Eurographics Symposium on Rendering 2012
Fredo Durand and Diego Gutierrez
(Guest Editors)

Volume 31 (2012), Number 4

Parameterization-Aware MIP-Mapping

Josiah Manson and Scott Schaefer

Texas A&M University

Figure 1: *From left to right, we show the input texture (1024^2), then the monster frog model drawn with the full-resolution texture and the fourth mip-levels (64^2) downsampled with a box filter that ignores texels that are not used in the model in addition to mipmap guttering and our parameterization-aware bilinear filter. The tent filter does not preserve details as well as our method and allows the background color of the texture to bleed in at texture seams.*

Abstract

We present a method of generating mipmaps that takes into account the distortions due to the parameterization of a surface. Existing algorithms for generating mipmaps assume that the texture is isometrically mapped to the surface and ignore the actual surface parameterization. Our method correctly downsamples warped textures by assigning texels weights proportional to their area on a surface. We also provide a least-squares approach to filtering over these warped domains that takes into account the postfilter used by the GPU. Our method improves texture filtering for most models but only modifies mipmap generation, requires no modification of art assets or rasterization algorithms, and does not affect run-time performance.

Categories and Subject Descriptors (according to ACM CCS): Three-Dimensional Graphics and Realism [Computer Graphics]: Color, shading, shadowing, and texture—

1. Introduction

Movies and games are filled with three-dimensional objects represented as triangle meshes. The geometry of these meshes is important, but much of the detail and interest of an object comes from the variation of colors on its surface. Texture mapping provides a way of annotating surfaces with information such as color. Typically the color at any point on a model is calculated from an image, called a texture, that is applied to the object. Textures are stored as two-dimensional grids of color samples (texels), but there is no obvious way of automatically mapping a point $p \in \mathbb{R}^3$ on a three-dimensional surface to a point $t \in \mathbb{R}^2$ in a texture. Instead, a parameterization $\Theta : \mathbb{R}^3 \to \mathbb{R}^2$ of the object's surface is usually supplied by an artist, where the surface of the object is cut into separate charts that are flattened into the plane. For triangle meshes, Θ is typically encoded as texture coordinates associated with each vertex of the triangles.

The projection of a point p on an object's surface to a pixel s on the screen is given by $\Phi : \mathbb{R}^3 \to \mathbb{R}^2$. To draw a textured surface, the graphics card (GPU) samples the value in the texture associated with a triangle at the coordinate $\Theta \circ \Phi^{-1}(s)$ in order to determine the color of the pixel. How-

ever, a pixel on the screen may correspond to a large area of the texture for distant objects, which necessitates filtering to avoid aliasing. Depending on how distant an object is, a filter h may integrate over many texels. Rather than compute exact filter integrals, GPUs use precalculated downsamplings of the texture that are stored in a mipmap [Wil83].

Filters can be expensive to evaluate with no distortion, but evaluation is even more complicated over the distorted support of $h(\Theta \circ \Phi^{-1}(s))$. Several algorithms have been designed to improve the quality of texture filtering since the invention of mipmaps. The simplest improvement is to generate mipmaps using more advanced filters derived from signal processing. More recently, researchers have focused on designing anisotropic filters that can be evaluated in real-time.

We solve a problem that is related to, but separate from, anisotropic filtering. Anisotropic filtering calculates texture samples by transforming the filter using a first-order approximation of $\Theta \circ \Phi^{-1}$, which is the affine transform defined by the Jacobian of $\Theta \circ \Phi^{-1}$. Describing the local distortion by an affine transformation is correct for a single point, but is only approximate when integrating over the support of h, unless the distortion from screen to texture space, $\Theta \circ \Phi^{-1}$, is uniform. GPUs typically approximate integration of h by sampling from a mipmap. However, at higher mip-levels, texels contain many triangles and the assumption of uniformity of $\Theta \circ \Phi^{-1}$ is violated.

We assume that the parameterization of the model is fixed and improve the texture quality without modifying the model or the base texture. Our observation is that we can correct for nonuniform distortion of Θ during mipmap generation and improve texturing, with or without anisotropic sampling. Our method uses the fact that Θ is view-independent to precompute mipmaps that prefilter the texture to correct for the nonuniformity of the parameterization of a surface in the support of h. Our method calculates corrects for parametric distortions introduced by Θ and show how to optimize texture reproduction when sampled trilinear postfilter. We solve the large-scale problem where parameterization changes between triangles, while anisotropic filtering solves the small-scale problem of sampling points on the screen. Figure 1 shows an example of the improvement obtained by using our method instead of tent filtering. The left model is drawn using the 1024^2 input texture, while the center and right images are drawn with 64^2 tent and parameterization-aware mipmapped (PAM) bilinear filters.

Anisotropic filtering on a GPU still benefits our method because anisotropic filtering adapts to changes of Φ, which are only known at run-time. Thus, anisotropic filtering and our method complement each other. We generate optimized mipmaps as a preprocessing step and improve image quality with no change to art assets or rasterization algorithms and no cost to run-time performance. As an added benefit, our method automatically ignores the unused portion of the texture that forms the background color, which prevents color bleeding at higher mip-levels.

2. Related Work

Mipmapping is a classic technique for improving the performance and quality of texture filtering for real-time rendering. The goal of mipmapping is to accelerate the calculation of downsampled images by arbitrary scales by interpolating between precomputed power-of-two scalings of an image. Trilinear mipmapped texture filtering was first published in 1983 [Wil83] and derives its name from the Latin phrase, "multum in parvo," which means "much in little." Mipmapping has had hardware implementations since the first texturing hardware, and even in the first consumer graphics card, the Voodoo 1, which was introduced in 1996.

