Patch Textures: Hardware Support for Mesh Colors

Agatha Mallett, Larry Seiler, and Cem Yuksel

Abstract—*Mesh colors* provide an effective alternative to standard texture mapping. They significantly simplify the asset production pipeline by removing the need for defining a mapping and eliminate rendering artifacts due to seams. This paper addresses the problem that using mesh colors for real-time rendering has not been practical, due to the absence of hardware support. We show that it is possible to provide full hardware texture filtering support for mesh colors with minimal changes to existing GPUs by introducing a hardware-friendly representation for mesh colors that we call *patch textures*, which can have quadrilateral or triangular topology. We discuss the hardware modifications needed for storing and filtering patch textures, including anisotropic filtering. This paper extends our previous work by discussing and comparing patch edge-handling approaches, including an option for sampling the textures of neighboring patches using an adjacency map. We also provide extensive discussions regarding data duplication, a partial implementation present in existing hardware, and the difficulties with providing a similar hardware support for Ptex.

Index Terms—Textures, Mesh Colors, Ptex, GPU Hardware, Texture Filtering, Barycentric Filtering, Anisotropic Filtering.

1 Introduction

Texture mapping is the standard method of adding data to a 3D object at a resolution higher than the underlying geometric detail. Unfortunately, texture mapping suffers from the problem that it requires a mapping from object space to texture space. For most object types, this mapping distorts the model's geometry and has discontinuities where the surface has been "cut" into separate pieces so as to lie flat within texture space, introducing *seams*. Packing these separate pieces into the texture space can also leave gaps of wasted space. These facts pose a challenge to technical artists, who must expend inordinate amounts of time mapping 3D objects. They must also deal with the limitations of texture mapping. For example, it is very difficult to increase texture resolution in a particular spatial region of the model after the texture has been painted.

Furthermore, these problems can also cause rendering artifacts along seams, where the filtering operations on either side of an edge produce inconsistent results. Methods that "hide" the seams by strategically placing them in texture space introduce additional difficulties for content creation, and struggle with mipmapping and anisotropic filtering. The task of resolving these rendering problems is therefore also left to the artist authoring the texture, and it often leads to overpainting by introducing additional gaps in packing and thereby wasting more memory. More-importantly, such manual fixes cannot completely eliminate filtering inconsistencies, and artifacts still show up in higher mipmap levels and lead to cracks on surfaces with displacement mapping.

One approach to resolving these issues has been to redefine the texture data to live directly on the mesh geometry itself. This is the approach taken by two alternatives to texture mapping: *ptex* [1] and *mesh colors* [2]. Although these methods have been extensively used in offline production rendering, neither technique has been adequately adapted for real-time rendering. This is primarily because there is no hardware support for these methods and software texture filtering implementations can be an order of magnitude slower. Recently, *mesh color textures* [3] were introduced for utilizing the existing GPU hardware for partially handling the filtering operations of mesh colors. This was achieved by converting mesh color data into a standard 2D texture. However, this conversion is not always exact, and it may require solving a complex optimization problem for generating coarser mipmap levels. Mesh color textures also introduce a substantial amount of shader complexity and they cannot handle anisotropic filtering.

In this paper, we show that providing full hardware support for mesh colors could be achieved by relatively minor modifications to existing texture storage and filtering operations of current GPUs. We introduce *patch textures*, a hardware-friendly representation of mesh colors, and describe the details of how patch textures could be stored and used with various filtering operations, including mipmapping and anisotropic filtering. We discuss the similarities and differences of alternative hardware implementations of patch textures, as-compared to standard 2D textures. In particular, we describe the differences in the sample positions in texture space, packing data for triangular patches, and storage requirements for mipmap levels. We also explain how the existing bilinear filtering operations could be modified for supporting barycentric filtering and we present different approaches for handling anisotropic filtering.

This paper extends our previous work [4] by describing an implementation of edge-crossing operations that requires more-significant hardware changes but simplifies the use of patch textures within algorithms that require accessing textures of multiple patches, such as parallax occlusion mapping [5]. This is achieved using hardware-managed adja-

• *A. Mallett and C. Yuksel are with the School of Computing, University of Utah, Salt Lake City, UT, 84112. L. Seiler is with Facebook Reality Labs. E-mail: agatha@geometrian.com, larryseiler@fb.com, cem@cemyuksel.com*

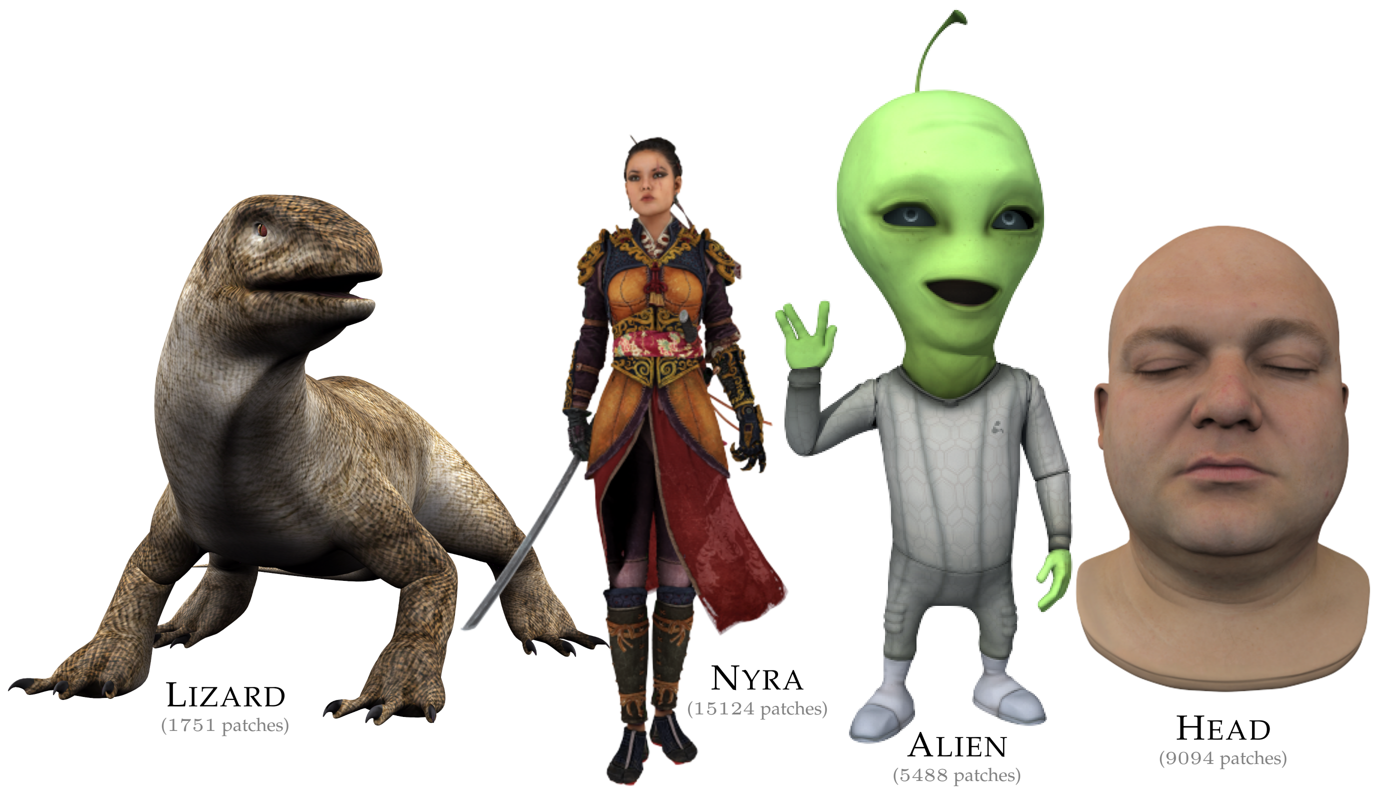


Fig. 1: *Models rendered using a GPU software implementation of patch textures. Texture filtering can be applied respecting topology, bypassing limitations of standard 2D texture mapping. With minimal hardware changes, patch textures could be implemented with comparable memory and performance to standard 2D textures, but with better quality and usability.*

cency maps, which would involve more substantial changes to existing GPU hardware. We also discuss potential approaches for minimizing data duplication and provide a detailed discussion on *corner-sampled images*, which can be considered a minimal implementation of patch textures, available on NVIDIA's Turing architecture. Finally, we discuss the difficulties inherent to providing similar hardware support for Ptex [1].

2 BACKGROUND

The problems of texture mapping are infamous in the computer graphics community. Defining a desirable mapping is time-consuming and often involves manual effort in practice. The filtering inconsistencies caused by seams reveal their locations on the rendered images and cause cracks on surfaces with displacement mapping. Changes to the model topology or geometry can have global effects and require completely regenerating the mapping and the corresponding textures. Therefore, texture mapping operations not only take a substantial amount of artist time, which dominates the cost of AAA video game production, but also limit the use of advanced GPU features, such as tessellation [6].

