

Mesh Color Textures

Cem Yuksel
University of Utah
cem@cemyuksel.com



Figure 1: Example models with mesh color textures we used in our tests. The mesh colors values are sampled from 2D texture maps. The images are generated on the GPU using our trilinear filtering method with our 4D texture coordinate representation.

ABSTRACT

The fundamental limitations of texture mapping has been a long standing problem in computer graphics. The cost of defining and maintaining texture coordinates and the seams that introduce various filtering inconsistencies lead some graphics applications to adapt alternative techniques that directly address these problems, such as mesh colors. However, alternatives to texture mapping introduce run-time cost that contradicts with the performance constraints of real-time graphics applications. In this paper we introduce mesh color textures that offer all benefits of mesh colors to real-time graphics applications with strict performance constraints. Mesh color textures convert the mesh color data to a format that can be efficiently used by the texture filtering hardware on current GPUs. Utilizing a novel 4D texture coordinate formulation, mesh color textures can provide correct filtering for all mipmap levels and eliminate artifacts due to seams. We show that mesh color textures introduce negligible run-time cost with no discontinuity in texture filtering. We also discuss potential future modifications to graphics hardware and API that would further simplify the use of mesh color textures in real-time graphics applications.

CCS CONCEPTS

• **Computing methodologies** → *Texturing*;

KEYWORDS

Mesh colors, texture mapping, texture filtering

ACM Reference format:

Cem Yuksel. 2017. Mesh Color Textures. In *Proceedings of HPG 2017, Los Angeles, CA, USA, July 28-30*, 11 pages.
<https://doi.org/10.1145/3105762.3105780>

1 INTRODUCTION

Virtually in all computer graphics applications texture mapping is the essential means by which high-frequency surface details are represented. Textures typically store surface colors, but they can also be used for storing other information like various material properties, precomputed lighting, surface normals, displacement, etc. Yet, texture mapping has severe fundamental flaws that inflate the manual labor cost, negatively impact the final image quality, and limit the use of advanced features like tessellation and displacement mapping. We group the problems of texture mapping into two categories: parameterization and filtering.

Parameterization is the process of defining a *mapping* from the model space to the (typically 2D) texture space. While researchers continue to develop ingenious ways to automate this process [Ray et al. 2010; Smith and Schaefer 2015; Tarini 2016], in practice it still requires extensive manual effort. Though the texture resolution on a part of the surface is often directly tied to its size, in reality the desired resolution also depends on the deformation that a surface undergoes through an animation sequence and the desired amount of detail an artist would like to add to a certain part of a model (for

example, an artist may want to have a higher resolution texture representation for the face of a character). Unfortunately, these require user intervention and preclude the extensive use of fully automated methods in practice. The parameterization process is further complicated by the fact that the 3D model creation and texture preparation are virtually separated. Thus, texture mapping is often completely excluded from the 3D modeling process, and post-iterations on the model geometry and topology after texture mapping may require redefining the parameterization and subsequently repainting the texture (partially or completely). Therefore, parameterization has been a tedious process even for highly skilled and experienced artists.

The second group of problems are related to the filtering operations that are used for computing the value of a texture at a point (or an area) on the model surface. Baring limited special cases, seams are unavoidable with texture mapping and it is extremely difficult, if possible at all, to make sure that the interpolated color values on either side of a seam would agree. Thus, seams lead to visible discontinuities on the surface. The process of hiding seams is typically left to the artist who paints the texture and her ability to manually match the interpolated colors (and causes additional manual labor). Even though minute differences in color values can be tolerated due to the limitations of human visual system, they can lead to unacceptable results when textures are used for geometric information, such as cracks on surfaces with displacement maps. In fact, this very problem was attributed as the primary reason that prevents widespread use of displacement maps in video games [Tatarchuk 2016], thereby limiting the use of the existing tessellation hardware on the GPUs. Moreover, mipmapping, the primary method by which minification filtering is computed, is broken because of seams: manual efforts of hiding seams cannot work beyond a few mipmap levels (if that), as the mipmap levels store wrong information by filtering the texture values based on proximity in texture space, rather than model space. Also, texels near seams can be shared by unrelated parts of the model. Similarly, anisotropic filtering near seams can produce incorrect results, which are also unavoidable.

In offline computer graphics applications these severe problems of texture mapping can be completely avoided by using alternatives of texture mapping that directly address these problems, like mesh colors [Yuksel et al. 2010] or Ptex [Burley and Laceywell 2008]. However, these custom solutions require custom filtering operations that are not supported by current GPUs. Having in mind that hardware texture filtering can be more than an order of magnitude faster than software implementations, custom texture filtering is entirely unacceptable for video games and other real-time graphics applications with performance constraints. In fact, Intel’s Larrabee architecture [Seiler et al. 2008], which proposed removing almost all fixed function units from the GPU, still kept the texture filtering unit for this very reason. Therefore, the problems of texture mapping are still sources of real concern for real-time computer graphics applications, and they continue to occupy a significant portion of artist time, which dominates the monetary cost of AAA video game production, as recently stated by developers from Bungie and Activision studios [Tatarchuk et al. 2015].

An ideal solution to the problems of texture mapping for real-time graphics applications must use the existing GPU hardware and

should impose minimal run-time computation cost. In this paper, we introduce such a solution that converts mesh colors to *mesh color textures* with 4D encoded texture coordinates, a format that can be efficiently used on existing GPUs. We also provide solutions to the limitations of mesh colors and offer relatively simple modifications to graphics hardware and API that would further simplify the use of mesh color textures in real-time graphics applications. In particular, this paper contains the following technical contributions:

- A method that converts mesh colors to a 2D texture representation with correct bilinear filtering,
- A novel 4D texture coordinate formulation that provides correct bilinear filtering for any mipmap level,
- A texture layout generation method and an optimization scheme that allows mipmap levels beyond vertex colors,
- A formulation that permits nonuniform mesh color resolutions on elongated triangles, and
- A solution for preventing anisotropic filtering from sampling unacceptable texture locations in future hardware.

As a result, we provide a comprehensive solution that directly addresses the problems of texture mapping in real-time graphics applications. The solution we provide adds negligible computation cost during rendering by utilizing existing texture filtering hardware, completely eliminates the need for model parameterization, and produces no filtering inconsistency in any mipmap level without any manual effort. Therefore, we not only minimize the need for manual labor, but also allow high-quality filtering operations with mipmaps and resolve the issues that lead to limited use of tessellation and displacement mapping. Most importantly, our solution provides the authoring benefits of mesh colors without introducing noticeable run-time overhead. Some example images of models with mesh color textures captured from our implementation are shown in Figure 1.

2 BACKGROUND

There is a large body of work on surface parameterization [Floater and Hormann 2005; Hormann et al. 2007; Sheffer et al. 2006]. However, parameterization suitable for texture mapping should be ideally bijective or at least injective, which is guaranteed by few methods. Some of them repeatedly split the surface until each piece can have a bijective parameterization [Lévy et al. 2002; Zhou et al. 2004] or grow (potentially) multiple regions starting from seed triangles [Sorkine et al. 2002]. Global parameterization methods can guarantee local injectivity by preventing triangle flips [Hormann and Greiner 1999; Sander et al. 2001; Schüller et al. 2013] and bound the triangle distortion [Aigerman et al. 2014; Lipman 2012; Poranne and Lipman 2014]. Bijectivity can be achieved through non-linear optimization to planarize either separate parts of a model [Zhang et al. 2005] or an entire model [Smith and Schaefer 2015]. Skeleton texture mapping [Madaras and Đuriković 2012] first computes a global skeleton for the model and then separates the surface into rectangular pieces. Volume-encoded UV-maps [Tarini 2016] can automatically parameterize the surface by mapping each point using its 3D position, thereby directly support multi-resolution representations. However, since a desirable parameterization for texture mapping is not always the one that minimizes triangle distortions and it can depend on the artist’s intentions and how the model will

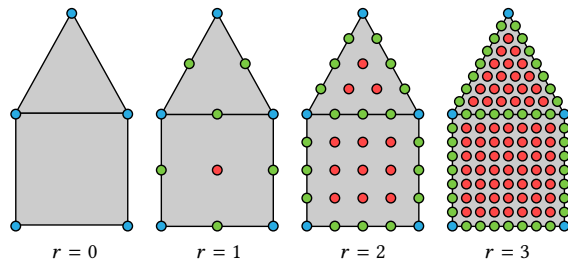


Figure 2: Mesh colors on triangles and quadrilaterals with different resolutions $R = 2^r - 1$: (blue) vertex colors, (green) edge colors, and (red) face colors.

be used, the parameterization process involves manual intervention in practice. Also, regardless of the parameterization method used, seams are unavoidable and they lead to problems in filtering.