The original description of mipmapping generated downsampled images using a box filter, but one can easily imagine using higher-order filters at each level, such as in a Gaussian pyramid [Bur81]. An obvious way to improve the quality of mipmapping is to use high-quality antialiasing filters to generate the images at each mip-level. Shannon's sampling theory [Sha49] states that, before sampling, a signal should be convolved with the *sinc* filter to remove frequencies higher than the sampling frequency. Convolving an image with *sinc* is expensive to compute because *sinc* has infinite support, so *sinc* is often windowed. The Lanczos filters are examples of windowed *sinc* filters.

Hummel [Hum83] described optimal prefilters for linearly-dependent postfiltering bases such as tent and cubic B-splines. However, Hummel only considers functions over an infinite, uniform grid. Kajiya and Ullner [KU81] solve a least-squares problem where intensities are constrained to be in the range $[0,1]$ to render fonts on CRT displays where the pixel response is approximately Gaussian. Least-squares downsampling [ZZZ*11] is a method for finding a downsampling that optimizes over point samples. In contrast, our work finds a downsampling that is optimized over all mip-levels at the same time to match the postfiltering performed by GPUs, handles boundary effects of the image and chart boundaries, and accounts for nonuniform parameterization of the surface during filtering.

Mipmaps work well for sampling high-frequency textures, but do not properly sample triangles with oblique projections onto the screen. Heckbert [Hec89] described the problem of filtering a warped image in its full generality before suggesting two sampling approaches. If the warping is an arbitrary function, one can sample the image at a high resolution and apply a postfilter to resample at the target resolution. This approach is commonly referred to as supersampling, and there has been research on the best sampling patterns to use [DW85, MN88, EDP*11, Fat11]. Postfilters can reduce aliasing or trade aliasing for noise, but because information about the input image and warp is not used, they cannot generate a band-limited signal.

The second approach is to approximate the warp as being affine within the support of the filter. The warp is defined by $\Theta \circ \Phi^{-1}$, and the Jacobian $J(\Theta \circ \Phi^{-1})$ at a single point provides a first order approximation of the warp as an affine transformation. Since the initial description of mipmapping, most subsequent work has been on how to sample images with affine transformations of filters. Elliptical weighted averages (EWA) [GH86] assumes that the pixel sampled on a screen has a radial Gaussian filter. Lin et al. [ZLW06] provide a similar method that samples with a tensor product of windowed *sinc* functions. Another approach is to decompose an anisotropic filter into a weighted sum of isotropic filters that approximate EWA. Feline [MPFJ99] is an approach for estimating an elliptical Gaussian filter by either several radial Gaussian filters or by trilinear filters. A recent paper [MP11] has extended the idea of Feline by building filters from the anisotropic samples provided by GPUs.

There are several approaches to accelerating anisotropic texture filtering. Summed area tables [Cro84] can be used to calculate the contribution from constant rectangular filters. Texture potential mipmapping [CS00] is a multiresolution extension of texture potential mapping [CS97], which stores summed line tables rather than area tables for quickly approximating affine transformations of box filters. Ripmaps [Hec89] are a simple extension of mipmaps, where nonuniform, power-of-two scalings in x- and y-coordinates are precomputed. While mipmaps take only $4/3$ of the original image's space, ripmaps take four times as much space to store. Ripmaps represent scaling in the x- and y- directions well, but are identical to mipmaps along the diagonal. Another method for precalculating anisotropically filtered noise textures is to decompose a texture into directional frequency bands [GZD08]. During sampling, the authors approximate the filtering kernel by taking a linear combination of the texture's frequency components.

Given a filter definition, one should take into account how the texture is mapped onto the surface of a 3D model. A common problem is that colors from unused portions of the texture can bleed through texture seams on the surface of the object. Guttering [WWOH08] can reduce this problem by creating an extended border of similar color around the chart boundary. The process of guttering can be automated through a push-pull algorithm [GGSC96, SSGH01] where texels outside of texture charts are not used during mipmap creation. Then, the image is successively upsampled by factors of two from the lowest resolution, overwriting only the unused texels.

One can also circumvent the problem of surface parameterization by storing textures on the surface itself, such as in Ptex [BL08] and Mesh Colors [YKH10]. Both of these methods implement mipmapping and anisotropic filtering calculated directly on a surface. Alternately, textures can be stored in three-dimensional space around a surface [BD02, LH06] to avoid surface parameterization. In general, these methods tend to be slower than typical texturing, either because they are more complicated or because they have no hardware support. In contrast, our method improves texture sampling when using the native trilinear sampling implemented by GPUs. Some methods [PCK04,RNLL10] also remove seams that result from trilinear filtering by changing the surface parameterization Θ, but do not fix other parameterization-induced filtering artifacts. We improve texture filtering without modifying Θ.

3. Parameterization-Aware Filtering

The mapping Θ between the surface of an object and a texture is often nonuniform, meaning that each triangle can have a different texel density. Additionally, triangles may overlap in texture space and some of the texture space may not be used. Most methods for generating mipmaps filter the texture without regard to how the texture is applied to the surface of an object. We show that image filtering operations can be performed directly on the surface of an object rather than in texture space.