Researchers have developed various methods that either improve the mapping process or provide an alternative to texture mapping [7]. Methods that try to hide the seams [8], [9], [10] complicate the texture-authoring process even further and do not provide a solution to anisotropic filtering. Sparse volumetric representations [11], [12], [13], [14] help the texture-authoring process, but they introduce restrictions on the model geometry, suffer from additional performance cost, and cannot handle anisotropic filtering. Volume-based parameterizations [15], [16] can improve the process of defining a mapping, but do not provide solutions for other problems of texture mapping.

Mesh colors [2] and ptex [1] provide alternative representations for defining textures by relying upon the model topology to define an implicit mapping from the model space to the texture data. Thus, they significantly improve the texture-authoring process by eliminating the issues caused by having an explicit mapping. Operations like model editing after defining the texture data and local resolution readjustment are trivially supported by these methods. They also solve the problem of filtering inconsistencies by either directly filtering across edges during rendering (as in ptex) or storing texture data directly along edges (as in mesh colors). Therefore, it is no surprise that these methods have increasing popularity for offline rendering. Mesh colors and ptex are closely related, since they can be considered as topological duals of each other in-terms of the locations of texture samples implicitly placed upon the model surface. However, the minor theoretical difference between them leads to important practical distinctions. Ptex must store the model topology information for filtering across edges. As a result, hiding the seams along edges can be challenging if the faces on either side of an edge have different resolutions. Mesh colors avoid these problems by storing colors directly along the edges (i.e. *edge colors*) and at the vertices (i.e. *vertex colors*), in addition to storing colors within the faces (i.e. *face colors*). The common drawback of mesh colors and ptex is that neither can currently take full advantage of the available texture-filtering hardware on the GPU. Therefore, texture filtering must be implemented in software, which can be up to an order of magnitude slower than hardware-accelerated texture filtering.

Recently, mesh color textures [3] were introduced for partially using the existing texture-filtering hardware to handle bilinear filtering operations of mesh colors. This is achieved by converting the mesh color data into a 2D texture. Unfortunately, this conversion is not lossless, especially when the texture values are clamped (e.g. between 0 and 255 with 8-bit color channels), and it involves solving an optimization problem for generating higher mipmap levels. Moreover, because of the non-power-of-two resolution progression of the mipmap chain, the individual mipmap levels must be stored as separate textures. Consequently, more texture objects are used, which decreases locality and requires more switching in the shader or API. Texture coordinates on different mipmap levels are computed from a compact 4D coordinate representation and trilinear filtering, now crossing between textures, must be implemented in software-emulation paths, both of which substantially increase the shader complexity. Anisotropic filtering is essentially impossible to emulate without seam artifacts: the GPU-computed sample locations can be completely nonsensical, since they are computed under the presumption that the texture is an ordinary 2D texture. Though the same problems with anisotropic filtering also exist with standard 2D textures along seams, they appear along every edge with mesh color textures. Nonetheless, mesh color textures offer the texture-authoring advantages of mesh colors with minimal performance overhead for real-time rendering, as-compared to standard 2D textures. The remaining problems regarding shader complexity, hardware-accelerated trilinear filtering, and anisotropic filtering require changes to the GPU hardware. The patch texture representation we introduce is designed to address these remaining problems.

3 Patch Textures

Our *patch texture* representation is trivial to generate from mesh colors. Our representation stores all texture data associated with each face separately. This simplifies the implementation of hardware texture-filtering operations and minimizes the changes to current GPU hardware needed to support patch textures. Thus, edge colors are stored twice (i.e. one for each face that meets at an edge) and all vertex colors are stored as many times as the number of faces using them. This data duplication has a relatively small impact, since the primary memory consumption for high-resolution mesh colors is due to the face colors, which are not duplicated.

It is typical to define mesh colors on a relatively low-resolution mesh, referred to as the *canvas mesh*. Arbitrary tessellations of this canvas mesh, such as ones generated via typical subdivision operations, can directly use the mesh color data of the canvas mesh. The GPU rendering pipeline facilitates this approach through the use of tessellation shaders. Thus, following the terminology of tessellation shaders, we refer to the canvas mesh faces as *patches*.

As with mesh colors, patch textures require that the model consist of only quadrilaterals and triangles. We handle these two types of primitives using two different texture types: *quad patch textures* and *triangle patch textures*, which involve different storage methods and filtering operations. A given set of patch textures is associated with a particular model and its topology, so it can only be used by this model or its arbitrary tessellations (whether precomputed or generated on the GPU at render time via tessellation shaders).

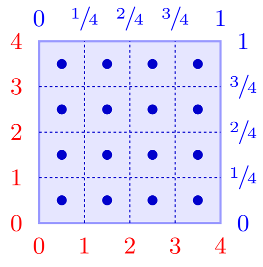
Mesh colors allow specifying the resolution of each patch independently. Yet, it is important for filtering consistency to match texture sample locations along an edge used by two patches, even when they have different resolutions. A simple way to achieve this for color data is to require that all patch resolutions be powers of two. Then, the texels of a lower-resolution patch texture fall exactly onto texel positions of the higher-resolution patch texture. The texture samples shared by the two patches can be specified independently, and the additional texture samples used by the higher-resolution patch are set as linear interpolations of the shared ones. This way, both patches agree on the texture values along the shared edge. Therefore, we assume this power-of-two restriction in our discussion of patch textures, though arbitrary resolutions can be easily supported, similarly to standard 2D textures.

Our patch texture representation is designed to be hardware-friendly. Nevertheless, we expect the exact hardware implementation of patch textures would be vendor-specific and may vary in different generations of future hardware. Therefore, in the following subsections, we discuss different alternatives for storage options and filtering operations.

3.1 Quad Patch Texture Storage

A standard 2D texture uses an array of discrete texel values to represent a continuous function on a space defined using normalized coordinates $\langle s, t\rangle$ such that $s, t \in [0, 1)$. These are multiplied by a width w and height h, the texture image resolution, to produce coordinates $\langle u, v\rangle$, where $u \in [0, w)$ and $v \in [0, h)$. In uv-space, texels are placed at half-integer coordinate positions. That is, texel $\langle i, j\rangle$ within the array is at position $\langle i + 0.5, j + 0.5\rangle$ in uv-space, as illustrated in Figure 2a. Bilinear texture filtering uses the four nearest texels to a sample position $\langle u, v\rangle$ (see Section 3.4). When u is within half a texel of 0 or w, or v is within half a texel of 0 or h, this requires accessing texel locations that are outside the $w \times h$ array of texels. Graphics APIs handle this by defining *wrap modes* that extend the texel array, e.g. by using a constant texel value outside the array, extending the edge texel values, or by replicating or mirroring the texel array. None of these are satisfactory for mapping textures onto an arbitrary model.

Patch textures solve this problem by placing texels at integer positions on the $\langle u, v\rangle$ coordinate grid. This is illustrated in Figure 2b, which shows a patch texture with $w = 4$ and $h = 4$ in uv-space that is specified using $(w+1) \times (h+1)$ texels. As a result, sampling within the texture st-space


(a) Standard 2D Texture

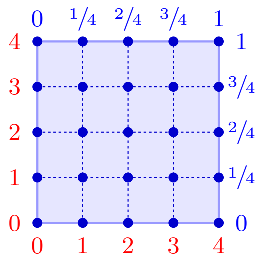
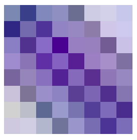
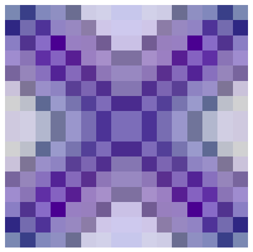

(b) Quad Patch Texture

Fig. 2: *Placement of texels in $\langle u, v\rangle$ and $\langle s, t\rangle$ coordinates for a 4×4 texture with **(a)** standard 2D textures and **(b)** patch textures.*


(a) Texels


(b) Standard 2D Texture

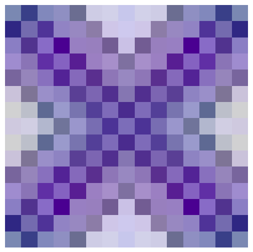

(c) Quad Patch Texture

Fig. 3: *Reflection wrap mode applied to **(a)** a set of texels, showing that **(b)** texels along the borders appear on both sides of the reflection lines with standard 2D textures, and **(c)** patch textures avoid this problem, such that the border texels appear once.*

never requires using texel values that are outside of the grid.

Wrap modes are defined for standard textures to allow accesses outside (and near the edges of) the texture *st*-space. To maintain orthogonality with existing texture features, wrap modes for patch textures may be defined to allow accesses outside of *st*-space. Figure 3 compares mirroring for a standard 2D texture and a quad patch texture. Notice that quad patch textures solve a known texel repetition problem of the mirror wrap mode with standard 2D textures.