For hiding the seams researchers proposed quad segmentation that maps each surface piece to a square portion of the texture space with half a pixel boundary [Nießner and Loop 2013; Purnomo et al. 2004]. This allows a limited number of mipmap levels by adjusting the texture coordinate accordingly for each level. The resolution of each quad piece can also be independently specified [Carr and Hart 2004]. Using a grid-preserving parameterization avoids the need for enforcing a square-shaped boundary for each piece [Ray et al. 2010]. Color discontinuities near the seams can also be hidden by triangulating the seams in texture space with additional computation in the fragment/pixel shader [González and Patow 2009]. The artifacts of displacement mapping can be fixed using analytically computed normals and a tile-based texture format [Nießner and Loop 2013].

Other approaches aim to avoid the problems of 2D parameterization by redefining texture representation. Octree textures [Benson and Davis 2002; DeBry et al. 2002] and brick maps [Christensen and Batali 2004] store colors efficiently in a 3D embedding, but they may require too many levels to avoid color bleeding across geometrically close surfaces and GPU implementations [Kniss et al. 2005; Lefebvre et al. 2005; Lefohn et al. 2006] must use multiple dependent texture lookups. Hash textures [Lefebvre and Hoppe 2006] avoid the color bleeding problem using a hash function in a 3D embedding, but nonuniform resolution adjustment is not permitted. Polycube maps [Tarini et al. 2004] use a parameterization onto the faces of multiple cubes that approximate the shape of the model, but they are not general enough to handle any model geometry. TileTrees [Lefebvre and Dachsbacher 2007] store texture values on 2D elements placed at the leaf nodes of an octree, thereby share the limitations of octree textures.

2.1 Mesh Colors

Mesh colors [Yuksel et al. 2010] eliminate the fundamental problems of texture mapping by defining the colors directly on the model surface. Using the existing topology of a polygonal mesh, mesh colors allow generating detailed textures on arbitrary surfaces without parameterization. Therefore, any model (containing triangles and quadrilaterals) can be painted with mesh colors directly without any manual or automatic parameterization effort.

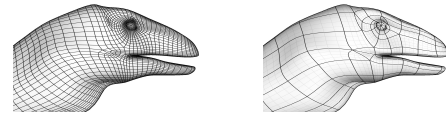


Figure 3: An example polygonal mesh and its low-resolution canvas mesh used for defining mesh colors.

Mesh colors extend the concept of vertex colors by introducing edge colors and face colors as shown in Figure 2. The color positions are evenly distributed in the barycentric space for triangles and bilinear space for quadrilaterals. Therefore, no additional data structure is needed, beyond the resolutions of each face, which can be adjusted independently as desired. This permits simultaneous model editing and texture painting. Furthermore, by placing the colors exactly on the vertices and along the edges, mesh colors avoid discontinuities in filtering operations.

It is often advantageous to define mesh colors on a low-resolution version of a mesh that uses a subset of its vertices. We refer to the mesh that is used for defining mesh colors as the *canvas mesh* (Figure 3). Mesh colors of a canvas mesh can be directly used by any arbitrary tessellation of the canvas mesh. Remeshing is not directly supported, but since each color value has a well-defined 3D position (defined by its embedding), resampling is trivial. Mesh colors also permit simultaneous model editing and texture painting.

All these properties make mesh colors an ideal alternative to texture mapping for a production pipeline that uses 3D painting [Lambert 2015]. For the purposes of real-time rendering, however, mesh colors share the same limitations with most other methods, that is a custom shader is required, which is often a deal-breaker. Moreover, while mesh colors provide correct mipmap treatment, the lowest resolution mipmap level is the vertex colors of the canvas mesh, which is an important limitation especially when the sizes of faces vary significantly, as it often is the case. Furthermore, since the resolution is defined uniformly on each triangle, elongated triangles can lead to extra color storage. The mesh color textures we introduce in this paper directly address all of these problems and make mesh colors a real alternative to texture mapping for real-time rendering as well.

2.2 Per-face Textures (Ptex)

Ptex [Burley and Laceywell 2008] avoids the problems of texture mapping by simply using a separate texture for each face of a model. While this may sound inefficient at first glance, it is far more preferable to the problems of texture mapping in a production environment. That is why Ptex is becoming increasingly popular for offline rendering applications, led by Disney Animation and Pixar studios.

Ptex and mesh colors are very closely related. In terms of the color placement topologies, Ptex and mesh colors are dual pairs. In other words, the color locations of mesh colors are exactly at the center of color locations used by Ptex (and vice versa), as shown in Figure 4. Mesh colors guarantee texture filtering continuity by placing colors exactly on the vertices and along the edges. Ptex, on the other hand, accesses the colors of neighboring faces during texture filtering to achieve continuity (by storing adjacency data per polygon).

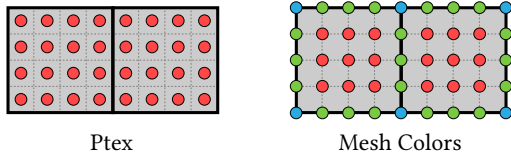


Figure 4: Color locations of Ptex and mesh colors on two square-shaped quadrilateral faces.

However, this theoretically minor difference in color placement makes Ptex less suitable for real-time graphics applications. First of all, Ptex cannot properly filter near extraordinary vertices that are not 4-valence (vertices where more or less than 4 edges meet) without special treatment during filtering. Unfortunately, extraordinary vertices are commonplace. More importantly, Ptex was designed for quad faces, not triangles, which are heavily used in real-time graphics applications, though it is possible to extend Ptex to equilateral triangles (by packing them as quads). Also, converting triangles to quadrilaterals via subdivision produces extraordinary vertices.

Therefore, in this paper we concentrate of mesh colors, instead of Ptex. Nonetheless, most of the methods we introduce in this paper can be trivially adapted to Ptex as well (with quad-only meshes).

3 MESH COLOR TEXTURES

Since we are targeting real-time graphics applications, we must convert mesh colors to 2D textures, which can be efficiently used by existing GPUs. This way, the GPU can directly handle bilinear texture filtering and we benefit from all existing hardware optimizations, including caching. Converting mesh colors to a 2D mesh color texture for bilinear filtering (Section 3.1) is relatively simple, but care must be taken to avoid seams. Mipmaps (Section 3.2) require special treatment that we efficiently handle at run-time using a 4D texture coordinate formulation (Section 3.3). Supporting mipmap levels beyond vertex colors (Section 3.4) is trivial at run-time, but requires additional considerations for generating the mipmap levels. Anisotropic filtering (Section 3.5) could also be achieved, but it would require hardware modifications to avoid software filtering. At the end of this section we also provide an extension to mesh colors that allows nonuniform mesh color resolutions on skinny triangles (Section 3.6).