By filtering over the surface of the object, we can enforce that the weight of a texel is proportional to the surface area that it covers on an object. Weighting by surface area means that if a texel is used in two overlapping charts, then the texel will be given twice the weight of a texel used in a single chart. Conversely, if a texel is not used on the surface of an object, it will be given zero weight. All of the filtering is performed in a preprocessing step, which means that the GPU can draw textured objects using completely standard texture sampling operations. Parameterization-aware filtering incurs absolutely no run-time performance penalty and is robust to degenerate inputs such as triangles that have zero area in either object space or texture space.

In typical texture sampling that does not account for parameterization, the coefficient c_h for the sampling filter h is calculated by evaluating integrals over texture space of the form

$$c_h = \iint_{\mathbb{R}^2} h(u,v) \sum_i \hat{c}_i \hat{b}_i(u,v)\, du\, dv,$$

where $\hat{c}_i$ are the coefficients (i.e. colors) of the texels, which are also associated with basis functions $\hat{b}_i$ that comprise the input image $\hat{I}$, where i denotes the translation of the basis functions. Although $\hat{b}_i$ can be arbitrary, GPUs typically multiply texture samples by bilinear basis functions for texture magnification. We therefore use a bilinear basis. In parameterization-aware filtering, we evaluate filters by integrating over the surface Ω of the object

$$c_h = \frac{\iint_\Omega h(\Theta(p)) \sum_i \hat{c}_i \hat{b}_i(\Theta(p))\, dp}{\iint_\Omega h(\Theta(p))\, dp},$$

where p are three-dimensional points on the surface and the functions h and $\hat{b}_i$ are defined in texture space, as before.

In typical texture filtering, no normalization is required because the integral of h is assumed to be unit, but we need a normalization term because the domain of h may be distorted when integrating over a surface. In practice, we evaluate c_h by summing over triangles in a mesh. Each triangle T_k of Ω has a barycentric basis with the triangle coordinates $(0,0),(1,0),(1,1)$. Let $\Gamma_k : \mathbb{R}^2 \to \mathbb{R}^3$ be the map from this barycentric basis to the triangle T_k. We can then write that

$$c_h = \frac{\sum_k \int_0^1 \int_0^x h(\Theta(\Gamma_k(x,y))) \sum_i \hat{c}_i \hat{b}_i(\Theta(\Gamma_k(x,y))) \Delta_k \, dy \, dx}{\sum_k \int_0^1 \int_0^x h(\Theta(\Gamma_k(x,y))) \Delta_k \, dy \, dx}, \tag{1}$$

where Δ_k is the absolute value of the determinant of the Jacobian of Γ_k and is simply equal to the area of triangle T_k.

Because h and b_i are typically piecewise polynomial functions that align with the texel grid, we cut the 3D polygons T_k so that their images $\Theta(T_k)$ in texture space intersect only one texel, which can be done quickly and robustly. Now that both b_i and h have a single polynomial, Equation 1 has a closed-form expression.

Our method integrates directly over 3D triangles by using their image in a barycentric space. Equivalently, we can integrate over the image of triangles in texture space $\Theta(T_k)$, which requires multiplying by $|J(\Theta^{-1})|$, where J is the Jacobian of a transform, instead of Δ_k to account for the change in variables. We say that our mipmaps are parameterization-aware, because we weight texels by their parametric distortion $|J(\Theta^{-1})|$, whereas standard filtering gives all texels equal weight. By integrating over the 3D triangles T_k rather than their image in texture space $\Theta(T_k)$, we avoid problems where 3D triangles in Ω map to degenerate triangles in the texture, which would make $J(\Theta^{-1})$ undefined.

A beneficial property of our optimization is that our method does not interfere with the anisotropic filtering that is performed by GPUs, because we solve a different problem from anisotropic filtering. First, consider when anisotropic filtering produces the correct result. If a square texture is mapped to a rectangle, anisotropic sampling will appropriately stretch the sampling filter by the aspect ratio. In this case, our method is unaffected by the change in parameterization and produces the same result as traditional image filtering, because the Jacobian of the parameterization is uniform and all texels have equal weights. Therefore, the result from anisotropic filtering is unaltered by our method and produces the correct result.

Our method produces different images only when the parameterization is nonuniform, which is exactly when the assumptions of anisotropic filtering are violated. Excluding developable surfaces, flattening a surface will always introduce distortions. Severe distortions within a texel are common at lower-resolution mip-levels, when texels cover many triangles of the surface. We show an example of nonuniform parameterization in Figure 2. In the top row of the figure, we show a flat, triangulated object on the left and its texture on the right. Although blue and green triangles have equal areas in object space, blue triangles use the majority of the texture. This means that the blue color dominates higher mip-levels with standard mipmap generation. In contrast, our method gives blue and green colors equal weight in the distance, while preserving the sharp transition between colors in the foreground. This effect can be seen in the middle row of Figure 2, where the plane turns blue in the distance instead of blue-green. Our method draws the correct color at high mip-levels in the distance. Comparing PAM box filtering in the middle and PAM constrained trilinear filtering on right, the images are similar but PAM trilinear filtering allows less blue to bleed into the green bands.

