High-Performance Graphics (2019)
T. Foley and M. Steinberger (Editors)

Patch Textures: Hardware Implementation of Mesh Colors

I. Mallett[1] and L. Seiler[2] and C. Yuksel[1]

[1]University of Utah
[2]Facebook Reality Labs

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*. We discuss the hardware modifications needed for storing and filtering patch textures.*

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* [BL08] and *mesh colors* [YKH10]. 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* [Yuk17] were introduced for utilizing the existing GPU hardware for partially handling the filtering operations of mesh colors. This is achieved by converting mesh color data into a standard 2D texture. Yet, 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 can 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 can 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.

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 like tessellation [TLS15].

Researchers have developed various methods that either improve the mapping process or provide an alternative to texture mapping [YLT19]. Methods that try to hide the seams [RNLL10, LFJG17, PCK04] complicate the texture-authoring process even further and do not provide a solution to anisotropic filtering. Sparse volumetric representations [BD02, CB04, LD07, LH06] 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 [THCM04, Tar16] can improve the process of defining a mapping, but do not provide solutions for other problems of texture mapping.

Mesh colors [YKH10] and ptex [BL08] provide alternative representations for defining textures by relying on 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 interms of the locations of texture samples implicitly placed on 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 on faces (i.e. *face colors*). The common drawback of mesh colors and ptex is that they cannot take 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 [Yuk17] 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 textures 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

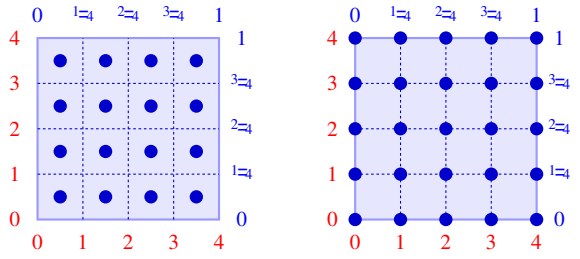
Our *patch texture* representation is trivial to generate from mesh colors. Our representation stores all texture data associated with each face separately. This simplifies the implementation of hardware texture-filtering operations and minimizes the changes to current GPU hardware needed to support 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 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, similar 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.



(a) Standard 2D Texture **(b)** Quad Patch Texture

Figure 1: *Placement of texels in $(u;v)$ and $(s;t)$ coordinates for a 4 4 texture with* ***(a)*** *standard 2D textures and* ***(b)*** *patch textures.*

3.1. Quad Patch Texture Storage

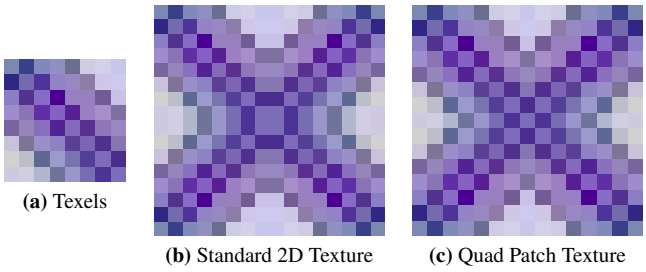
A standard 2D texture uses an array of discrete texel values to represent a continuous function on a space defined using normalized coordinates $hs;t i$ such that $s;t\ 2\ [0;1)$. These are multiplied by a width w and height h, the texture image resolution, to produce coordinates $hu;v i$, where $u\ 2\ [0;w)$ and $v\ 2\ [0;h)$. In uv-space, texels are placed at half-integer coordinate positions. That is, texel $hi;j i$ within the array is at position $hi + 0{:}5;\ j + 0{:}5i$ in uv-space, as illustrated in Figure 1a. Bilinear texture filtering uses the four nearest texels to a sample position $hu;v i$ (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 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 $hu;v i$ coordinate grid. This is illustrated in Figure 1b, which shows a patch texture with $w = 4$ and $h = 4$ in uv-space that is specified using $(w + 1)$ $(h + 1)$ texels. As a result, sampling within the texture st-space never requires using texel values that are outside the grid. Wrap modes for patch textures may be defined to allow accesses *outside* st-space, but this is not required for any area within the patch texture. Figure 2 compares mirroring for a standard 2D texture and a quad patch texture. Notice that quad patch textures solve the texel repetition problem of standard 2D textures with the mirror wrap mode.

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 $(2^i + 1)$ $(2^j + 1)$ texels. 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 with tessellation), so the overall storage overhead can be negligible in practice.