3.1 Bilinear Filtering

For bilinear filtering it is sufficient to generate a single 2D texture from mesh color values. We first assign a set of 2D texture coordinates for the vertices of each face separately to form our *texture layout*. Then, we copy the mesh color values to a 2D texture using these coordinates. Finally, we assign colors for some texels between faces that are used during bilinear filtering. Note that our treatment of this process bears similarities to prior work [Carr and Hart 2002; Purnomo et al. 2004], but our method differs with its ability to handle correct bilinear filtering for triangles.

To generate the texture layout, we place the vertices of each face exactly at texel centers, such that the edges lie precisely horizontally, vertically, or diagonally, with as many texels in between the vertices as specified by the face resolution (Figure 5). This way, all mesh

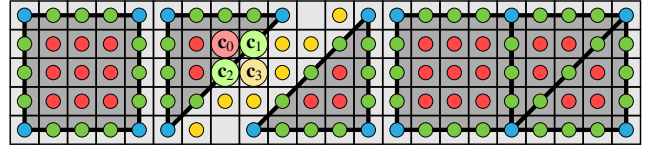


Figure 5: Texture layout of mesh colors on a 2D texture.

color values (for vertices, edges, and faces) correspond to the center of a unique texel and assigning the texel color values becomes trivial.

In the mesh color representation, the color value along an edge only depends on the edge colors (and the vertex colors). This is how filtering continuity across an edge is guaranteed. In our mesh color texture representation, filtering continuity is guaranteed automatically for horizontally and vertically placed edges. However, diagonally placed edges require some extra effort. The colors along these edges not only depend on the edge colors (such as c_1 and c_2 in Figure 5) as desired, but also the two other nearby colors, only one of which is associated with the face (c_0) and other is on the opposite side of the edge (c_3). Fortunately, we can ensure that the color along the edge only depends on c_1 and c_2 by simply setting

$$c_3 = c_1 + c_2 - c_0. \quad (1)$$

As a result, bilinear filtering of these four color values using parameters $s, t \in [0, 1]$ simplifies down to $sc_1 + (1-s)c_2$ along the edge, where $s = 1 - t$. Thus, we merely need to add a single texel-wide border near diagonally placed edges for storing the interpolation colors c_3 on the opposite side of the edge. This way, an edge that is placed diagonally for one face can be placed vertically or horizontally for the other face that shares it, without causing filtering discontinuity. This is an important property for supporting arbitrary triangular meshes.

However, most color formats used for texture storage only support values between 0 and 1. Though it is likely that c_3 would be within those limits when c_0, c_1 , and c_2 have similar values, there is no guarantee. If c_3 cannot be represented correctly, we cannot achieve filtering continuity. Therefore, when we detect that a color component of c_3 is negative, we modify the original mesh colors by adding $\delta = -c_3/3$ to c_1 and c_2 , and subtracting δ from c_0 . This provides the minimum required change in color values to set the new value $c_3' = 0$. Similarly, if a color component of c_3 is greater than 1, we use $\delta = (1 - c_3)/3$. It is possible that some of these colors may be modified again due to similar constraints for different faces. Yet, since the modification brings color values closer, it is guaranteed to converge after a few iterations.

Note that in the absolute worst case, when the edge colors are black ($c_1 = c_2 = 0$) and the face color is white ($c_0 = 1$), the color values would be modified by 0.33. In a more typical case, the modification (if any) would be minor, as it is only needed when the face color is at least twice as bright as the average of the edge colors, and the maximum modification of $c_0/3$ only happens when both edge colors are black. Nonetheless, in practice it might be a good idea to apply the color modification as the texture is painted, so that the user can immediately respond, if a modification occurs.

Another issue to consider is that Equation 1 effectively converts the bilinear filtering near the diagonal edge to barycentric filtering



Figure 6: Mesh colors on (a) two triangles that form a square filtered using (b) a barycentric filter near diagonal edges, revealing the diagonal edge without causing discontinuity, and (c) a bilinear filter that filters across the diagonal edge.

that interpolates the three nearest colors. This is indeed the desired behavior and avoids filtering discontinuities, but it results in a different filtering behavior near edges that are placed diagonally on the texture layout. If the two triangular faces of an edge have the same resolution and if they both place this edge diagonally, the switch to barycentric filtering near the edge might visually reveal the edge as shown in Figure 6b (even though it would not cause discontinuity). That is why, in this special case, it is a better idea to copy the color for c_3 directly from the corresponding color of the opposing face, so that we can get continuous bilinear interpolation across the diagonal edge (Figure 6c). This would also eliminate the need for Equation 1 for the copied texels and the possibility of any color modification. Alternatively, two such triangles can be packed as a single quad, virtually eliminating the diagonal edge altogether (and the wasted space for duplicating the edge colors and the borders).

Obviously, converting mesh colors to a mesh color texture using this procedure would copy each edge color (at most) twice and each vertex color multiple times. Yet, the minimal wasted space (mostly due to color duplication) is a small price we pay for achieving hardware-accelerated bilinear filtering. Note that the number of duplicated colors can be minimized using a lower resolution canvas mesh and by overlapping the shared edges of neighboring faces on the texture layout, as shown in Figure 5 (right).

Since mesh color textures generated in this form guarantee continuous bilinear filtering, they are suitable for displacement mapping. However, it is important to note that even though the color values are copied without resampling (i.e. interpolation), since the currently supported hardware tessellation pattern does not align with the mesh color positions, a tessellated vertex inside a triangular face does not correspond to a single mesh color value and its value must be interpolated. A variant of mesh colors addresses this issue by placing face colors using the current hardware tessellation pattern [Schäfer et al. 2012], but mesh colors defined with this pattern cannot be converted to a mesh color texture without resampling. Quadrilateral faces, however, do not require interpolation as long as the tessellation resolution matches the resolution of the face.

3.2 Generating Mipmaps

We assume that the mesh color resolutions use powers of 2. This assumption has limited practical consequence and it also helps when two neighboring faces have different resolutions. Let R_i denote the resolution of a face i (i.e. the number of colors along an edge).

When $R_i = 2^{r_i} - 1$ for $r_i \in \mathbb{Z}^*$, if two faces i and j sharing an edge have different resolutions $r_i < r_j$, the edge colors for i can be freely painted and the additional edge colors for j are interpolated. Therefore, when using mesh colors, it is almost always a good idea to restrict the face resolutions to powers of 2 (minus one).

We follow the mipmap generation process of the original mesh colors method [Yuksel et al. 2010], which is different than mipmap generation for standard 2D textures. Therefore, we cannot simply create the mipmap levels from the mesh color texture directly. Instead, the mipmap levels for mesh colors must be computed first, and then each level must be converted to a mesh color texture separately. Better results can be achieved using a geometry-aware filtering scheme [Manson and Schaefer 2012].

Note that the lowest resolution for mesh colors is vertex colors (i.e. $r_i = 0$ and so $R_i = 0$). Therefore, mipmap levels can be generated with decreasing resolution down to vertex colors. For generating higher mipmap levels we continue filtering the vertex colors, but the storage resolution is not reduced. While this is certainly a limitation, it is a substantial improvement in comparison to standard 2D textures with seams, for which we cannot even expect to have more than a few reliable mipmap levels in general.¹

3.3 Trilinear Filtering with Mipmaps

The main difficulty with trilinear filtering is that the 2D texture coordinates for each mipmap level of mesh color textures are different. Unfortunately, naively storing a separate texture coordinate for each mipmap level would be too expensive. Therefore, we introduce a simple 4D texture coordinate representation that allows quickly computing the 2D texture coordinates for any mipmap level.