Quad patch textures are stored in the same way as standard 2D textures. Since we restrict texture resolutions to powers of two, a quad patch texture may have resolution $w = 2^i$ and $h = 2^j$, where i and j are non-negative integers, requiring a storage of $(2^i + 1) \times (2^j + 1)$ texels. We discuss how this impacts memory layout in §3.3. All modern GPUs fully support textures with non-power-of-two sizes, so all texture formats and compression modes used with standard textures may be used with patch textures. Render-to-texture would render the overlapping edge texels in both of the patch textures that meet at an edge.

3.2 Triangle Patch Texture Storage

In a simplistic implementation, triangle patch textures can be stored in a rectangular array with roughly half the texture area unused. Quad-dominant meshes are common (especially when tessellation is being used), so the overall storage overhead can be negligible in practice.

Alternatively, any triangular patch texture can be sliced, and the top piece rotated around to fit into the wasted space, as shown in Figure 4. Consider a triangular mesh texture

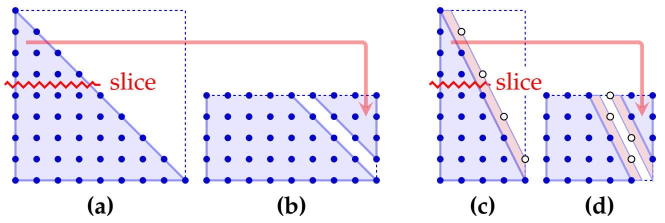


Fig. 4: *Alternative storage options for triangles: **(a)** any triangular patch texture can be stored in a quadrilateral array with roughly half of the texels wasted, or **(b)** it can be sliced and repacked to avoid wasted storage. This is possible even when the triangular patch texture has unequal resolutions along its sides **(c,d)**. The open circles represent texels that are outside the triangle but that must be defined for barycentric interpolation (see §3.4).*

with both major edges 8 units long (9 texels). When packed simplistically, the data occupies the same space as a 9×9 standard texture (Figure 4a). However, the top four rows of this triangle can instead be cut off and rotated to be stored beside the bottom five rows (Figure 4b). The result is that the triangular patch texture can be stored in a 9×5 rectangle with no wasted space.

Triangular mesh textures need not have the same size for the two major edges. Consider a triangular patch texture with a horizontal edge length of only 4 units (5 texels), as in Figure 4c. In this case, a 5×9 texel array compacts to a 6×5 array (Figure 4d). Note that the cut must be made through the longer of the two main edges so that the extra texels outside the triangle do not overlap when the upper and lower halves are packed.

Accessing texels in this compacted format requires a simple remapping of the $\langle u, v\rangle$ coordinates. Let W and H be the width and height of the triangle in texels, respectively, rotating the triangle if necessary so that $W \geq H$. Let S be the number of texels below the slice, with the restriction that $S \geq H/2$. If $v < S$, the texel is accessed normally. Otherwise, the $\langle u, v\rangle$ coordinate is remapped to $\langle W - u - 1, 2S - v - 1\rangle$.

A bilinear filtering operation that requires texels that are both above and below the slice may be accomplished in a manner similar to how filtering is performed across opposite edges of a texture that uses the edge wrapping mode. This is more efficient if S is a multiple of the memory allocation tile size for the GPU. E.g., if texels are stored in 4×4 blocks, S would be $H/2$ rounded up to a multiple of 4.

3.3 Mipmap Storage

A *mipmap chain* [17] stores successively smaller versions of the base texture map to allow filtering at varying resolutions. Given a standard 2D texture with width w_0 and height h_0, the resolution $w_\ell \times h_\ell$ of mipmap level ℓ is computed by a power-of-two reduction from the base level (i.e. $\ell = 0$), using

$$w_\ell = \max\left(\lfloor w_{\ell-1}/2\rfloor, 1\right) \qquad (1)$$
$$h_\ell = \max\left(\lfloor h_{\ell-1}/2\rfloor, 1\right) \qquad (2)$$
$$\ell \in \left\{1, 2, \cdots, \lceil \log_2(\max(w_0, h_0))\rceil\right\} \; .$$

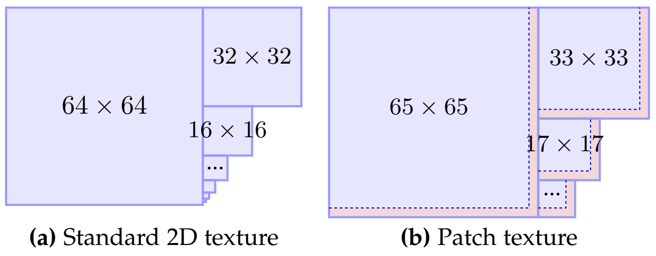


Fig. 5: *A pictorial comparison of mipchains for a* 64×64 *texture size. Padding is required to align data to GPU texture tiles.*

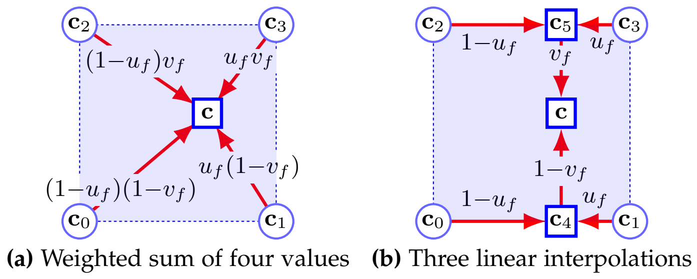


Fig. 6: *Bilinear filtering alternatives used in current GPUs.*

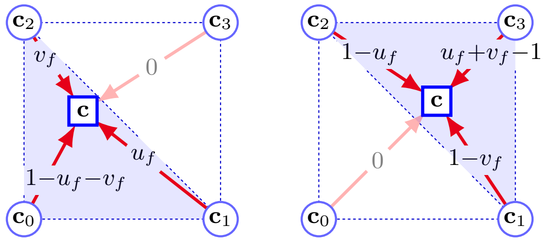


Fig. 7: *Barycentric filtering using weighted sum of texels.*

For a standard 2D texture, the size of the texel array at mipmap level ℓ is $w_\ell \times h_\ell$. Patch textures use the same mip sizes, but the texel array size is one larger than the width and height, so the size of a patch texel array at mipmap level ℓ is $(w_\ell + 1) \times (h_\ell + 1)$.

In-general, GPUs store texels in $n \times m$ tiles (where n and m are small powers of two) in-order to reduce memory bandwidth for 2D accesses. For example, if $n = m = 4$ and the texel size is 32 bits, then a single tile stores 64 bytes, which is a typical cacheline size. As a result, textures are aligned and padded to store a multiple of the tile width and height. For a standard 2D texture with a power-of-two resolution, each mipmap level except for the few smallest has a width and height that are multiples of the tile size.

When the size of the texel array does not evenly divide the GPU tile size, the affected data must be padded out until it does. This increases the texture storage requirements, with larger textures experiencing proportionally less wastage. Figure 5 illustrates parts of the mipmap chain for a 64×64 standard 2D texture and the corresponding patch texture, which pads out each mipmap level to a tile boundary in u and v. Note that the percentage of wasted space due to tiling decreases with increasing resolution.

The lowest-resolution patch textures are 1×1 in size but store 2×2 texels, each of which corresponds to the color at a patch vertex. With mesh colors, it is possible to generate additional mipmap levels using larger precomputed filter sizes that take adjacent geometry into account, in-order to further reduce aliasing. Thus, the mipmap chain of a patch texture can have multiple 2×2 texel mipmap levels at the end of the chain, each of which filters a larger, multi-patch, region of the model.

3.4 Bilinear and Barycentric Filtering

Quad patch textures use bilinear filtering similar to that of standard 2D textures. Given a sample position $\langle u, v\rangle$, assuming that $\langle u, v\rangle$ is at least half a texel away from the borders of the texture, bilinear filtering on standard 2D textures uses the 2×2 texel region with array indices $\langle i, j\rangle$, $\langle i+1, j\rangle$, $\langle i, j+1\rangle$, and $\langle i+1, j+1\rangle$, where the integer indices are defined as $i = \lfloor u - 0.5\rfloor$ and $j = \lfloor v - 0.5\rfloor$. The bilinear filter weights are determined using fractional coordinates $\langle u_f, v_f\rangle$, where $u_f = u - 0.5 - i$ and $v_f = v - 0.5 - j$.

For patch textures, the texels are on integral $\langle u, v\rangle$ positions, rather than half-integral positions as they are for standard textures. As a result, the four texels accessed are selected using integer indices defined as $i = \lfloor u\rfloor$ and $j = \lfloor v\rfloor$. The bilinear filter weights are determined using fractional coordinates $\langle u_f, v_f\rangle$, such that $u_f = u - i$ and $v_f = v - j$. Unlike standard textures, this works for any sample position within the borders of the texture.