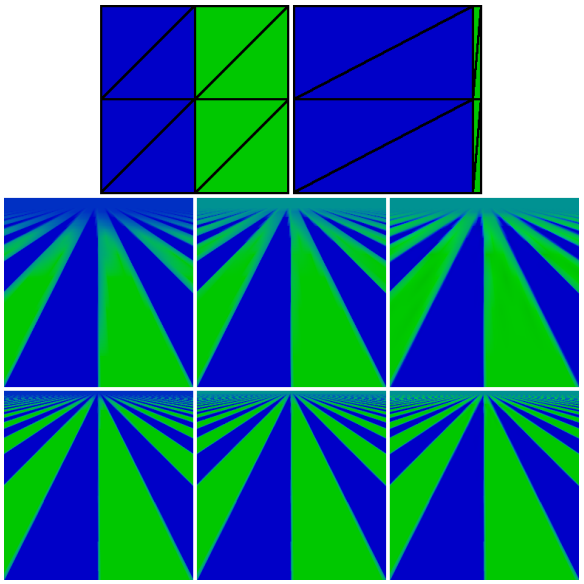

Figure 2: *The top images show the parametric distortion with object space left and texture space right. The middle row shows trilinear mipmapping and the bottom row shows 16x anisotropic mipmap sampling. From left to right: box, PAM box, and PAM constrained trilinear mipmaps.*

In fact, anisotropic filtering and our method are complementary. Because the projection Φ is only known at runtime, anisotropic filtering is required to accurately filter surfaces with respect to their orientation to the camera. On the other hand, anisotropic filtering cannot correctly filter when the Jacobian of Θ changes over multiple triangles. Combining our mipmaps with anisotropic filtering provides a good approximation of $\Theta \circ \Phi^{-1}$ integrated over the entire filter's support. The bottom row of Figure 2 shows how texture filtering is improved by a combination of anisotropic filtering and parameterization-aware mipmapping. Anisotropic filtering samples from higher-resolution mip-levels, which improves image quality, but the plane still turns blue in the distance for the left image. Anisotropic filtering combined

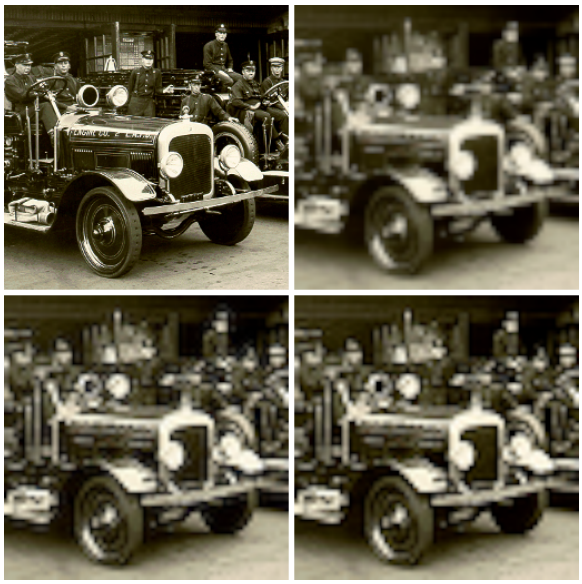

Figure 3: *We show an example of a high-resolution image (top left) downsampled using a box filter (top right), optimized for bilinear reconstruction (bottom left), and optimized for bilinear reconstruction constraining values to [0,1] (bottom right).*

with our method draws crisp lines that correctly appear blue-green in the distance.

4. Optimal Trilinear Approximation

GPUs store downsampled images as a small set of precalculated images, called a mipmap, in order to accelerate downsampling at arbitrary resolutions. The GPU approximates colors at intermediate resolutions by trilinearly interpolating between the color samples of the two closest resolutions. Our goal is to make the color samples calculated by trilinear filtering match the color of the input image $\hat{I}$ as accurately as possible, which we do by solving a least-squares optimization as suggested by Kajiya and Ullner [KU81]. Our contribution is to simultaneously optimize the entire mip-stack so that we find the optimal representation over all resolutions rather than within a single two-dimensional image. This means that, instead of filtering each image in the mip-stack separately, the filtered images are all interdependent. We also incorporate corrections for parametric distortions in our optimization and explicitly handle the effects that chart boundaries have on the optimization, whereas Kajiya and Ullner [KU81] assume that pixels reside on an infinite plane and can all be treated identically.

Performing a least-squares optimization significantly improves downsampled image quality but can introduce ringing. In Figure 3 we show an example of an image that is

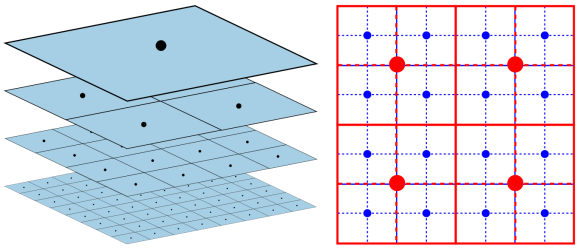

Figure 4: *A mipmap can be visualized as a stack of overlaid images as shown on the left. The alignment between neighboring resolutions is shown on the right.*

downsampled followed by upsampling with a bilinear filter. Comparing the box filter (top right), and least-squares optimization (bottom left) shows that the optimized filter reproduces the original image more accurately and appears less blurry but suffers from some ringing artifacts. As described by Kajiya and Ullner [KU81], some of the ringing results in values that are outside of a monitor's displayable range. Finding the best values to reproduce an image on a physical device requires optimizing for values constrained to [0,1]. We show the results of this constrained optimization on the bottom right of Figure 3. This image appears sharp, with fewer artifacts than the unconstrained optimization.