Let \mathbf{u}_ℓ be the *non-normalized* 2D texture coordinates for mipmap level ℓ . At runtime we can compute it using

$$\mathbf{u}_\ell = \mathbf{u}_s / 2^\ell + \mathbf{u}_\delta, \quad (2)$$

where \mathbf{u}_s is a 2D *scalable coordinate* representing the portion of the texture coordinate that changes for each mipmap level and \mathbf{u}_δ is a 2D *constant offset*, forming our 4D texture coordinate representation. In this formulation $\mathbf{u}_0 = \mathbf{u}_s + \mathbf{u}_\delta$ is the texture coordinate for the highest resolution mipmap level with $\ell = 0$. Note that in practice the division in Equation 2 can be replaced with multiplication by storing $\mathbf{u}_s / 2^{\ell_{\max}}$ instead of \mathbf{u}_s , where ℓ_{\max} is the maximum mipmap level, though we have not included this optimization in our tests.

For the sake of simplicity, in this section we only consider mipmap levels up to vertex colors (determined separately for each face based on its resolution). We refer to these mipmap levels as *valid* mipmap levels in this section. We discuss how to handle higher level (lower resolution) mipmap levels in the next section.

The 4D texture coordinate formulation in Equation 2 can be used in numerous ways. To create a desirable layout for our mesh color textures, however, we must follow three simple rules:

- (1) \mathbf{u}_ℓ of each canvas mesh vertex for each valid mipmap level ℓ must be at the center of a texel. This means that \mathbf{u}_s must be an even multiple of 2^{ℓ} (where $R_i = 2^{r_i} - 1$ is the face resolution, as mentioned above) and the \mathbf{u}_δ values must be

¹Texture mapping by tiling a small texture over a surface can have proper mipmap levels down to a single pixel, but this approach is seldom useful for texturing complex models (such as characters).

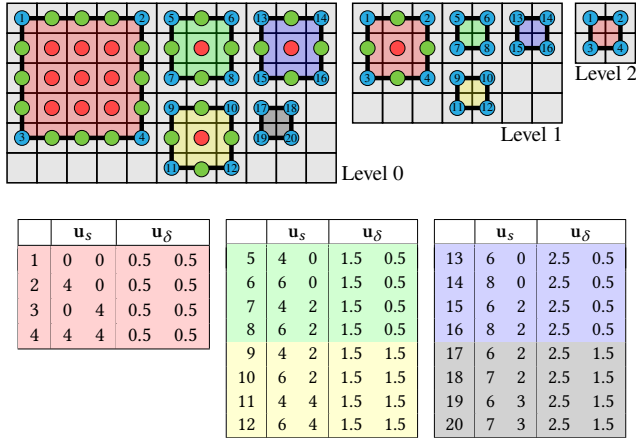


Figure 7: An example texture layout for 5 quad faces (each colored differently) on 3 mipmap levels and the list of 4D texture coordinates for each vertex of each face.

some integer plus 0.5 to move the texture coordinate to the texel center (Figure 7). Thus, u_s provides integer values for all valid mipmap levels.

- (2) The offset u_δ must be the same for all vertices of a face and the u_s values must vary by exactly 2^{r_i} (obeying the first rule). This places a quad face onto a $2^{r_i-1} + 1$ square texel region for all valid mipmap levels ℓ , allocating exactly as many texels as needed for copying vertex, edge, and face colors, as shown in Figure 7. Note that since we map each canvas mesh face independently, we assign each vertex as many separate coordinates as the number of faces sharing it. Also, because all offsets u_δ of all vertices of a face must be the same, if desired, it is possible to store u_δ per face, instead of storing them separately for each vertex of each face.
- (3) The mapped positions of separate faces must not overlap for any valid mipmap level. This can be accomplished by simply making sure that the u_δ values of a face are no smaller than the preceding faces (placed on the left or above) in the texture layout.

As long as these three rules are followed, any packing algorithm can be used for generating the texture layout. In our experiments we have used the packing algorithm outlined in Algorithm 1 (where $b = 1$). This algorithm begins with pairing triangles into rectangles that contain two triangles with 2 texel spacing between them to make room for the extra texels needed during bilinear filtering (such as the yellow texels in Figure 5). Each quad face is placed in a separate rectangle. Then, we sort the rectangles based on their sizes (using first height then width). Starting with the largest rectangle, we place them on the mesh color texture in scanline order (left to right then top to bottom). If placing a rectangle on the current line would make the length of the line (the width of the texture) larger than a desirable value (taken as the square root of the total number of colors to be copied), we move to the next line (past the last non-empty texel of the line). $u_{\delta,y}$, the vertical coordinate of u_δ , is the same for all rectangles placed on the same row and it is taken as the number of preceding rows (when $b = 1$). $u_{\delta,x}$, the

Algorithm 1: Texture layout generation with 4D coordinates.

```

1 Form rectangles from faces.
2 Sort rectangles from large to small.
3  $u_s \leftarrow [0 \ 0]$ 
4  $u_\delta \leftarrow [0.5 \ 0.5]$ 
5 foreach rectangle do
6   if reached end of row then
7     Find next row
8      $u_\delta \leftarrow u_\delta + [0 \ b]$ 
9      $u_s \leftarrow$  next row starting position  $-u_\delta$ 
10  end
11  Make sure that  $u_s$  is an even multiple of  $2^{r_i}$ .
12  Place rectangle using the four texture coordinates:
13     $u_s, u_\delta$ 
14     $u_s + [0 \ 2^{r_i}], u_\delta$ 
15     $u_s + [2^{r_i} \ 0], u_\delta$ 
16     $u_s + [2^{r_i} \ 2^{r_i}], u_\delta$ 
17     $u_{s,x} \leftarrow u_{\delta,x} + 2^{r_i}$ 
18     $u_{\delta,x} \leftarrow u_{s,x} + X_i + b$ 
19 end

```

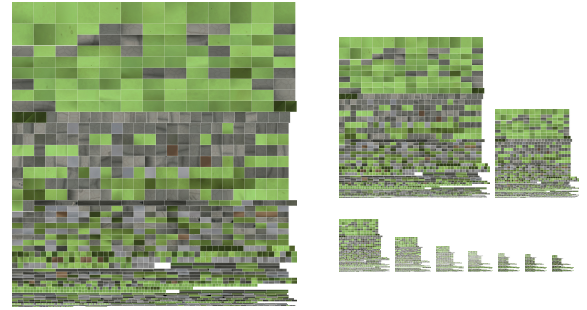


Figure 8: A mesh color texture and its mipmap levels.

horizontal coordinate of u_δ , is the number of rectangles on the left side (when $b = 1$) plus an additional offset X_i , where $X_i = 0$ for rectangles containing quad faces and $X_i = 2$ for rectangles containing triangle pairs (to account for the constant separation between triangles). Before placing a rectangle on a new row, we make sure that the horizontal component of u_s is an even multiple of 2^{r_i} (and skip texels if necessary).² An example layout generated using this procedure is shown in Figure 8. Note that this is just one way of generating a valid layout. We leave the exploration of an optimal layout generation (producing optimal packing) to future work.

While forming rectangles from faces, it is possible to group neighboring faces into larger rectangles (as in Figure 5 (right)). In our tests, however, we haven't used this extra optimization that would reduce the number of duplicated color values.

²In most cases u_s is an even multiple of 2^{r_i} , but exceptions are possible, in which case we waste some pixels to ensure that the first rule is satisfied. Notice the two rectangular white holes near the bottom of the textures in Figure 8.