Regardless of how the texels and weights are determined, bilinear filtering can be computed using either a weighted sum of four texel values (Figure 6a) or three linear interpolations (Figure 6b). The first method has fewer dependent operations and so allows more parallelism. The second method uses one fewer multiplier by rewriting the $u_f\mathbf{c}_1 + (1 - u_f)\mathbf{c}_0$ linear filtering equation as $\mathbf{c}_0 + u_f(\mathbf{c}_1 - \mathbf{c}_0)$.

Before the introduction of floating-point textures, GPUs typically used the second method to reduce the number of multiplies required. However, floating-point texel values would require normalizing for each linear interpolation stage, so the first method is typically used in modern GPUs. In effect, the first method extends a CPU-style fused mul-add instruction into a fused sum of four multiplications.

For triangular patches, the mesh colors method uses barycentric filtering on the three closest texels. Nonetheless, bilinear filtering can be used to approximate a barycentric lookup. Indeed, mesh color textures [3] use bilinear filtering for triangles, which requires storing additional texels near diagonally placed triangle edges, and ensuring seamless filtering along such edges may require modifying the given mesh color values when the texel values are clamped. However, it can be used as a fallback option for backward compatibility in a patch textures implementation.

A better solution, at minimal hardware cost, is to use the bilinear filter logic to blend three texel values using the triangle's barycentric coordinates. We present three alternatives for using bilinear filtering logic to perform barycentric filtering. In each, the first step is to determine whether barycentric filtering uses the lower-left three texels of the 2×2 texel region or uses the upper-right three texels. This is determined from $\langle u_f, v_f\rangle$. If $u_f + v_f < 1$, we use the lower-left texels $\mathbf{c}_0$, $\mathbf{c}_1$, and $\mathbf{c}_2$. If $u_f + v_f > 1$, we use the upper-right texels $\mathbf{c}_1$, $\mathbf{c}_2$, $\mathbf{c}_3$. If $u_f + v_f = 1$, either set of texels may be used and the equations linearly interpolate $\mathbf{c}_1$

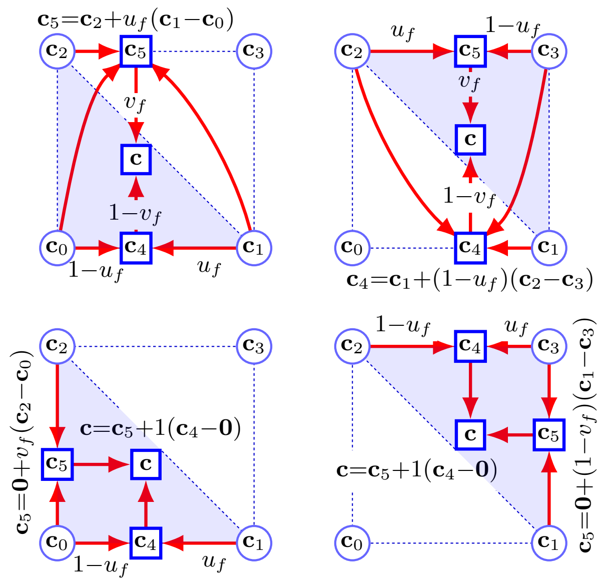

$\mathbf{c}_5{=}\mathbf{c}_2{+}u_f(\mathbf{c}_1{-}\mathbf{c}_0)$
$\mathbf{c}_2$
$\mathbf{c}_5$
$\mathbf{c}_3$
v_f
$\mathbf{c}$
$1{-}v_f$
$\mathbf{c}_0$
$\mathbf{c}_4$
$\mathbf{c}_1$
$1{-}u_f$
u_f
$\mathbf{c}_4{=}\mathbf{c}_1{+}(1{-}u_f)(\mathbf{c}_2{-}\mathbf{c}_3)$
$\mathbf{c}{=}\mathbf{c}_5{+}1(\mathbf{c}_4{-}\mathbf{0})$
$\mathbf{c}_5{=}\mathbf{0}{+}v_f(\mathbf{c}_2{-}\mathbf{c}_0)$
$\mathbf{c}_5{=}\mathbf{0}{+}(1{-}v_f)(\mathbf{c}_1{-}\mathbf{c}_3)$

Fig. 8: *Barycentric filtering using three linear interpolations using* ***(top)*** *a similar construction to bilinear filtering and* ***(bottom)*** *an alternative that eliminates the need for the third multiplication.*

and $\mathbf{c}_2$.

Figure 7 illustrates the weighted-sum technique for barycentric interpolation. Note that the weights are different from the quad patch case. The selection of weights depends on which triangle the sample position falls into. Figure 8 illustrates two ways to use three linear interpolations to perform barycentric interpolation. A few multiplexors are used to change the weights and texture values fed into the linear interpolation stages, but the multiply/add logic in each linear interpolator remains the same.

Note that the filtering methods in Figure 6 and Figure 7 retain the GPU filtering logic unchanged. Only the control logic and multiplexers used to select inputs to the filtering logic are changed. Therefore, supporting patch texture filtering involves inexpensive and relatively minor changes to existing GPUs.

Finally, trilinear filtering on 2D textures is implemented by linearly interpolating the results of filtering two faces in the mipmap chain, regardless of what kind of filtering is performed on those individual faces. Therefore, trilinear filtering requires no hardware changes in order to be used with patch textures.

3.5 Anisotropic Filtering

One of the advantages of our patch texture representation is that it makes it relatively easy to support anisotropic filtering. Anisotropic filtering is typically implemented by using a weighted sum of multiple trilinear filtering probes along a line in texture space from an appropriate pair of mipmap levels, as is suggested by the OpenGL specification [18]. This approach can be used for handling patch textures as well since each trilinear filter probe within the bounds of the patch accesses only texels specified within the patch. However, care must be taken if the sampled texture coordinates fall outside of the patch.

One approach is to simply not take a sample if its location is outside of the patch. In effect, this clips the part of the anisotropic filter kernel that falls outside of the patch. Note that even if the filter kernel is clipped by patch edges, the center of the kernel is guaranteed to be inside the patch. This clipping obviously changes the filter kernel shape. The net effect is simply to limit the level of anisotropy, such that the subset of the regular anisotropic filtering samples that fall outside of the patch boundaries are ignored, and the final result is computed using only the valid texture samples that are inside the patch boundaries. The part of the kernel that is clipped is responsible for approximating the texture values on the neighboring patch. However, the screen-space derivatives used for computing the kernel can be different for this other patch, so this clipping ensures that only the part of the kernel that is guaranteed to be properly computed is considered. Indeed, this is the solution implemented by mesh colors with software filtering [2].

If the application desires to include the clipped part of the filter kernel, this can be achieved by multi-sample anti-aliasing (MSAA) with centroid sampling. In this case, each patch that covers at least one of the multi-sample locations in a pixel is responsible for computing its part of the pixel footprint in texture space. The combined result forms the approximation of the filtered texture value.

The hardware implementation of anisotropic filter kernel clipping can be handled in two ways. The first detects and discards samples that are outside of the patch and assigns weights only to the samples that are inside the patch. This requires testing the sample locations prior to bilinear/barycentric filtering. The second method assigns weights as if all samples were valid, then steps through the samples, ignoring the ones that are outside of the patch. This must be followed by normalizing the result by the accumulated weight of all samples inside the patch.

Note that the problem of sampling invalid texture locations is not specific to our patch textures. Standard 2D texture mapping also suffers from similar issues along seams, with the additional problem of needing to bilinearly filter against samples outside of the patch. Indeed, this is arguably a more-serious problem along the seams of standard 2D textures, because the texture samples that fall outside of a patch can read arbitrary data from the texture, depending on how the mapping was defined. A similar solution that clips the filter kernel has been applied for the related problem of Ptex as well [19]. The separation of patch textures provides a convenient way to handle filter kernel clipping in hardware, since filters are always clipped only at patch edges.

An alternative solution to clipping is to use the hardware-supported edge-clamp mode to in-effect move samples outside of the patch to the closest position along the edges of the patch. This alternative would approximate the result of anisotropic filtering by using the closest-available texture data from the patch. Again, the result can deviate from the intended anisotropic filtering computation, but at least all samples are taken from a nearby valid location. One advantage of this approach is that it requires no hardware change and has no performance impact.

In Section 4 we provide yet another alternative that would allow sampling the texture of the neighboring patch, but it involves a more complex representation of mesh textures and additional expense.

3.6 Accessing Patch Textures in Shaders

Patch textures could be implemented as a collection of bindless textures. Bindless texture handles are typically 8 bytes. How this handle is used to access the actual texture is implementation-dependent. As for the standard 2D texture case, it may be sometimes possible to encode necessary meta-information directly into the handle alongside a pointer to the data, so that no additional indirection is required, but as in OpenGL [20], we leave such considerations to the implementer. Treating a mesh texture as a collection of bindless textures eliminates the need for any hardware modification or software API changes for accessing them.