The trilinear interpolant is best understood by visualizing the mipmap as a stack of images as shown on the left of Figure 4, where basis functions are centered on texels and are shown as black dots. The stack defines a rectangular solid parameterized by u, v, w and the images are evenly distributed in w from 0 to n. Given an input image $\hat{I}$ at resolution 2^n, the mipmap of $\hat{I}$ is a stack of $n+1$ images $I = (I_0, I_1, ..., I_n)$. The images are indexed in order of the distance at which they are displayed so that I_0 has same resolution as $\hat{I}$ and the resolution of I_w is 2^{n-w}. The canonical trilinear basis function is the tensor product of unit tent functions, and the trilinear basis function, $b_j(u,v,w)$, of image w is scaled in u and v by 2^w but is not scaled in w. The index j is a triplet of integers that indicates the translation and scaling of b_j.

We minimize the error of a sample by minimizing the difference between $\hat{I}$ and I over all distances.

$$\min_{c_j} \sum_k \int_{-\infty}^{\infty} \int_0^1 \int_0^x \left| \sum_j b_j(\Theta(\Gamma_k(x,y)),w)c_j - \sum_i \hat{b}_i(\Theta(\Gamma_k(x,y)))\hat{c}_i \right|^2 \Delta_k \, dy \, dx \, dw.$$

The system is large, but sparse, so we solve the system using conjugate gradients implemented in TAUCS [TCR03]. When w is outside of the range $[0,n]$, we reproduce the behavior of GPUs by sampling from the closest available mip-level. For $w < 0$, there is more than one pixel per texel and the image is magnified using bilinear interpolation. The weight from the negative values of w is infinite and adds

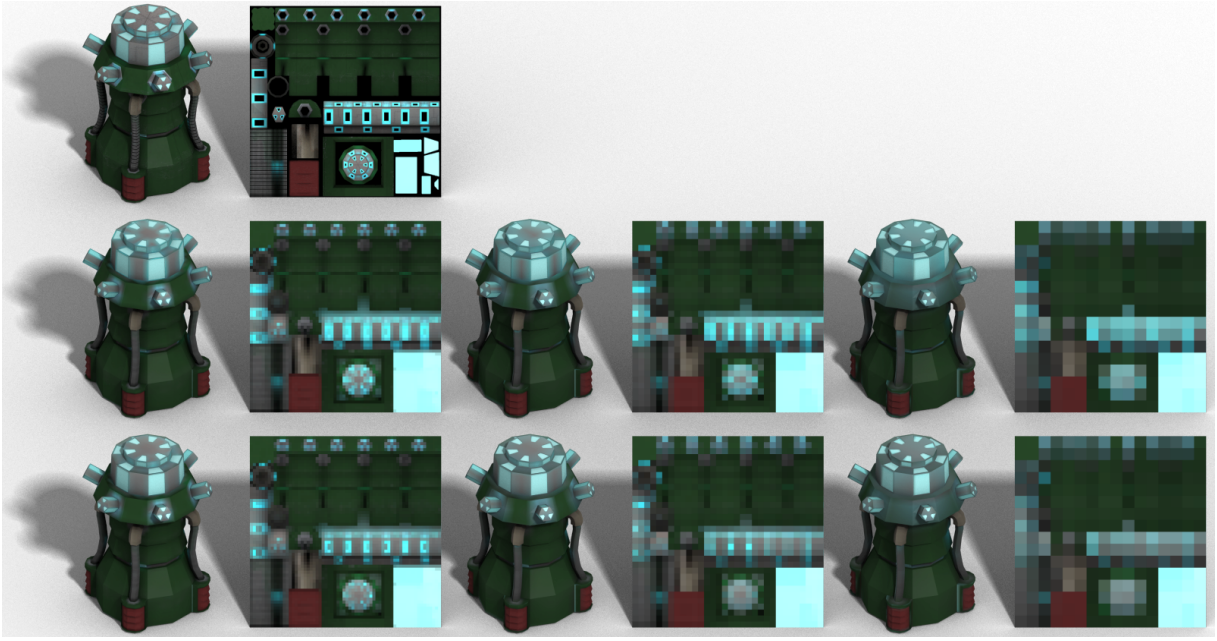

Figure 5: *The top right model is drawn with the full* 1024^2 *input texture. The row of models directly below that are drawn with* 64^2*,* 32^2*, and* 16^2 *box filtered textures that ignore texels that do not intersect triangles. The bottom row uses PAM box filtered textures at the same resolutions and does a better job of preserving the original texture.*

only to the basis functions of I_0, which constrains $\hat{I} = I_0$. Likewise, for $w > n$, more than one copy of the texture may exist in a pixel and we constrain I_n to be the area-weighted average of $\hat{I}$. Constraining I_0 and I_n dramatically reduces the size of the linear system we solve. At a small cost to image quality, we can instead optimize color reproduction for each mip-image individually by using a bilinear basis instead of a trilinear basis. If we use box basis functions instead of bilinear or trilinear functions, the linear system is diagonal and each texel is independent. This means that, for box filtering, the least-squares solution reduces to the filtering described in Section 3 and is quite fast.

We show how the trilinear basis functions of neighboring resolutions overlap on the right side of Figure 4. We use blue for one resolution and red for half the blue resolution. Solid lines show the primal grid, dashed lines show the dual grid, and dots show the centers of basis functions. Because the dual grids over which trilinear basis functions are defined do not nest, we must cut triangles by a grid that is twice the resolution of the high-resolution image (the half-texel grid).