(a) Texels **(b)** Standard 2D Texture **(c)** Quad Patch Texture

Figure 2: *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.*

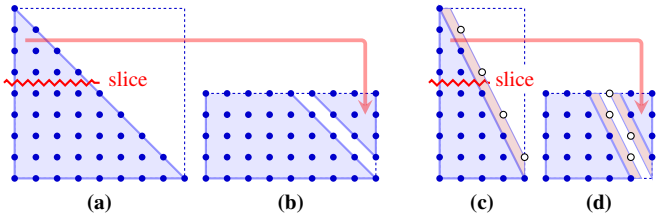


Figure 3: *Alternative storage options for triangles:* ***(a)*** *any triangular patch texture can be stored in a quadrilateral array with roughly half of the texels wasted, or* ***(b)*** *it can be sliced and repacked to avoid wasted storage. This is possible even when the triangular patch texture has 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.*

Alternately, any triangular patch texture can be sliced and the top piece rotated around to fit into the wasted space, as shown in Figure 3. Consider a triangular mesh texture 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 3a). However, the top four rows of this triangle can instead be cut off and rotated to be stored beside the bottom five rows (Figure 3b). The result is that the triangular patch texture can be stored in a 9 5 rectangle with no wasted space. Filtering across the slice may be accomplished in a manner similar to how filtering is performed across opposite edges of a wrapped texture. 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 3c. In this case, a 5 9 texel array compacts to a 6 5 array (Figure 3d). 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.

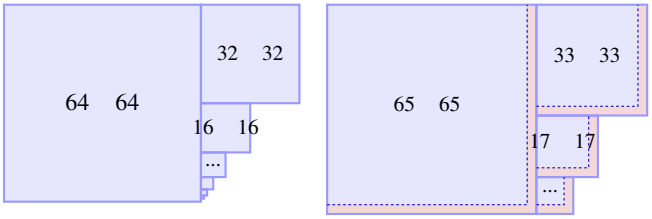
3.3. Mipmap Storage

A *mipmap chain* 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 $w`$ $h`$ of mipmap level $`$ is computed by a power-of-two reduction from the base level (i.e. $` = 0$), using

$$w` = \max(bw`_{-1}=2c;\ 1) \quad (1)$$

$$h` = \max(bh`_{-1}=2c;\ 1) \quad (2)$$

$$` \ 2\ f1;2;\quad ;d\log_2(\max(w_0;h_0))eg :$$



(a) Standard 2D texture **(b)** Patch texture

Figure 4: *A pictorial comparison of mipchains for a 64 64-unit texture size. Padding is required to align data to GPU texture tiles.*

For a standard 2D texture, the size of the texel array at mipmap level ` is $w_` \quad h_`$. 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 ` is $(w_` + 1) \quad (h_` + 1)$.

In-general, GPUs store texels in $n \quad m$ 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 cache line 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.

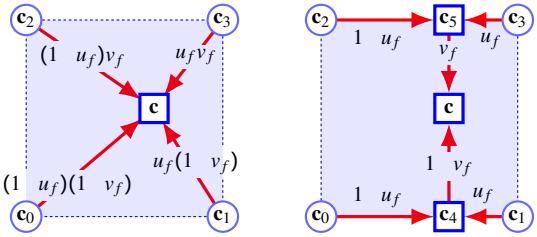
When the size of the texel array does not evenly divide the GPU tile size, the affected data must be padded out until it does. This increases the texture storage requirements, with larger textures experiencing proportionally less wastage. Figure 4 illustrates parts of the mipmap chain for a 64 64 standard 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 reduces 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 $hu; vi$, assuming that $hu; vi$ is not close to the borders of the texture, bilinear filtering on standard 2D textures uses the 2 2 texel region with array indices $hi; ji$, $hi + 1; ji$, $hi; j + 1i$, and $hi + 1; j + 1i$, where the integer indices are defined as $i = bu \quad 0{:}5c$ and $j = bv \quad 0{:}5c$. The bilinear filter weights are determined using fractional coordinates $hu_f; v_f i$, where $u_f = u \quad 0{:}5 \quad i$ and $v_f = v \quad 0{:}5 \quad j$.

For patch textures, the texels are on integral $hu; vi$ 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 $i = buc$ and $j = bvc$. The bilinear filter weights are determined using fractional coordinates $hu_f; v_f i$, such that $u_f = u \quad i$ and $v_f = v \quad j$.