From the software programmer's perspective, the main additional complexity of using a set of bindless textures, as-opposed to using a single standard 2D texture, is sending their handles to the shader. When using a small number of bindless textures, their handles can be specified as uniform shader variables. Under the typical use-case of patch textures, however, a large number of bindless textures may need to be created. These handles can be sent by using shader storage buffer objects, per-vertex attributes, or some new API.

4 Hardware-Managed Adjacency Map

The patch texture structure we described in the previous section is a simple way to implement mesh colors with minimal modification to existing GPU hardware. However, it does not provide a mechanism for accessing the texture data on neighboring patches.

4.1 Uses for Adjacency Data

Accessing the texture data of neighboring patches is often completely unnecessary. Since the mesh colors structure defines texels along patch edges, all filtering operations (including anisotropic filtering, up to the within-patch approximations suggested in §3.5) can be performed without accessing data from neighboring patches. Nonetheless, such accesses are needed for complex algorithms that "walk" over the model surface by sampling different locations.

For example, parallax occlusion mapping [5] is a commonly used technique that may need to access the texture data of multiple patches. Parallax occlusion mapping begins sampling the texture at the texture coordinates of the pixel. Then, depending upon the texture data, it iteratively modifies the texture coordinate and samples the texture again at the modified texture coordinate. These iterative modifications of the texture coordinate can move the texture sample location outside of the patch boundary. Unfortunately, clamping the texture coordinate to keep the sample location within the bounds of the patch leads to visual artifacts with this technique, as shown in Figure 9.

Note that a similar problem also exists with standard 2D textures when the algorithm crosses a seam boundary. Indeed, visual artifacts near seams can be more severe with texture mapping, as shown in Figure 10. In practice, however, this is remedied by strategically placing the seams during mapping, such that the unavoidable artifacts caused by crossing seams can be hidden from the camera view. In the case of patch textures, this problem occurs at every patch boundary. While clamping the texture coordinate may alleviate the visual artifacts for some textures, completely eliminating them requires accessing the texture data of neighboring patches.

4.2 Existing Software Solutions

With patch textures, sampling data from a neighboring quad or triangle requires providing its bindless texture handle. Therefore, an algorithm that samples multiple locations on the surface not only needs to check when the texture coordinates are outside of the quad or triangle, but also needs to obtain the bindless texture handle of the neighboring patch and modify the texture coordinates appropriately, which requires access to the adjoining patch's relative orientation. These requirements significantly complicate the implementation of such algorithms.

4.3 Hardware Support for Adjacency Data

Our patch textures method can be extended to support adjacency information. This could greatly simplify algorithms that require access to the texture data on neighboring patches. Note that bilinear and barycentric filtering operations do not require accessing neighboring patch textures, while anisotropic filtering can be handled without it, as explained above in §3.5.

The basic idea is to store an *adjacency map*, similarly to Ptex [1]. For each edge of a patch, the adjacency map contains a reference to the neighboring patch and its orientation. Texture filtering hardware must detect when an edge is crossed. This can be easily accomplished by checking whether the coordinates are outside the patch (i.e. not within $[0, 1]$). Existing GPUs already support this for handling bilinear filtering near the edges of cubemaps that are stored as six separate faces. However, crossing a patch edge also requires accessing the adjacency information. This can be handled by storing the adjacency data as a part of the texture descriptor. This requires additional logic to fetch the adjacency data whenever the coordinates are outside the patch.

While such operations can be easily done in software, hardware support would not only make the software implementation simpler but would provide opportunities for additional optimizations. Hardware could prefetch adjacent patch handles so that when the shader requests a sample from texture coordinates that are outside the patch, the hardware can determine which edge is crossed, modify the texture coordinate using the neighbor patch's orientation, and sample the texture of the neighboring patch. This process could potentially proceed recursively up to some maximum limit, although at increased access latency.

The texture definition data referenced by the bindless texture handle is cached on typical GPUs. Accessing adjacent patches should often hit in this cache, thus avoiding extra latency. When the adjacent patch misses in the cache, it is likely to be accessed directly later. As a result, there should be relatively low added overhead looking up texture definition data for adjacency operations.

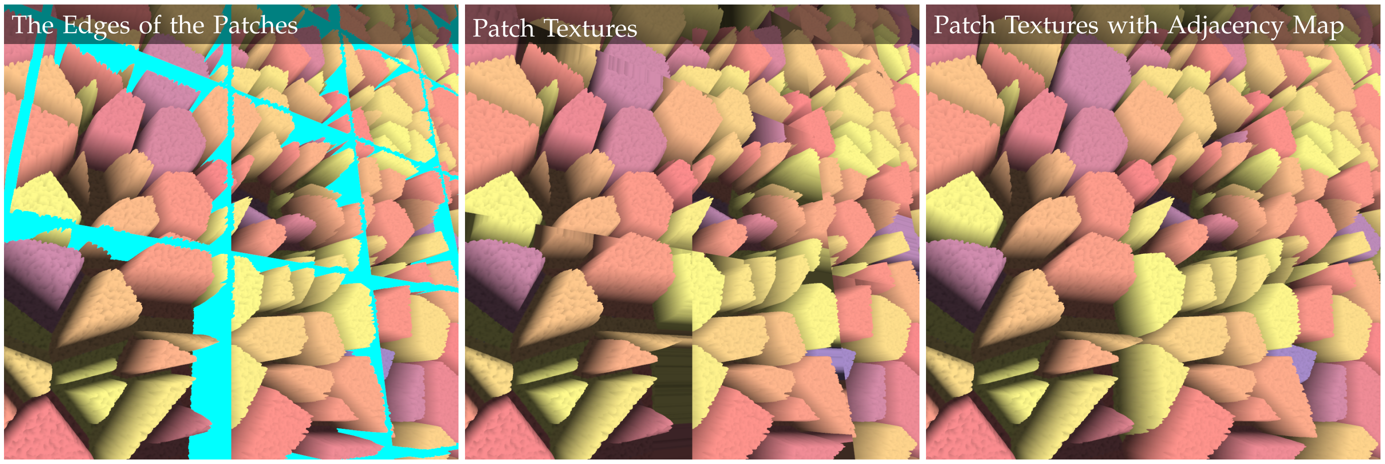


Fig. 9: *Parallax occlusion mapping [5] with patch textures. Notice that clamping the texture coordinates leads to inconsistent results near patch edges. Accessing the texture data of the neighboring patches using an adjacency map is required for achieving consistent results without visual artifacts.*

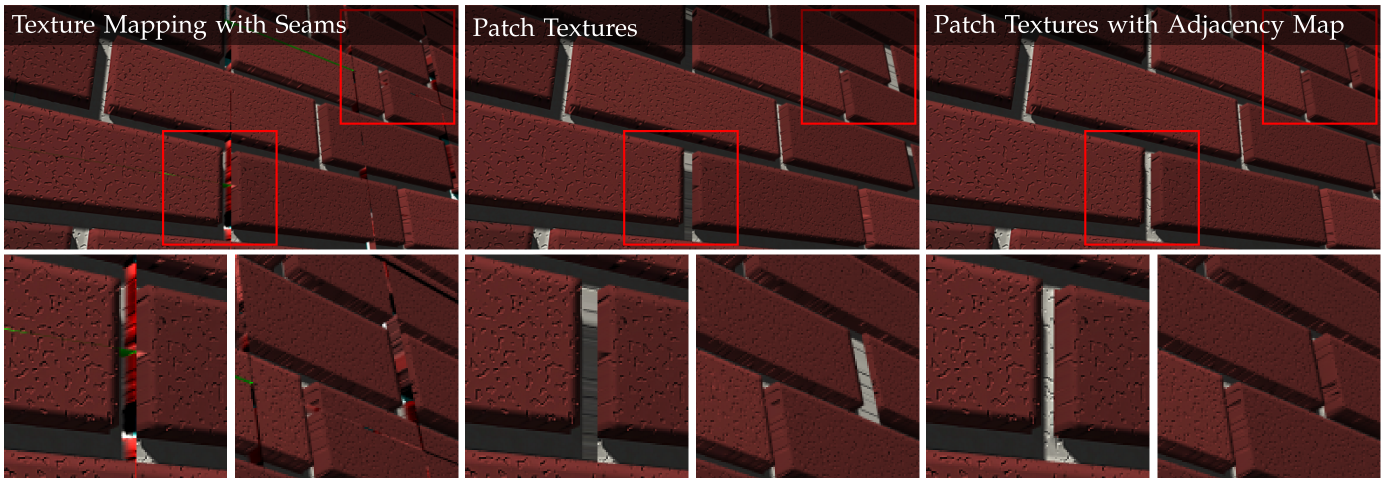


Fig. 10: *Parallax occlusion mapping [5] with texture mapping and patch textures.* ***Left:*** *Texture mapping leads to cracks on the surface near seams.* ***Middle:*** *Clamping the texture coordinates with patch textures alleviates, but does not eliminate, visual artifacts.* ***Right:*** *Adding an adjacency map that allows cross-patch walking for sampling out-of-patch data fixes all problems.*

Note that there is no need to explicitly store adjacency information around vertices between patches. All of the patches that meet at a vertex can be accessed by sequentially stepping across edges around the vertex. Also note that indirection occurs only at patch boundaries, not at every primitive boundary, so that if patches are relatively large, then edge-crossing, and especially multi-step edge-crossing, should be relatively rare.