Conceptually, basis functions of color samples from outside of the image's domain can intersect the image. GPUs define multiple methods for dealing with this problem. In OpenGL, borders are treated differently based on the *wrap mode* of the image, of which three are commonly used. One approach is to add a one texel border around the image so that color values are defined for all basis functions that intersect the image. The second and third approaches require no extra storage by defining the border values to be equal to values within the image. If the image tiles, border colors are defined by taking the modulo of a texel's index, while nonrepeating images clamp border colors to be equal to the nearest color in the image. Fortunately, it is simple to match our optimization to the wrap mode used to sample textures. The examples in the paper all use the clamp functionality in the optimization except for Figure 2, which uses wrap mode in the optimization.

5. Results and Discussion

We compare our parameterization-aware filtering using a box filter versus naïve box filtering in Figure 5. The input texture is shown in the top left corner of the image. In this example, there is relatively little unused texture space, so the background color has less of an effect on the filtered images. However, this model exploits symmetry to reuse parts of the texture and contains overlapping charts. For example, there are four gray pipes with red grills in the model, but only one instance of these objects in the texture. Moreover, some of the triangles in the three-dimensional surface map degenerately to lines in the texture.

Our method handles all of these issues properly. The top row shows the model drawn with its 1024^2 input texture. The middle row shows the results from using a standard

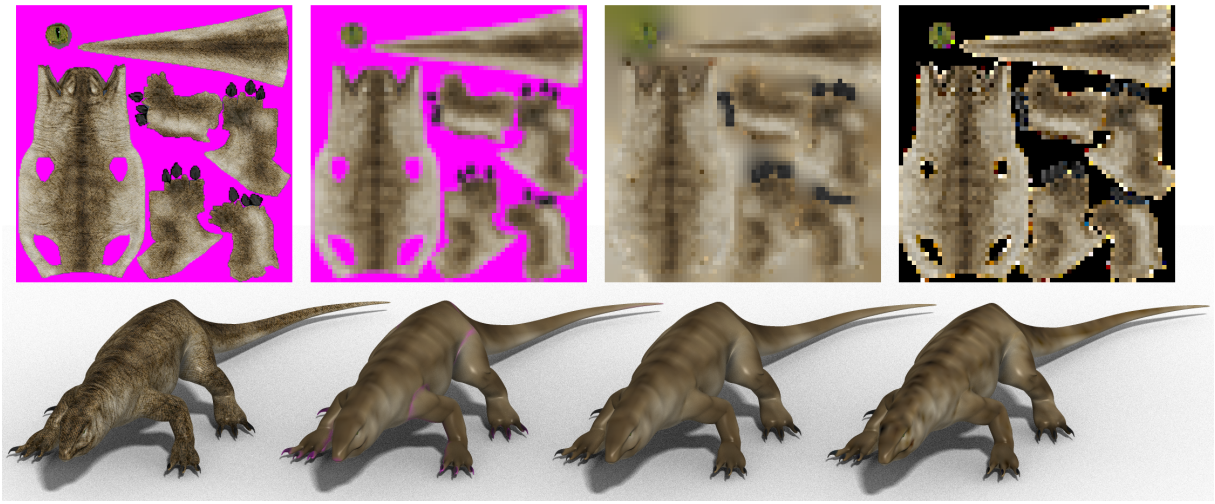

Figure 6: *We show an example model with its* 1024^2 *input texture on the left. The subsequent models are drawn with downsampled textures at* 64^2 *resolution calculated using a box filter, a box filter that ignores texels that do not intersect triangles and uses guttering, and our parameterization-aware bilinear filter. Unused parts of the input texture are magenta to illustrate the color bleeding that occurs with image filtering that does not use information about the model.*

box filter and the bottom row shows the results from using our PAM (parameterization-aware mipmapping) box filter. From left to right, the resolutions of the textures in the middle and bottom rows are 64^2, 32^2, and 16^2. One can see that parameterization-aware filtering significantly reduces the amount of blue that bleeds into the grey at the top of the models. Each of the blue lights is inset slightly, and the wall of the inset is given a disproportionately large fraction of the texture space. Our PAM filter weights these insets proportionally to their surface area rather than their texture area to reproduce the appearance of the original model more faithfully. We should note that our method even handles the case when two triangles with different Jacobians overlap, where a texel will be weighted by the sum of the surface areas that intersect that texel.

A consequence of using Θ to weight downsampled colors by how often they appear on a surface is that unused parts of a texture are given no weight, so that the colors between charts have no effect on downsampled images. We show an example of the effect from using different filters on a lizard model in Figure 6, where we modified the input texture to be magenta between charts in order to emphasize how the background color affects filtering. From left to right, we show the model drawn with its 1024^2 input texture, and models with textures downsampled to 64^2 using a box filter, a box filter that ignores unused parts of the texture, and our PAM bilinear filter. Standard box filtering does not take into account which texels are visible on the model and allows the unused texels colored in magenta to influence the color of texels at seams. Ignoring unused texels in a box filter approximates the effect of our parameterization-aware box filter, but is not sufficient by itself. The support of the box downsampling filter is smaller than the support of the trilinear filter used by hardware to sample the texture, which means that at least a one texel border of similar colors must be added around charts. Our method can directly optimize for trilinear filtering, which takes into account the full support of the sampling filter, fractional texel coverage, and overlapping/degenerate charts.