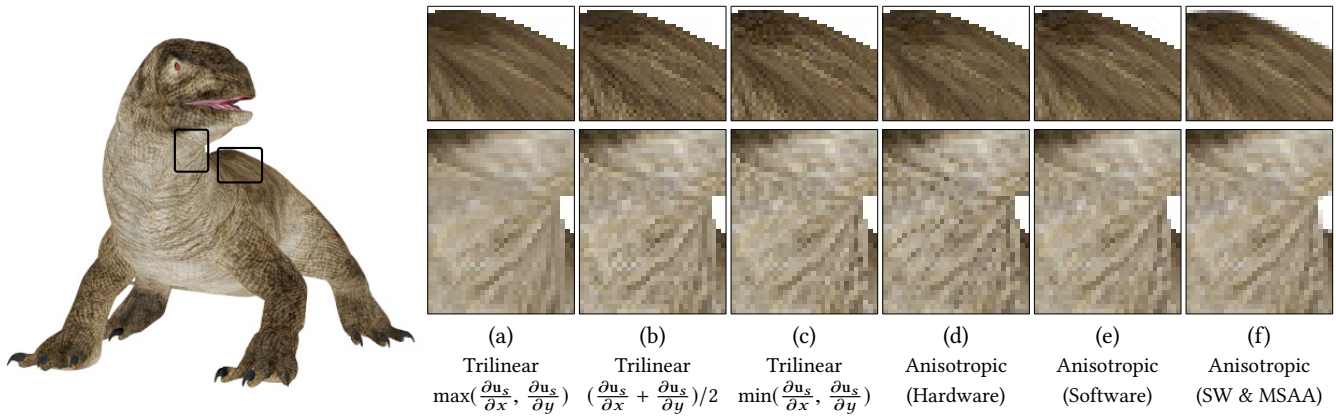


Figure 9: Options for trilinear and anisotropic filtering using mipmaps. For trilinear filtering we can use (a) the maximum screen space derivatives, (b) the average derivatives, or (c) the minimum derivatives. For anisotropic filtering the minimum derivatives are used. (d) Hardware anisotropic filtering introduces inaccuracies that reveal the edges of the canvas mesh. (e) Software anisotropic filtering can resolve this but software filtering introduces run-time computation cost. (f) For correct anisotropic filtering anti-aliasing is needed to consider the contributions of each face that corresponds to a pixel.

Using this method we can generate a texture layout with 4D coordinates and use it for generating a mesh color texture for each mipmap level. Yet, this procedure only supports mipmap levels up to vertex colors. Since faces with higher resolution mesh colors are placed first near the top part of the texture, the bottom/right part for higher mipmap levels can be simply cropped (as in Figure 7), as the bottom part would correspond to invalid mipmap levels (beyond vertex colors) for faces with lower-resolution mesh colors.

During rendering, we determine the mipmap level using the screen space derivatives of \mathbf{u}_s . The filtering result using different trilinear filtering options are shown in Figure 9. We perform two hardware accelerated bilinear filtering operations using the two closest mipmap levels, and linearly interpolate the result to achieve trilinear filtering. In our implementation we store each mipmap level in a different texture.

3.4 Mipmaps beyond Vertex Colors

Mipmap levels up to vertex colors can be arguably enough for many applications, since higher mipmap levels are needed only when the faces of the canvas mesh appear smaller than a pixel on the rendered image. Nonetheless, the procedure explained above can be extended to support mipmap levels beyond vertex colors.

These higher mipmap levels are stored as vertex colors, though they are generated via additional filtering operations. Therefore, if all faces have the same mesh color resolution, there is a simple solution for supporting higher mipmap levels. In that case, we can use the same method explained above and merely replace 2^ℓ in Equation 2 with $2^{\min(\ell, m)}$, where m is the number of mipmap levels up to vertex colors (such that $m = r_i \forall i$). However, for more typical cases where faces can have different mesh color resolutions, this simple modification is not enough.

Note that our 4D texture coordinate representation places the vertices of the canvas mesh at the exact centers of texels for all mipmap levels up to vertex colors. For higher mipmap levels, however, the

scalable portion of the texture coordinate ($\mathbf{u}_s/2^\ell$ in Equation 2) may not be whole numbers, meaning that the vertices of the canvas mesh would no longer be at the exact centers of some texels. This leads to two problems:

- (1) Mesh color values for the higher mipmap levels beyond vertex colors cannot be directly copied to texels. Instead, the mesh color values must be represented as bilinear combinations of texel colors.
- (2) At higher levels the texel regions that correspond to neighboring rectangles of the layout can overlap as shown in Figure 10a.

The second problem is easy to solve. All we need to do is to add a single texel separation between rectangles while generating the layout, as shown in Figure 10b. This can be accomplished by merely using $b = 2$ in Algorithm 1. Since this separation is encoded in the constant (\mathbf{u}_s) portion of the texture coordinate, the separation is preserved for all mipmap levels. This separation ensures that for any mipmap level ℓ there is a 2 texel size difference between the \mathbf{u}_ℓ values of neighboring rectangles. At higher mipmap levels, these new separation texels can be included in the bilinear interpolation. This simple solution guarantees that each texel would correspond to no more than a single rectangle. Thus, we guarantee that the texel regions of neighboring rectangles never overlap.

The first problem, however, is not as easy to solve. When texture coordinates of vertices do not correspond to texel centers, we cannot simply copy the mesh color values. Instead, we must assign colors to the texture, such that the interpolated values would correspond to the mesh color values we aim to represent. Unfortunately, accurately reproducing the mesh color values from the interpolated texel colors might require storing texel values that are negative or greater than one. Though the texel colors are easy to compute using bilinear extrapolation of the original mesh color values, there is no guarantee that the extrapolated color values would be within the limits $[0, 1]$. In fact, for higher mipmap level where the texture coordinates \mathbf{u}_ℓ for the vertices of a face are close,

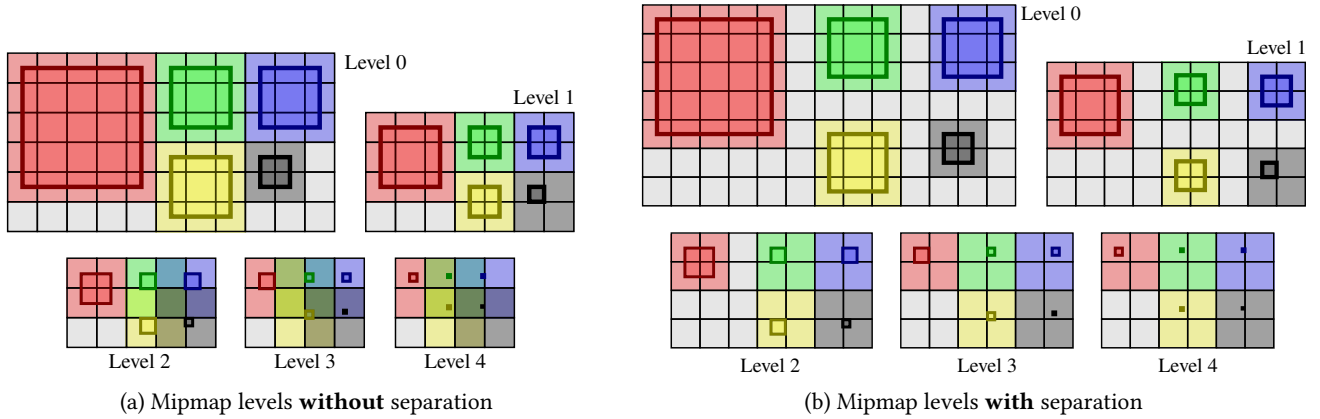


Figure 10: Example texture layouts generated with and without separation, showing which texels are associated with which faces. Notice that without separation the texels that correspond to different faces begin to overlap at higher mipmap levels (beyond vertex colors). Adding a single texel separation ensures that each face is mapped to a distinct set of texels.

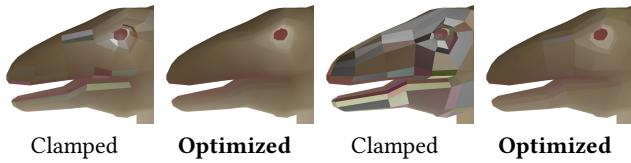


Figure 11: Two mipmap levels beyond vertex colors, generated using clamping that can produce severe artifacts, and our optimization that resolves this problem.

even minute differences between the original mesh color values can lead to extrapolated texel color values that are far outside of the limits $[0, 1]$. If the texture format does not permit storing values outside of $[0, 1]$, we cannot ignore this problem by simply clamping the extrapolated color values, because that might easily introduce substantial amount of error to the interpolated mesh color values computed during filtering, as shown in Figure 11.