4.4 Software Simulation

Our GPU software implementation of an adjacency map incurs the cost of an additional indirection every time an edge is crossed. For limiting this additional cost, the implementation can provide an upper bound on how many edges can be crossed (such as 1) for each texture-sampling operation, with coordinates handled differently once this upper bound is reached (e.g. by clamping). A hardware implementation could optimize this process by hiding the corresponding latency better.

We note that if there is hardware support for the adjacency map, it can also be used for improving the quality of anisotropic filtering by allowing anisotropic filtering probes to cross an edge, when needed.

Obviously, providing hardware support for adjacency maps would involve a more substantial modification of the current GPU hardware, adding additional logic for both accessing the adjacency map and for effectively prefetching adjacent patches' handles. Even with hardware support, using an adjacency map would still lead to some overhead. However, hardware support for adjacency maps is only needed for algorithms that modify the texture coordinate to be sampled. Again, typical texture filtering operations we explain in the previous section do not require an adjacency map.

5 Experiments and Evaluation

We evaluate the additional memory footprint of our patch texture representation using four example models shown

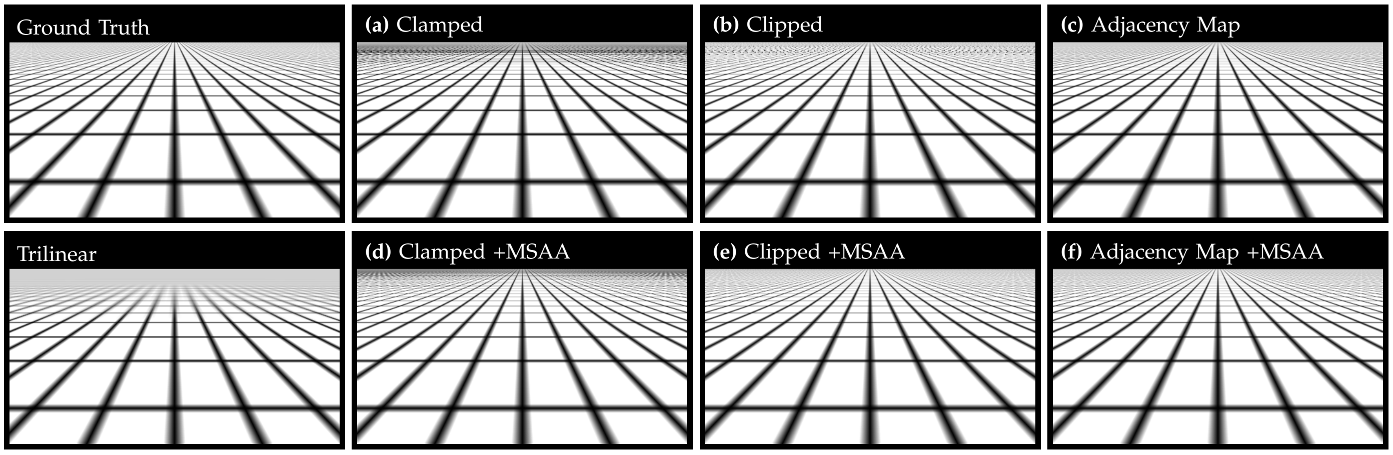


Fig. 11: *Contrived example showing different anisotropic filtering alternatives. Each cell in the grid is a single patch, meaning there are both many minified patches, and they are correlated to changes in the texture: a worst-case. Clipping and adjacency maps perform well, although only adjacency maps give the correct answer. See also Fig. 13 for a more-realistic scenario.*

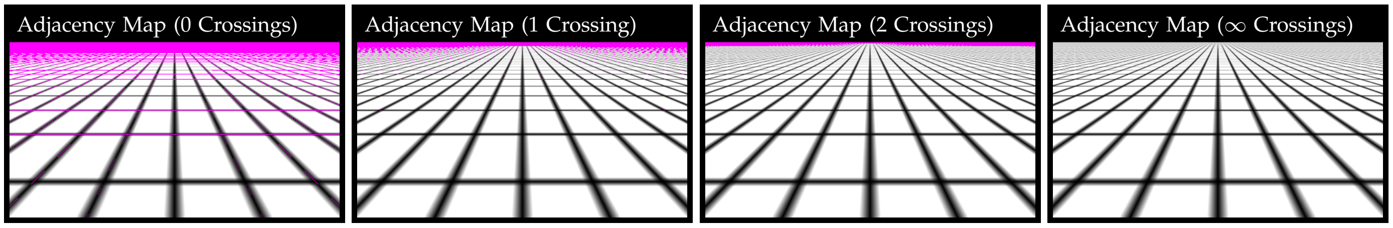

Adjacency Map (0 Crossings)
Adjacency Map (1 Crossing)
Adjacency Map (2 Crossings)
Adjacency Map (∞ Crossings)

Fig. 12: *Visualization of added operations due to edge crossing. With only one edge crossing, almost all patch edges are handled correctly. Only corners and cases with many tiny, minified patches are handled incorrectly. All but the most extreme cases are handled by two edge crossings.*

TABLE 1: *Texel footprint with mipmap levels.*

Model	Mesh Colors	Patch Textures					
		1×1 Tiles		2×2 Tiles		4×4 Tiles	
	Texels	Texels	(%)	Texels	(%)	Texels	(%)
LIZARD	8.46 M	8.68 M	(103%)	8.91 M	(105%)	9.37 M	(111%)
NYRA	31.13 M	32.27 M	(104%)	33.44 M	(107%)	35.88 M	(115%)
ALIEN	11.64 M	11.88 M	(102%)	12.12 M	(104%)	12.60 M	(108%)
HEAD	34.08 M	35.23 M	(103%)	36.40 M	(107%)	38.79 M	(114%)

in Figure 1, rendered with our GPU software emulation of patch textures. Note that these models include both quadrilateral and triangular patches. The results are presented in Table 1 in comparison to the size of the original mesh color data. Using 1×1 tiles, the only additional storage cost comes from duplicated edge and vertex colors, which is between 2% and 4% for these examples. This is mainly due to the fact that these are relatively low-resolution models with high-resolution mesh colors. Using 2×2 or 4×4 tiles increases this overhead up to 8% or 15%, respectively, for these models. Considering that standard 2D textures also incur a similar or even more storage overhead due to packing and padding around seams, we consider this extra storage cost acceptable. Moreover, models optimized for mesh colors (with lower-resolution canvas meshes) can further reduce the overhead.

We also simulate anisotropic filtering modes with our implementation in Figures 11, 12, and 13. Against a ground truth computed with supersampling, we show the effect of two different ways to handle filtering patch edges: clamping sample coordinates to remain within the center sample's patch and clipping the filter to avoid sample values outside the patch. Multisampling can be added on top of each strategy, as-well. The main beneficiary of multisampling is filter clipping, since the sub-pixel samples "fill in" the clipped filter value along border pixels [2], [19].

In Figure 11, we demonstrate something of a worst-case: there are many patches that are highly minified, and the texture data is correlated to the patch boundaries. We see that the adjacency map produces the best result, since the filter kernel is not modified. Whereas, clamping produces bad results due to the texture changing values near patch edges, and clipping requires multisampling to properly filter some regions. In Figure 12, we show a visualization of the added computation due to edge crossing operations. Without any crossings, we can see that most patch boundaries are computed incorrectly. With one edge crossing, most patch boundaries become accurate, although corners and cases where many patches are minified together are still incorrect. With an unbounded number of crossings, the correct result is obtained.

It is worth stressing that these differences are largely visually indistinguishable even in contrived examples such as this. In Figure 13, we show a more-typical model with a reasonable texture. The patches are still unrepresentatively small, as this model was converted from a model with standard texturing, but even so, any filtering variant is acceptable—and indeed, the results are largely indistinguishable.