Our filter preserves the color of the surface, but the tent filter picks up a magenta tint, especially in the last image, where the tent-filtered lizard is almost entirely magenta. The lizard appears slightly lighter in the last images with our method, because the lighter color of the lizard's underside is added to the average color. Moreover, our PAM filter preserves details better than the tent filter, as can be seen by comparing the 256^2 images where the tent-filtered texture is blurrier than the PAM-filtered texture. Although we could have used a different background color to reduce the filtering artifacts, any color will still incorrectly affect the filtering results. There are existing methods for removing texture seams [PCK04, RNLL10] that also prevent the background from affecting the surface color, but these methods do not correct for other artifacts due to parameterization. We show a similar result in Figure 1, where it is clear that our parameterization-aware method preserves details better than conventional texture filtering and eliminates color bleeding across chart boundaries.

In Figure 7, we show a graph of the errors from using a box filter compared to parameterization-aware box, bilinear, and trilinear filters for Figure 2. This example utilizes the entire texture, so differences in the graph are entirely from dif-

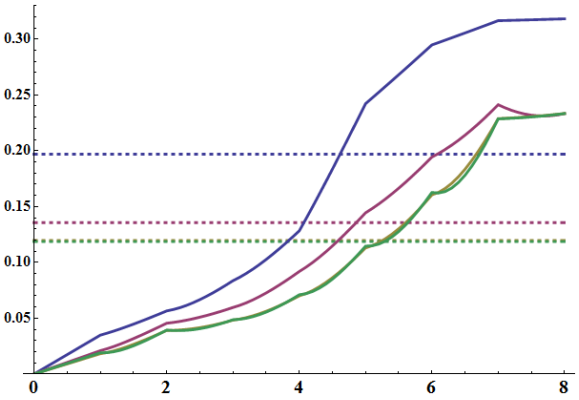

Figure 7: *Graphs of the errors of textures in Figure 2 measured at different mip-values from zero through eight. The filters used are box (blue), PAM box (red), PAM constrained bilinear (yellow), and PAM constrained trilinear (green).*

ferences in parameterization, and not from background color bleeding. The root mean square error (RMSE) of the approximation measured at different mip-values is shown by solid lines and the RMSE over all mip-levels is shown by dashed lines. The PAM bilinear and trilinear filters have noticeably lower error than the box filter. The bilinear filter has lower errors at integer mip-levels than the trilinear filter, but has a slightly higher overall error. One can see that correcting for parametric distortion significantly reduces the RMSE, especially at low resolutions.

We show the times taken to calculate mipmaps using a variety of filters at different input sizes in Table 1, all calculated for the lizard model on an Intel Core i7-2600k. Calculation time mostly depends on the number of texels in charts rather than triangles in the mesh as long as there are fewer triangles than texels, because triangles are cut to a fine grid. We should emphasize that mipmap generation only needs to be performed once, after which results can be stored. Hence, mipmap generation time should not be an issue for most applications. Nevertheless, the worst case of solving for mipmaps that are optimized for trilinear filtering from a 1024^2 source image takes slightly more than one minute. Error graphs suggest that optimizing for bilinear reconstruction is almost as good as optimizing for trilinear reconstruction, but takes a fifth of the time. Therefore, we expect most people to use PAM bilinear filtering instead of PAM trilinear filtering. Parameterization-aware box filtering is very fast because no linear system has to be built or solved. In models with parametric distortion, parameterization-aware box filtering provides a significant reduction in error and costs only a second of preprocessing time.

An important point is that the filtering operations we describe in this paper assume a linear color space. Image formats often use only eight bits per color channel, which is

	64	128	256	512	1024
box	0.004	0.004	0.011	0.034	0.134
tent	0.004	0.008	0.037	0.164	0.697
Lanczos 3	0.033	0.151	0.677	3.064	13.58
PAM box	0.030	0.051	0.124	0.338	1.061
PAM bi.	0.166	0.344	0.993	3.703	15.48
PAM tri.	0.411	1.291	4.699	18.606	75.68

Table 1: *Times measured in seconds to construct mipmaps for the lizard from different input resolutions. Traditional filters that ignore parameterization are shown on top, while our filters are shown on the bottom.*

insufficient precision to represent dark colors well. Therefore, most images are stored in the sRGB color space, which provides more bits for low intensities. This requires that we convert images from sRGB to a linear space, perform our sampling, and convert our results back to sRGB before saving. For rasterization, we perform all calculations in a linear floating point format and use the *EXT_framebuffer_sRGB* extension of OpenGL to rasterize into the sRGB color space.

6. Conclusions and Future Work

This paper presents a method for calculating downsampled images for display as textures in three-dimensional applications. We minimize the difference between a source image and bilinearly or trilinearly interpolated textures and correct for distortions introduced by mesh parameterization. We perform all calculations as a preprocessing step to improve image quality without changing the renderer or reducing runtime performance.

There are several extensions to our method that we would like to investigate. One of the assumptions we made in our method is that mesh geometry is static. In practice, meshes are often animated, which means that the optimal texture is different for each frame of animation. It is trivial to modify Equation 1 to use the average Jacobian over all frames, but how often a frame is displayed may not be known in advance, such as in a game. In practice, most realistic objects undergo nearly isometric deformations, and our method will perform well even when using a single, static pose.