Instead of simply clamping the extrapolated texel color values that exceed the limits $[0, 1]$, we propose modifying the original mesh color values of a face, such that the extrapolated texel color values reside within $[0, 1]$. Thus, we convert the problem of computing the extrapolated texel colors to an optimization problem that computes the minimal changes to the original mesh color values of the mipmap level that would produce extrapolated texel color values within the limit $[0, 1]$.

We perform the optimization separately for each color channel (red, green, and blue) and for each face. Let \mathbf{x} denote the change in the mesh color values that would make the extrapolated value at a point k with extrapolation weights \mathbf{w}_k equal to a limit value $L_k \in \{0, 1\}$ (whichever limit is violated), and let C_k be the extrapolated value at point k using unmodified mesh color values. Our goal is to solve for \mathbf{x} and the points k we consider are texel centers where the extrapolated values are out of bounds $[0, 1]$. Note that vectors \mathbf{x} and \mathbf{w}_k are four dimensional ($d = 4$) for quadrilateral faces (using bilinear interpolation) and three dimensional ($d = 3$)

for triangles (using barycentric interpolation). If adding \mathbf{x} to the original mesh color values would make the extrapolated value at k equal to the limit value L_k , then we can write

$$\mathbf{w}_k^T \mathbf{x} = L_k - C_k. \quad (3)$$

We can solve \mathbf{x} using least square minimization of the form

$$\min_{\mathbf{x}} \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2, \quad (4)$$

where the vector \mathbf{b} and the matrix \mathbf{A} are

$$\mathbf{b} = \mathbf{S} \begin{bmatrix} L_1 - C_1 \\ L_2 - C_2 \\ \vdots \\ L_n - C_n \end{bmatrix}_{n \times 1} \quad \text{and} \quad \mathbf{A} = \mathbf{S} \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \vdots \\ \mathbf{w}_n^T \end{bmatrix}_{n \times d}, \quad (5)$$

where $n \leq 9$ is the number of texels that correspond to the face and \mathbf{S} is an $n \times n$ diagonal binary matrix, such that $S_{k,k} = 1$ only if $C_k \notin [0, 1]$ and $S_{k,k} = 0$ otherwise. Thus, matrix \mathbf{S} determines which ones of the n texels are considered as constraints in the optimization. When only a few points k are out of bounds, the optimization problem is under-constrained, but we can still find a solution to $\mathbf{A}^T \mathbf{A} \mathbf{x} = \mathbf{A}^T \mathbf{b}$ numerically using conjugate gradients that would minimize $\|\mathbf{x}\|$. Since this is a very small system, the solution can be computed quickly.

However, after the optimization, the modification of the mesh color values \mathbf{x} might cause other texels of the face to be out of bounds $[0, 1]$ (even if they were previously within bounds). Therefore, we perform this optimization in multiple steps by introducing the constraints one by one. At each step h we add the constraint for the point k with the largest $|L_k^h - C_k^h|$ value, where the values for L_k^h and C_k^h are determined incorporating the \mathbf{x}^{h-1} values computed in the previous step (where $\mathbf{x}^0 = 0$). The maximum number of steps needed is bounded by n . Using this optimization we substantially reduce the interpolation error during bilinear filtering (Figure 11).

Even though each step of this optimization can be quickly solved, since this process is repeated for (potentially) each face of the canvas

mesh, the cumulative processing time can be significant, which was up to several seconds in our tests.

Also, it is possible that the modifications on the mesh color values for a particular face can cause other interpolated texel colors of other faces to fall out of bounds. To ensure that the modification on a vertex color would not lead to any discontinuity, this optimization problem must be solved globally, considering all interpolated texels and all related vertex colors. This is certainly possible to do using a much larger system of equations. However, this optimization is needed only for higher mipmap levels (and only if the image format is restricted to $[0, 1]$). Therefore, in practice, depending on the target application, minor discontinuities due to this modification might be acceptable and the colors used for each face can be optimized independently. Without this optimization, however, the resulting color differences can be severe (Figure 11).

3.5 Anisotropic Filtering

Hardware anisotropic filtering can be enabled while using mesh color textures. However, current GPUs automatically determine the sample locations for anisotropic filtering and they offer no mechanism for restricting these sample locations. Therefore, some of the mesh color texture samples can be taken outside of the face area on the texture layout. As a result, some anisotropic filtering samples may not correspond to the face being shaded, leading to inaccuracies in filtering and revealing the edges of the canvas mesh as shown in Figure 9d.

Unfortunately, we can offer no solution for this using existing GPUs, besides handling the anisotropy in software. However, this can be potentially remedied in future GPU hardware by introducing a relatively simple test. To avoid the anisotropic filtering issues, all we need to do is to make sure that the texture sample locations are contained within the texture-space footprint of the shaded fragment's triangle. The texture samples that are outside can be discarded to avoid invalid samples. Even though the triangle information may not exist during fragment shading, it is easy to perform this test using the barycentric coordinates of the fragment and their screen space derivatives. Eliminating these invalid samples, along with multi-sampling for antialiasing, would produce the desired anisotropic texture filtering result (Figure 9f).

However, if antialiasing is not used, limiting anisotropic filter sample locations may lead to inaccuracies near the edges of the canvas mesh. Yet, the severity of these inaccuracies depend on the mesh color values. If the mesh color values on the other side of a canvas edge are not substantially different, the inaccuracies would be minimal. Otherwise, the resulting image might contain aliasing artifacts in shading, which would be similar to typical pixelation artifacts that appear with no antialiasing (Figure 9e). Nonetheless, if anisotropic filter sample locations are not limited (as in current GPUs), the inaccuracies would likely be more severe (Figure 9d).

3.6 Nonuniform Mesh Colors

One limitation of the original mesh colors method is that a single resolution value is used for determining the resolution of each face. As a result, elongated triangles (with two long edges and one short edge) may occupy more color samples than necessary (along the

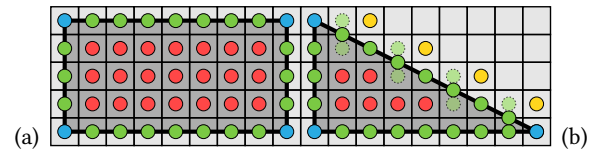


Figure 12: Texture layout for converting nonuniform mesh colors to a 2D texture.

short edge). While this does not lead to any problems in filtering, it causes needless increase in color storage.

It is trivial to address this problem for elongated quadrilaterals by simply defining a secondary resolution value per face, as shown in Figure 12a. In some cases, pairs of elongated triangles can be represented as elongated quadrilaterals in the canvas mesh, but this is not always possible. Fortunately, we can apply the same solution to elongated triangles by introducing a secondary resolution. In this case, the primary resolution would determine the resolution of the two long edges and the secondary resolution is for the shorter edge. The face color locations are determined by two of the edges (with different resolutions), similar to elongated quadrilaterals, as shown in Figure 12b.

While converting such triangles to mesh color textures, the texel pairs that correspond to the diagonal edge colors must be computed via interpolation (Figure 12b). In this case, if the interpolated colors are negative or greater than 1, we can simply modify the corresponding diagonal edge color.

Note that since we restrict the resolutions to powers of 2 (minus one), there is no need to consider three different resolutions for the three edges of a triangle. The only cases that would not collapse a triangle (in texture space) would have the same resolution along two of the edges and an equal or a lower resolution along the third edge (i.e. an elongated isosceles triangle).