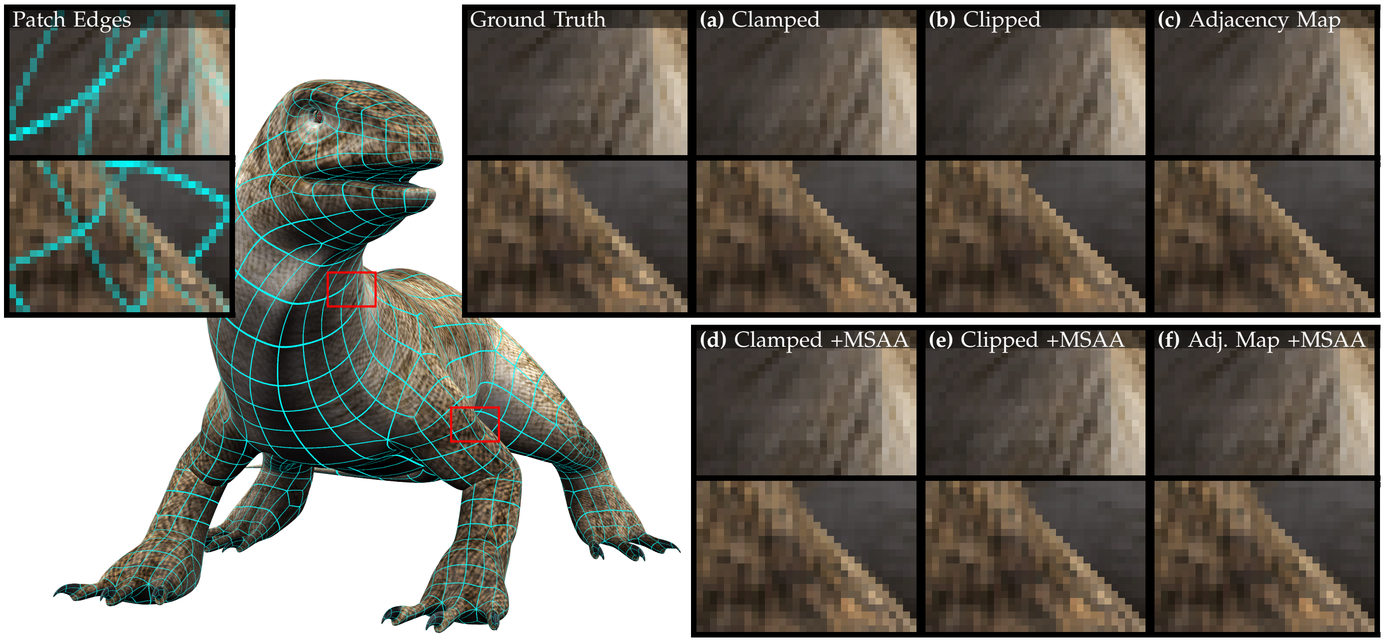


Fig. 13: *Anisotropic filtering alternatives for the* LIZARD *model. Out-of-patch samples can be* ***(a)*** *clamped to patch edges, or* ***(b)*** *the filter itself can be clipped. The most-correct solution is to* ***(c)*** *retrieve samples from the neighboring patch using an adjacency map, although this significantly increases complexity.* ***(d–f)*** *Multisampling improves filtering results.*

6 Discussion

Patch textures are not intended to replace all applications of standard texture mapping. In some cases, especially when using relatively simple models, a texture mapping can be defined easily, and seams can be completely hidden by placing them at locations that are not visible to the camera. Also, when a small texture is tiled over a coarse model's surface, texture mapping may be advantageous over patch textures.

For texturing complex models, however, patch textures facilitate the substantial improvements in the asset production pipeline provided by mesh colors. At render time, they provide storage efficiency by eliminating the wasted space in UV mapping. They also completely avoid seam artifacts and, as-such, they are particularly suitable to hardware tessellation and displacement mapping, avoiding the cracks on surfaces due to seams when using standard texture mapping. (Displacement mapping along edges between two patches with different patch texture resolutions can still lead to cracks due to floating-point precision issues; this can be avoided by carefully picking the edge tessellation resolutions.)

6.1 Data Duplication

Our current representation accepts the duplication of mesh colors data at patch corners and edges for the purpose of simplicity. More-complex schemes that reduce this overhead could be employed but implementing them might require more-complicated hardware alterations. Additionally, since most of the data is expected to be taken up by patch-interior face data, which is not duplicated, there is little to be gained (that is, the vertex data per-patch scales as $O(1)$, the edge data by $O(n)$, and the face data by $O(n^2)$, where n corresponds to the resolution of one side of a patch).

If the overhead attendant to a single patch is proportionally large, this means that the interior data is not very detailed, or completely absent. We suggest using traditional vertex colors to handle such cases. But, since such models are by-assumption very coarse in texture detail, the overhead of using patch textures for these cases as well may be acceptable, for being small in an absolute sense.

One approach to reducing the overhead of data duplication would be to ensure that the shared edge texels are only stored in one or the other patch texture. In this case, the texture data of the edge can be accessed using an adjacency map when needed. Whether this is ultimately practical depends on the details of specific GPU designs. However, a potential hardware implementation could make use of several existing GPU capabilities: storing the edge descriptors in the primitive data allows loading vertex/edge texels at the same time as the primary patch texture, and GPU filtering hardware already supports filtering discontiguous texels when e.g. wrap mode is enabled and filtering crosses an edge.

6.2 Corner-Sampled Textures

The recent NVIDIA Turing GPU architecture provides support for corner-sampled textures in the Vulkan API (`VK_NV_corner_sampled_image`). This new texture format implements many features of our patch textures. Therefore, starting with an implementation of corner-sampled textures, providing hardware support for patch textures would be even simpler.

Corner-sampled textures store the texture samples exactly at the same locations as patch textures. The only difference between a quad patch texture and a corner-sampled texture is the definition of the texture width and height. A quad patch texture with width w and height h is equivalent to a corner sampled texture with width $w + 1$ and height $h + 1$.

Corner-sampled textures provide mipmapping support, similar to our patch textures, provided that the "round-up" mode is selected for computing the mip-level sizes[1]. The mipmap levels of corner-sampled textures can be generated down to 2×2 resolution. Unlike our patch textures, corner-sampled textures do not support additional mipmap levels with 2×2 resolution. In theory, this limitation of corner-sampled textures can lead to reduction in texture filtering quality when higher mipmap levels are needed. However, such cases arise only when patches are smaller than pixels on the rendered image. In these cases, geometric aliasing can be a greater concern. Thus, we would expect that this limitation of corner-sampled textures would not likely be an important concern for most applications in practice.

A more-important limitation of corner-sampled textures, as compared to patch textures, is the fact that they do not support barycentric filtering. As we explain in §3.4, providing barycentric filtering would involve relatively minor changes to the existing bilinear filtering logic. Therefore, we would expect that future GPU hardware can easily provide support for barycentric filtering. Furthermore, it is possible to use bilinear filtering for triangles as well. This requires providing additional texture data near the diagonally placed edges of triangles, similar to the mesh color textures structure [3]. A minor problem with this solution is that some texture data may have to be modified slightly to ensure that texture filtering provides consistent results on both sides of all edges. Yet, we would expect that this minor modification would not be an important concern in practice. Also, in most cases it might be acceptable to keep the texture data as-is since the resulting filtering inconsistencies tend to be difficult to notice.

Another potential concern is the fact that storing corner-sampled texture data for triangles wastes about half of the texture storage. Nonetheless, this is a minor concern, given that quad-dominant meshes are commonly used during model authoring, as mentioned in §3.2.

In summary, we can conclude that corner-sampled textures provide a minimal implementation of patch textures. The minimal set of features supported by corner-sampled textures is sufficient to provide hardware filtering support for mesh colors in practical applications. It should be noted that, while it might seem that corner-sampled textures could easily provide support for Ptex, Ptex actually uses the sample locations of standard 2D textures (see the discussion below for more details), and so the sample locations are fundamentally incompatible.

6.3 Hardware Support for Ptex

Some of the concepts we describe in this paper could be applied for providing a similar hardware support to Ptex [1] as well. Note that the texture sample locations of Ptex directly correspond to the ones used by standard 2D textures. Indeed, Ptex can be implemented in packed 2D textures [21], including support for mipmapping [22] and without duplicating edge data [23]. As a result, bilinear filtering near edges requires accessing the texture data of the neighboring faces with Ptex. The original Ptex structure uses an adjacency map for this task. This process could be simplified using something similar to a quad patch texture by copying data from neighboring patches. The resulting per-face texture shown in Figure 14 is identical to a standard 2D texture, except that the $\langle s, t\rangle$ and $\langle u, v\rangle$ coordinates are defined differently. The additional texture samples outside of the face boundary are copied from the neighboring faces. This allows support for bilinear filtering using only a single texture's data without the need for an adjacency map.