(a) Weighted sum of four values **(b)** Three linear interpolations

Figure 5: *Bilinear filtering alternatives used in current GPUs.*

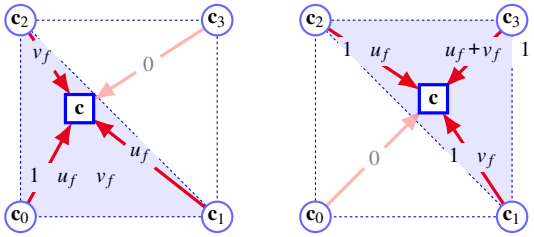


Figure 6: *Barycentric filtering using weighted sum of texels.*

However the texels and weights are determined, bilinear filtering can be computed using either a weighted sum of four texel values (Figure 5a) or three linear interpolations (Figure 5b). The first method allows more parallelism and is simpler when filtering floating-point texel values. The second method uses one fewer multiplier by rewriting the $u_f\mathbf{c}_1 + (1 \quad u_f)\mathbf{c}_0$ linear filtering equation as $\mathbf{c}_0 + u_f(\mathbf{c}_1 \quad \mathbf{c}_0)$. However, floating-point texel values would require normalizing in each linear interpolation stage.

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 [Yuk17] 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 backwards 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 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 $hu_f; v_f i$. 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 $\mathbf{c}_1$ and $\mathbf{c}_2$.

Figure 6 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 7 illustrates two ways to use three linear interpolations to perform barycentric interpolation. A few multiplexors are used to change the weights and texture values input to the linear interpolation stages, but the multiply/add logic in each linear interpolator remains the same.

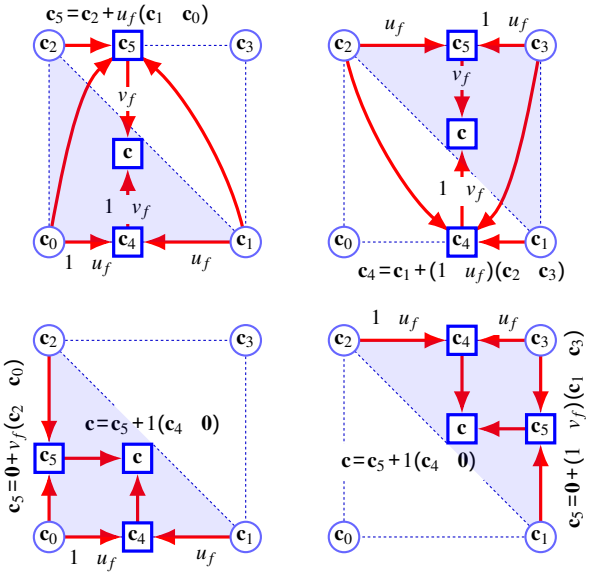


Figure 7: *Barycentric filtering using three linear interpolations using* ***(top)*** *a similar construction to bilinear filtering and* ***(bottom)*** *an alternative that eliminates the need for the third multiplication.*

For 2D textures, trilinear filtering 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 bilinear filtering operations along a line in texture space from an appropriate mipmap level. This approach can be used for handling patch textures as well. However, care must be taken about sampling the texture coordinates outside of the patch.

One approach is to simply not take a sample if its location is outside the patch. In-effect, this clips the part of the anisotropic filter kernel that falls outside the patch. Note that, if the filter kernel is clipped by one patch edge, since its center is guaranteed to be inside the patch, the larger portion of the kernel remains. This clipping obviously changes the filter kernel shape, but this is unlikely to introduce visible visual artifacts: the net effect is simply to limit the level of anisotropy, and the final result is computed using valid texture samples. 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. Therefore 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 [YKH10].

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 the patch. This must be followed by normalizing the result by the accumulated weights of the samples that are 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 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 is defined. A similar solution that clips the filter kernel has been applied for standard texture mapping as well [Tot13], but it only works for certain mappings and must be handled in software. 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 for anisotropic filtering 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. While the result can deviate from the intended anisotropic filtering computation, this is unlikely to produce visible artifacts, since all samples are taken from a nearby valid location. The advantage of this approach is that it requires no hardware change and has no performance impact.

3.6. Accessing Patch Textures in Shaders

Patch textures could be implemented as a collection of bindless textures. Following the existing graphics API for bindless textures, first, bindless texture handles must be generated for each patch texture. These handles can be flexible-enough to include all necessary information for accessing the texture data in shaders without any additional indirection cost. Thus, all that is needed for accessing a patch texture is its bindless texture handle, along with the patch coordinates to sample. 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 only 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 either using shader storage buffer objects or per-vertex attributes.