4 RESULTS

We tested the performance of mesh color textures in comparison to standard 2D textures on multiple GPUs spanning multiple generations using different models and different texture and screen resolutions up to 4K. Some of the models we used in our tests are shown in Figure 1.

Our experiments revealed that mesh color textures with mipmap filtering using the 4D texture coordinates introduce a negligible overhead as compared to standard 2D textures. Figure 13 summarizes our experimental results. Each one of our experiments contained a single model with different resolutions rotating in front of the camera at varying distances using different resolutions of mesh color textures and standard 2D textures with matching resolutions, rendered using different viewport resolutions. In these experiments a single texture lookup was performed per fragment in an extremely simple fragment shader with no other operations. Thus, the experiments directly measure the difference in texture lookup performance for mesh color textures and standard 2D textures.

On relatively new GPUs we observed an average overhead less than 0.01 milliseconds per frame. In fact, for some camera angles, mesh color textures provided even faster render times than com-

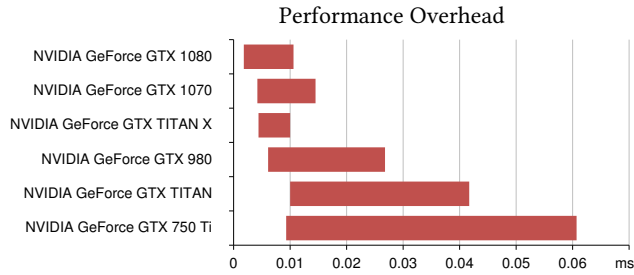


Figure 13: Performance overhead of mesh color textures (in milliseconds) as compared to standard 2D textures, showing the range of overhead values we observed with different experiments (minimum to maximum) measured on different graphics hardware.





pletely hardware-accelerated standard 2D textures. While we could observe a more pronounced average overhead on older GPUs, it is still negligible, especially considering the benefits of mesh colors over 2D textures for both authoring (without parameterization) and filtering quality (without seam artifacts).

It is not surprising that the mesh color textures produce a similar performance as 2D textures. Even though our trilinear filtering explicitly samples two separate textures, trilinear filtering with standard 2D textures also involves sampling two separate mipmap levels of a texture. Therefore, we attribute the minor overhead we observed to the few extra operations we need to perform in software and the difference in the texture coordinates that could impact cache behavior. The difference in cache behavior is also an explanation for some camera angles that happen to provide slightly faster results with mesh color textures.

In terms of quality, since we generated our mesh color textures directly from previously painted 2D textures, the rendered results with mesh color textures and standard 2D textures are virtually indistinguishable. However, since standard 2D textures contain seams, when using standard 2D textures with mipmapping, the seams are revealed as seam artifacts even with the first few mipmap levels in our examples. The results with mesh color textures, on the other hand, contain no seam artifacts. This highlights why it is important to use mesh color textures at run time, rather than converting 3D painted mesh colors to standard 2D textures for real-time rendering.

The layout generation method we used (Algorithm 1) can produce tightly packed rectangles with limited wasted space for the highest resolution mipmap level. At higher mipmap levels with lower resolution, however, a larger percentage of texels are wasted due to imperfect packing (see Figure 8). Yet, since lower resolution mipmap levels have fewer pixels, the total percentage of the texture data wasted with unused texels is comparable to typical uses of standard 2D textures, as shown in Table 1. However, this analysis does not include duplicated vertex and edge colors for mesh color textures or the duplicated colors along seams for standard 2D textures. It also does not consider wasted space due to mapping distortion. On the other hand, since the width and height of a mipmap level with mesh color textures are typically larger than half of the preceding level, mesh color textures occupy more memory for storing the mipmap levels, as compared to a standard

Table 1: Wasted texture space due to packing

				
	Alien	Lizard	Head	Nyra
Standard 2D Texture	26%	39%	19%	17%
Mesh Color Texture	15%	15%	15%	16%
Mipmap level $\ell = 0$	12%	8%	7%	8%
Mipmap level $\ell = 1$	18%	19%	17%	18%
Mipmap level $\ell = 2$	28%	33%	31%	33%
Mipmap level $\ell = 3$	35%	50%	48%	50%
Mipmap level $\ell = 4$	42%	66%	63%	62%
Mipmap level $\ell = 5$	48%	76%	71%	68%
Mipmap level $\ell = 6$	48%	78%	66%	67%
Mipmap level $\ell = 7$	45%	78%	62%	63%
Mipmap level $\ell = 8$	46%	78%	61%	61%
Mipmap level $\ell = 9$	48%	78%	62%	59%

2D texture with the same highest resolution. For the examples in Figure 1 this storage overhead varies between 8% and 15% with 10 mipmap levels.

5 LIMITATIONS AND FUTURE WORK

One disadvantage of using mesh color textures is that trilinear filtering adds some extra lines of code to the shader, in comparison to standard 2D textures that can be filtered using a single API call. Even though this may lead to negligible performance penalty, considering that shader complexity can be an important concern in practice, it would be helpful to hide this complexity within the graphics API. In particular, if future graphics API would allow specifying custom resolutions for mipmap levels, all mipmap levels of the mesh color textures can be handled within a single texture resource. Furthermore, a new texture sampling function that directly uses our 4D coordinates would hide the extra complexity of mesh color textures from the programmer, and parts of the computation can be accelerated to eliminate the overhead using an optimized low-level implementation of our method.

Resolving the limitations regarding anisotropic filtering, however, would likely require hardware modification. We have described a relatively simple modification that could prevent the GPU from sampling invalid texture locations. Such a feature would also resolve the anisotropic filtering problems of standard texture mapping. However, since the seams with standard texture mapping also induce other types of texture filtering artifacts, resolving the anisotropic filtering problems alone would not be an important improvement for standard texture mapping.

Another problem arises with multi-sample anti-aliasing (MSAA). Since MSAA avoids shading each sub-pixel sample and instead calls the fragment shader using the extrapolated (or interpolated) values at the pixel center, the extrapolated 4D texture coordinate can easily fall outside of the texture layout footprint of a face. Similar to anisotropic filtering, this reveals the edges of the canvas mesh. Unlike anisotropic filtering, however, in this case we have direct control over the texture sample location. Therefore, this can be

easily fixed by performing a corrected barycentric interpolation for computing the texture coordinate that avoids extrapolation. The software anisotropic filtering example using MSAA in Figure 9f is generated using this approach. Alternatively, this issue can be easily avoided using *centroid* sampling, which ensures that the 4D texture coordinate is not extrapolated.

Mesh color textures assign multiple texture coordinates per canvas mesh vertex. While traditional 2D textures also assign multiple texture coordinates for some vertices, mesh color textures would typically require more storage for texture coordinates.

Another possible future hardware modification could be triangle tessellation using the tessellation pattern of mesh colors. This way, displacement mapping with tessellation can be done without the need for resampling the colors.

Mesh color textures also inherit some limitations of mesh colors. Since mesh colors rely on the topology of the canvas mesh, arbitrary remeshing would require resampling the mesh colors. Even though mesh color textures convert the mesh color data into 2D textures and thereby provide the option of using existing 2D image processing tools, 3D painting is still preferable with mesh colors.

Finally, since converting mesh colors to a mesh color texture may require modifying a small percentage of mesh color values, not all of the original mesh colors may be exactly reproduced from the mesh color texture.

6 CONCLUSION

We have shown how mesh colors can be converted to a format that is ideal for current GPUs, making mesh colors a viable choice for real-time graphics applications with strict performance constraints. Thus, the content creation pipeline for real-time graphics applications can begin enjoying the benefits of mesh colors, which are no longer exclusive to offline graphics applications. Furthermore, since mesh color textures resolve the filtering problems of texture mapping, mipmaps can be used without incorrect filtering concerns and hardware features like tessellation and displacement mapping can be utilized more broadly.

