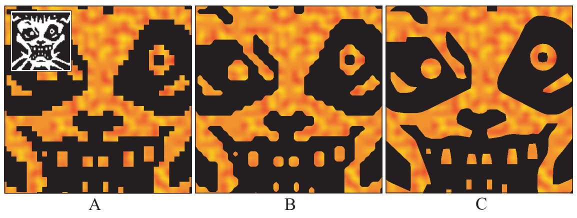


Figure 9: *The 64×64 bitmap shown in the inset is used to modulate a procedural noise shader. (A) Results of performing a standard, nearest-neighbor lookup on the texture. (B) In an effort to improve the quality, we interpolate the pixels and threshold the shader at 0.5, a technique often employed in games to improve the quality of textures used in this manner. (C) Using a silmap texture of the same resolution yields a visibly better result.*