However, ensuring consistent filtering on either side of an edge is nontrivial with Ptex when the two faces sharing the edge have different resolutions. It is not clear how the additional texture samples along the borders of per-face textures must be specified to ensure consistent bilinear filtering near edges shared by faces with different resolutions. Existing software implementations [22], [23] make an attempt to minimize the discrepancy of filtered texture values on either side of the face edges, but such discrepancies (i.e. seam artifacts) cannot be eliminated. This is because with per-face textures, the locations of texture samples on the model surface on either side of a seam do not overlap unless the neighboring faces have the same resolution. This is unlike mesh colors (and our patch textures), which eliminate this discrepancy completely by placing the texture samples differently.

Also, there is no good solution for selecting the correct texture data for the corners of a face texture near extraordinary vertices (vertices where more or fewer than four edges meet). Furthermore, Ptex uses a different representation for storing texture data on triangles, so our solution of

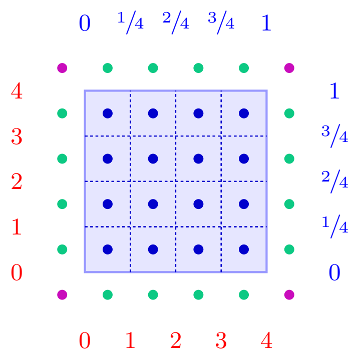


Fig. 14: *A possible texture format to provide hardware support for Ptex [1], using the same texture sample locations as standard 2D textures, but with additional texture samples along the borders (shown in green and purple). This modifies the $\langle u, v\rangle$ and $\langle s, t\rangle$ coordinates, as compared to 2D textures. The corresponding face covers the shaded area of the texture space. The texture sample data for the samples along edges (shown in green) should be copied from the neighboring faces. The corner samples (shown in purple) are copied from the second neighbors. Note that providing consistent texture sample data for the additional texture samples can be problematic when neighboring faces have inconsistent resolutions and near extraordinary vertices.*

1. The existing implementation allows using the standard "round-down" mode with corner-sampled textures as well, but this would not produce desirable mip-level sizes and would diverge from the mip-levels we define for our patch textures.

barycentric filtering cannot be applied directly.

Therefore, a practical solution to providing support for Ptex might be converting the Ptex data to mesh colors and then using our patch texture representation. This, however, would mean resampling the texture data, which is undesirable. Thus, generating the texture data using mesh colors, instead of Ptex, would be preferable.

6.4 Limitations

The main limitation of our approach is rendering models with many patches. When viewed from a distance, the patches will be close together in screen space, causing nonlocal access. Indeed, having many patches will produce many texture handles. In our implementation, we managed this with bindless textures.

A significantly larger number of texture handles is a clear disadvantage. Further work is required to quantify the performance impact of this. Anecdotally, while our implementation, which lacks many of the optimizations we discuss, had comparable performance to ordinary 2D texturing, our test setup is a software emulation, and so conclusions about performance are difficult to substantiate rigorously. In any case, we expect that performance would be affected by the amount of texture data, as well as how it is used within the shader.

Finally, although our test models were converted from regular 3D models, and therefore comprise many patches, this is unrealistically perverse: models designed for mesh colors in the first place can have far fewer patches.

7 Conclusion

We have introduced patch textures, a hardware-friendly representation of mesh colors. Patch textures allow mesh colors to be implemented with only minimal changes to existing GPUs, and thereby resolve the main difficulty of using mesh colors in interactive and real-time rendering applications.

Acknowledgments

We thank Christer Sveen for use of the alien character, Murat Afşar for the lizard model, Lee Perry-Smith for the head model, and Paul Tosca for the Nyra model. This project has been supported in part by a grant from Facebook Reality Labs.

References

[1] B. Burley and D. Lacewell, “Ptex: Per-face texture mapping for production rendering,” *CG Forum*, vol. 27, no. 4, pp. 1155–1164, 2008.

[2] C. Yuksel, J. Keyser, and D. H. House, “Mesh colors,” *ACM Transactions on Graphics*, vol. 29, no. 2, pp. 15:1–15:11, 2010.

[3] C. Yuksel, “Mesh color textures,” in *High-Performance Graphics (HPG 2017)*, 2017.

[4] A. Mallett, L. Seiler, and C. Yuksel, “Patch Textures: hardware implementation of Mesh Colors,” in *Proceedings of the Conference on High-Performance Graphics*, ser. HPG ’19, M. Steinberger and T. Foley, Eds. Goslar, Germany: The Eurographics Association, 7 2019, pp. 39–44. [Online]. Available: https://diglib.eg.org/handle/10.2312/hpg20191194

[5] Z. Brawley and N. Tatarchuk, “Parallax occlusion mapping: Self-shadowing, perspective-correct bump mapping using reverse height map tracing,” in *ShaderX^3: Advanced Rendering with DirectX and OpenGL*, W. F. Engel, Ed. Charles River Media, 2004, ch. 2.5, pp. 135–154.

[6] N. Tatarchuk, A. Lefohn, and P.-P. Sloan, “Frontiers in real time rendering,” in *ACM SIGGRAPH 2015 Courses*, 2015.

[7] C. Yuksel, S. Lefebvre, and M. Tarini, “Rethinking texture mapping,” *Computer Graphics Forum (Proceedings of Eurographics 2019)*, vol. 38, no. 2, 2019.

[8] N. Ray, V. Nivoliers, S. Lefebvre, and B. Levy, “Invisible Seams,” *Computer Graphics Forum*, 2010.

[9] S. Liu, Z. Ferguson, A. Jacobson, and Y. Gingold, “Seamless: Seam erasure and seam-aware decoupling of shape from mesh resolution,” *ACM Trans. Graph.*, vol. 36, no. 6, pp. 216:1–216:15, 2017.

[10] B. Purnomo, J. D. Cohen, and S. Kumar, “Seamless texture atlases,” in *Proc. of Symp. on Geometry Processing*, 2004, pp. 65–74.

[11] D. Benson and J. Davis, “Octree textures,” *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 785–790, 2002.

[12] P. H. Christensen and D. Batali, “An irradiance atlas for global illumination in complex production scenes,” in *Proc. of Rendering Techniques*, ser. EGSR’04, 2004, pp. 133–141.

[13] S. Lefebvre and C. Dachsbacher, “Tiletrees,” in *Proc. of the Symposium on Interactive 3D Graphics and Games*, 2007, pp. 25–31.

[14] S. Lefebvre and H. Hoppe, “Perfect spatial hashing,” in *ACM Trans on Graphics*, vol. 25, 2006, pp. 579–588.

[15] M. Tarini, K. Hormann, P. Cignoni, and C. Montani, “Polycube-maps,” *ACM Trans. Graph.*, vol. 23, pp. 853–860, 2004.

[16] M. Tarini, “Volume-encoded uv-maps,” *ACM Transactions on Graphics*, vol. 35, no. 4, pp. 107:1–107:13, 2016.

[17] L. Williams, “Pyramidal parametrics,” in *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, 1983, pp. 1–11.

[18] May 2018. [Online]. Available: https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_texture_filter_anisotropic.txt

[19] R. Toth, “Avoiding texture seams by discarding filter taps,” *Journal of Comp. Graphics Techniques (JCGT)*, vol. 2, no. 2, pp. 91–104, 2013.

[20] O. Wiki, “Bindless texture — opengl wiki,” 2019, [Online; accessed 5-November-2020]. [Online]. Available: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Bindless_Texture

[21] S. Kim, K. Hillesland, and J. Hensley, “A space-efficient and hardware-friendly implementation of ptex,” in *SIGGRAPH Asia 2011 Sketches*, 2011, pp. 1–2.

[22] J. McDonald Jr and B. Burley, “Per-face texture mapping for real-time rendering,” in *ACM SIGGRAPH 2011 Studio Talks*, 2011, pp. 1–1.

[23] J. McDonald Jr, “Eliminating texture waste: Borderless realtime ptex,” 2013, Game Developers Conference (GDC) Talk.

Agatha Mallett received BS degrees in Computer Science and Pure Mathematics from the University of New Mexico in 2014. She is currently working toward a PhD in Computer Science at the School of Computing, University of Utah. Her research interests include physically principled rendering theory and algorithms, spectra and color, high-performance computing, and hardware architectures for graphics.

Larry Seiler received an MS in Computer Science from Caltech in 1980 and a PhD in Computer Science from the Massachusetts Institute of Technology in 1985. After a career spent mostly developing graphics algorithms as part of GPU hardware teams, Larry is now a Research Scientist at Facebook Reality Labs. His research focuses on power efficient rendering and display algorithms for AR and VR devices.

Cem Yuksel is an associate professor in the School of Computing at the University of Utah. Previously, he was a postdoctoral fellow at Cornell University, after receiving his PhD in Computer Science from Texas A&M University in 2010. His research interests are in computer graphics and related fields, including physically-based simulations, realistic image synthesis, rendering techniques, global illumination, sampling, GPU algorithms, graphics hardware, modeling complex geometries, knitted structures, and hair modeling, animation, and rendering.