ACKNOWLEDGEMENTS

We thank Christer Sveen for the alien character, Murat Afşar for the lizard model, Lee Perry-Smith for the head model, and Paul Tosca for the Nyra model. We also thank the anonymous reviewers for their valuable feedback and suggestions on improving this paper.

REFERENCES

Noam Aigerman, Roi Poranne, and Yaron Lipman. 2014. Lifted Bijections for Low Distortion Surface Mappings. *ACM Trans. Graph.* 33, 4, Article 69 (July 2014), 12 pages.

David Benson and Joel Davis. 2002. Octree Textures. *Proc. SIGGRAPH '02, ACM Transactions on Graphics* 21, 3 (2002), 785–790.

Brent Burley and Dylan Lacey. 2008. Ptex: Per-Face Texture Mapping for Production Rendering. In *Eurographics Symposium on Rendering 2008*. 1155–1164.

Nathan A. Carr and John C. Hart. 2002. Meshed Atlases for Real-time Procedural Solid Texturing. *ACM Trans. Graph.* 21, 2 (2002), 106–131.

Nathan A. Carr and John C. Hart. 2004. Painting Detail. *Proc. SIGGRAPH '04, ACM Trans. on Graphics* 23, 3 (2004), 845–852.

Per H. Christensen and Dana Batali. 2004. An Irradiance Atlas for Global Illumination in Complex Production Scenes. In *Proc. of Rendering Techniques (EGSR'04)*. 133–141.

David DeBry, Jonathan Gibbs, Deborah DeLeon Petty, and Nate Robins. 2002. Painting and Rendering Textures on Unparameterized Models. *Proc. SIGGRAPH '02, ACM Trans. on Graphics* 21, 3 (2002), 763–768.

Michael S. Floater and Kai Hormann. 2005. Surface Parameterization: a Tutorial and Survey. In *Advances in multiresolution for geometric modelling*. 157–186.

Francisco González and Gustavo Patow. 2009. Continuity Mapping for Multi-chart Textures. *ACM Trans. Graph.* 28, 5, Article 109 (Dec. 2009), 8 pages.

K. Hormann and G. Greiner. 1999. MIPS: An Efficient Global Parameterization Method. In *Curve and Surface Design Conference Proceedings 1999*. 153–162.

Kai Hormann, Bruno Lévy, and Alla Sheffer. 2007. Mesh Parameterization: Theory and Practice Video Files Associated with This Course Are Available from the Citation Page. In *ACM SIGGRAPH 2007 Courses (SIGGRAPH '07)*. Article 1.

Joe Kniss, Aaron Lefohn, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. 2005. Octree Textures on Graphics Hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*. 16.

Thibault Lambert. 2015. From 2D to 3D Painting with Mesh Colors. In *ACM SIGGRAPH 2015 Talks (SIGGRAPH '15)*. Article 72, 1 pages.

Sylvain Lefebvre and Carsten Dachsbacher. 2007. TileTrees. In *ISD '07*. 25–31.

Sylvain Lefebvre and Hugues Hoppe. 2006. Perfect Spatial Hashing. *Proc. SIGGRAPH '06, ACM Trans. on Graphics* 25, 3 (2006), 579–588.

Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. 2005. Texture Sprites: Texture Elements Splatted on Surfaces. In *ISD '05: Proc. Symposium on Interactive 3D Graphics and Games*. 163–170.

Aaron E. Lefohn, Shubhabrata Sengupta, Joe Kniss, Robert Strzodka, and John D. Owens. 2006. Glift: Generic, Efficient, Random-access GPU Data Structures. *ACM Trans. on Graphics* 25, 1 (2006), 60–99.

Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérôme Maillot. 2002. Least Squares Conformal Maps for Automatic Texture Atlas Generation. *Proc. SIGGRAPH '02, ACM TOG* 21, 3 (2002), 362–371.

Yaron Lipman. 2012. Bounded Distortion Mapping Spaces for Triangular Meshes. *ACM Trans. Graph.* 31, 4, Article 108 (July 2012), 13 pages.

Martin Madaras and Roman Đurikovič. 2012. Skeleton Texture Mapping. In *Proceedings of the 28th Spring Conference on Computer Graphics (SCCG '12)*. 121–127.

Josiah Manson and Scott Schaefer. 2012. Parameterization-Aware MIP-Mapping. *Comput. Graph. Forum* 31, 4 (June 2012), 1455–1463.

Matthias Nießner and Charles Loop. 2013. Analytic Displacement Mapping Using Hardware Tessellation. *ACM Trans. Graph.* 32, 3, Article 26 (July 2013), 9 pages.

Roi Poranne and Yaron Lipman. 2014. Provably Good Planar Mappings. *ACM Trans. Graph.* 33, 4, Article 76 (July 2014), 11 pages.

Budirijanto Purnomo, Jonathan D. Cohen, and Subodh Kumar. 2004. Seamless Texture Atlases. In *SGP '04: Proc. Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*. 65–74.

Nicolas Ray, Vincent Nivolières, Sylvain Lefebvre, and Bruno Lévy. 2010. Invisible Seams. *Computer Graphics Forum* (2010).

Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe. 2001. Texture Mapping Progressive Meshes. In *Proceedings of SIGGRAPH '01*. 409–416.

Henry Schäfer, Magdalena Prus, Quirin Meyer, Jochen Süßmuth, and Marc Stamminger. 2012. Multiresolution Attributes for Tessellated Meshes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (ISD '12)*. 175–182.

Christian Schüller, Ladislav Kavan, Daniele Panozzo, and Olga Sorkine-Hornung. 2013. Locally Injective Mappings. *Computer Graphics Forum (proceedings of Symposium on Geometry Processing)* 32, 5 (2013).

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugarman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: A Many-core x86 Architecture for Visual Computing. *ACM Trans. Graph. (Proc. of SIGGRAPH 2008)* 27, 3, Article 18 (Aug. 2008), 15 pages.

Alla Sheffer, Emil Praun, and Kenneth Rose. 2006. Mesh Parameterization Methods and Their Applications. *Foundations and Trends in Computer Graphics and Vision* 2, 2 (2006), 105–171.

Jason Smith and Scott Schaefer. 2015. Bijective Parameterization with Free Boundaries. *ACM Trans. Graph.* 34, 4, Article 70 (July 2015), 9 pages.

Olga Sorkine, Daniel Cohen-Or, Rony Goldenthal, and Dani Lischinski. 2002. Bounded-distortion Piecewise Mesh Parameterization. In *Proceedings of the Conference on Visualization '02 (VIS '02)*. 355–362.

Marco Tarini. 2016. Volume-encoded UV-maps. (2016).

Marco Tarini, Kai Hormann, Paolo Cignoni, and Claudio Montani. 2004. PolyCube-Maps. *Proc. SIGGRAPH '04, ACM Trans. Graph. (TOG)* 23, 3 (2004), 853–860.

Natalya Tatarchuk. 2016. Open Problems in Real-time Rendering. In *ACM SIGGRAPH 2016 Courses (SIGGRAPH '16)*. Article 20.

Natalya Tatarchuk, Aaron Lefohn, and Peter-Pike Sloan. 2015. Frontiers in Real Time Rendering. In *ACM SIGGRAPH 2015 Courses (SIGGRAPH '15)*. Article 13.

Cem Yuksel, John Keyser, and Donald H. House. 2010. Mesh colors. *ACM Transactions on Graphics* 29, 2, Article 15 (2010), 11 pages.

Eugene Zhang, Konstantin Mischaikow, and Greg Turk. 2005. Feature-based surface parameterization and texture mapping. *ACM Trans. Graph.* 24, 1 (2005), 1–27.

Kun Zhou, John Snyder, Baining Guo, and Heung-Yeung Shum. 2004. Iso-charts: Stretch-driven Mesh Parameterization Using Spectral Analysis. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing (SGP '04)*. 45–54.