Acknowledgements

This work was supported in part by NSF CAREER award IIS 1148976 and the NSF Graduate Research Fellowship Program. We would also like to thank Bay Raitt for the model of the monster frog, Murat Afsar for the model of the lizard, and the Los Angeles Fire Department for the image of the fire truck.

References

[BD02] BENSON D., DAVIS J.: Octree textures. In *Proceedings of ACM SIGGRAPH* (2002), pp. 785–790. 3

[BL08] BURLEY B., LACEWELL D.: Ptex: Per-face texture mapping for production rendering. In *Eurographics Symposium on Rendering 2008* (2008), pp. 1155–1164. 3

[Bur81] BURT P.: Fast filter transform for image processing. *Computer Graphics and Image Processing 16*, 1 (1981), 20–51. 2

[Cro84] CROW F. C.: Summed-area tables for texture mapping. In *Proceedings of ACM SIGGRAPH* (1984), pp. 207–212. 3

[CS97] CANT R., SHRUBSOLE P.: Texture potential mapping: A way to provide antialiased texture without blurring. In *Visualization and Modelling* (1997), pp. 223–240. 3

[CS00] CANT R., SHRUBSOLE P.: Texture potential mip mapping, a new high-quality texture antialiasing algorithm. *ACM Transactions on Graphics 19*, 3 (2000), 164–184. 3

[DW85] DIPPÉ M. A. Z., WOLD E. H.: Antialiasing through stochastic sampling. In *Proceedings of ACM SIGGRAPH* (1985), pp. 69–78. 2

[EDP*11] EBEIDA M. S., DAVIDSON A. A., PATNEY A., KNUPP P. M., MITCHELL S. A., OWENS J. D.: Efficient maximal poisson-disk sampling. *ACM Transactions on Graphics 30* (2011), 49:1–49:12. 2

[Fat11] FATTAL R.: Blue-noise point sampling using kernel density model. *ACM Transactions on Graphics 28*, 3 (2011), 48:1–48:10. 2

[GGSC96] GORTLER S. J., GRZESZCZUK R., SZELISKI R., COHEN M. F.: The lumigraph. In *SIGGRAPH* (1996), pp. 43–54. 3

[GH86] GREENE N., HECKBERT P.: Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics and Applications 6*, 6 (1986), 21–27. 3

[GZD08] GOLDBERG A., ZWICKER M., DURAND F.: Anisotropic noise. In *Proceedings of ACM SIGGRAPH* (2008), pp. 54:1–54:8. 3

[Hec89] HECKBERT P.: *Fundamentals of Texture Mapping and Image Warping*. Master's thesis, University of California, Berkeley, 1989. 2, 3

[Hum83] HUMMEL R.: Sampling for spline reconstruction. *SIAM Journal on Applied Mathematics 43*, 2 (1983), 278–288. 2

[KU81] KAJIYA J., ULLNER M.: Filtering high quality text for display on raster scan devices. In *Proceedings of ACM SIGGRAPH* (1981), pp. 7–15. 2, 5

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. In *Proceedings of ACM SIGGRAPH* (2006), pp. 579–588. 3

[MN88] MITCHELL D. P., NETRAVALI A. N.: Reconstruction filters in computer-graphics. *ACM Computer Graphics 22* (1988), 221–228. 2

[MP11] MAVRIDIS P., PAPAIOANNOU G.: High quality elliptical texture filtering on gpu. In *Symposium on Interactive 3D Graphics and Games* (2011), pp. 23–30. 3

[MPFJ99] MCCORMACK J., PERRY R. N., FARKAS K. I., JOUPPI N. P.: Feline: Fast elliptical lines for anisotropic texture mapping. In *Proceedings of ACM SIGGRAPH* (1999), pp. 243–250. 3

[PCK04] PURNOMO B., COHEN J. D., KUMAR S.: Seamless texture atlases. In *Proceedings of SGP* (2004), pp. 65–74. 3, 7

[RNLL10] RAY N., NIVOLIERS V., LEFEBVRE S., LÉVY B.: Invisible seams. In *Proceedings of EUROGRAPHICS Symposium on Rendering* (2010). 3, 7

[Sha49] SHANNON C.: Communication in the presence of noise. *Proceedings of the IRE 37*, 1 (1949), 10–21. 2

[SSGH01] SANDER P. V., SNYDER J., GORTLER S. J., HOPPE H.: Texture mapping progressive meshes. In *SIGGRAPH* (2001), pp. 409–416. 3

[TCR03] TOLEDO S., CHEN D., ROTKIN V.: TAUCS 2.2. http://www.tau.ac.il/ stoledo/taucs/, 2003. 5

[Wil83] WILLIAMS L.: Pyramidal parametrics. In *Proceedings of ACM SIGGRAPH* (1983), pp. 1–11. 2

[WWOH08] WANG H., WEXLER Y., OFEK E., HOPPE H.: Factoring repeated content within and among images. In *SIGGRAPH* (2008), pp. 14:1–14:10. 3

[YKH10] YUKSEL C., KEYSER J., HOUSE D. H.: Mesh colors. *ACM Transactions on Graphics 29*, 2 (2010), 1–11. 3

[ZLW06] ZHOUCHEN LIN L. W., WAN L.: First order approximation for texture filtering. In *Pacific Graphics Poster* (2006). 3

[ZZZ*11] ZHANG Y., ZHAO D., ZHANG J., XIONG R., GAO W.: Interpolation-dependent image downsampling. *IEEE Transactions on Image Processing 20*, 11 (2011), 3291–3296. 2