Mesh Colors with Hardware Texture Filtering

Cem Yuksel[*]
University of Utah

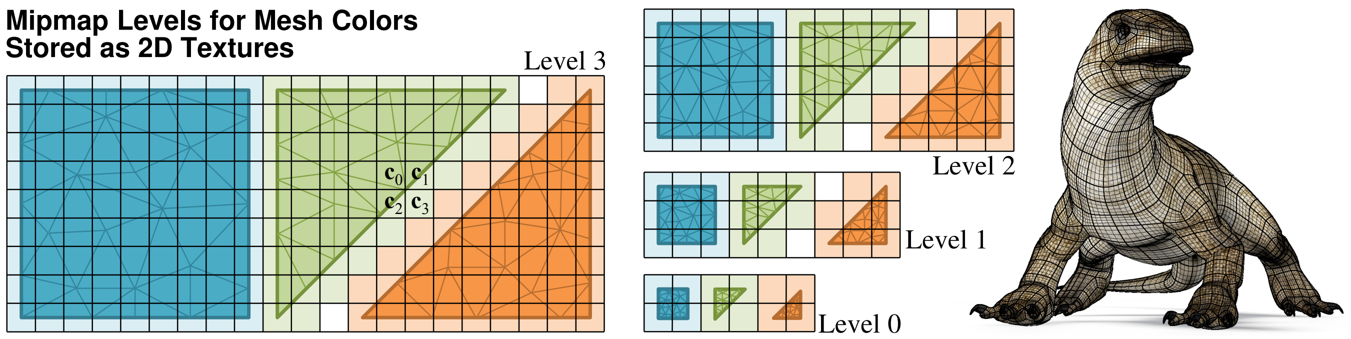

Figure 1: *Mesh colors converted to 2D textures at different mipmap levels, and mesh colors applied on a lower-resolution tessellation of a polygonal model (prepared by Murat Afşar).*

Abstract

Texture mapping has fundamental limitations that cause important practical problems, such as the amount of manual labor needed for specifying the mapping and the visual artifacts in texture filtering that appear near seams. The mesh colors method [Yuksel et al. 2010] was proposed as an alternative to texture mapping that resolves these problems. However, mesh color filtering has been slow for real-time rendering, simply because mesh colors cannot directly use the existing texture filtering hardware on today's GPUs. We introduce a method that converts mesh colors into a format that is similar to 2D textures (Figure 1), thereby allowing the use of existing texture filtering hardware on the GPU for mesh color filtering. We also discuss potential future modifications to the texture filtering hardware for fully supporting mesh color filtering.

Keywords: Mesh colors, texture mapping, texture filtering

Concepts: •**Computing methodologies** → *Texturing;*

Limitations of Texture Mapping

Texture mapping works well when a small texture is tiled over the surface of a model. However, 2D textures are often used differently by splitting a model into multiple pieces, planarizing these pieces, and distributing them over the texture space, such as the example in Figure 2. While there exists numerous automatic methods for generating a texture mapping layout, in practice this process involves a substantial amount of manual labor. Furthermore, modifications to the model may require regenerating this layout; therefore, texture paint is typically performed subsequent to modeling.

Texture mapping by splitting the model into pieces also causes problems in texture filtering near seams. It is often impossible to make sure that the colors on either side of a seam would match perfectly (and that the seam would be completely invisible), because the colors are computed at different parts of the texture map. In practice, it is possible to hide the seams by carefully painting around them (the blue region in Figure 2), but they still show up in mipmap levels. In fact, after only a few levels of mipmap filtering, the colors near seams can be influenced by unrelated parts of the model, depending on their proximity in the texture space. Therefore, mipmap filtering produces incorrect results and the number of mipmap levels that can be used without visible filtering artifacts near seams is limited. The inaccuracies in texture filtering are especially important for displacement mapping, since they can lead to visible cracks along the seams.

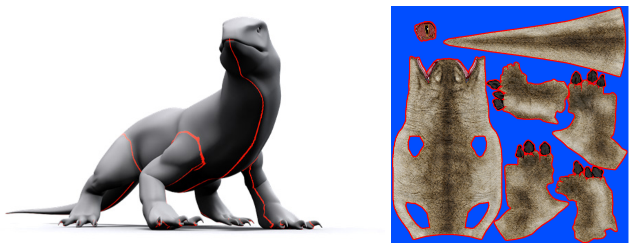

Figure 2: *An example character model and its 2D texture layout. The red curves on the model and the texture indicate the seams.*

These are fundamental problems of texture mapping that are caused by the fact that the model and the texture are defined in different spaces and that a general model cannot be mapped onto the texture space without introducing seams or substantial deformation.

Mesh Colors

Mesh colors [Yuksel et al. 2010] eliminate the fundamental problems of texture mapping by defining the colors directly on the model surface. Thus, mesh colors exist in the same space as the model surface. Using the existing topology of a polygonal mesh, mesh colors allow generating detailed textures on arbitrary polygonal meshes without having to specify mapping coordinates. By avoiding the need for a mapping between the model and the texture spaces, mesh colors eliminate the problems caused by mapping.

[*]e-mail: cem@cemyuksel.com

SIGGRAPH '16, July 24-28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4282-7/16/07
DOI: http://dx.doi.org/10.1145/2897839.2927446

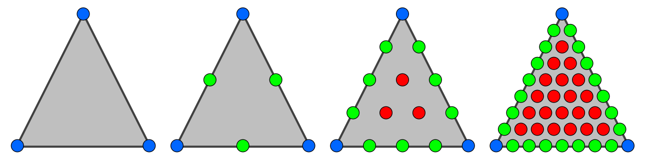

Figure 3: *Mesh colors on triangles with different resolutions: (blue) vertex colors, (green) edge colors, and (red) face colors.*

Mesh colors can be considered as an extension of vertex colors. Indeed, the lowest resolution mesh colors on a face only includes vertex colors. Higher resolution mesh colors include colors exactly on the edges (edge colors) and inside the face (face colors), as shown in Figure 3. The colors are spaced evenly in the barycentric space on a triangle or the bilinear space on a quadrilateral. The resolution of each face can be specified independently.

By placing colors exactly along the edges, mesh colors avoid filtering artifacts of seams. The color value anywhere on a face can be computed by using the vertex, edge, and face colors of the face, without considering the neighboring faces. Moreover, mipmap levels generated from mesh colors are guaranteed to be correct and they allow pre-filtering all the way down to vertex colors. Perhaps most importantly, mesh colors eliminate the labor of specifying mapping coordinates by directly using the polygonal mesh as a canvas for 3D painting.

On the other hand, the existing texture filtering units we have on commercial GPUs are not designed to perform the texture filtering operations of mesh colors. While it is possible to implement mesh color filtering using GPU shaders, this introduces substantial amount of additional code complexity. More importantly, it is known that software texture filtering operations can be up to an order of magnitude slower than hardware implementations.

Mesh Color Filtering on Hardware

Since the existing texture filtering hardware is designed for 2D (as well as 1D and 3D) textures, we convert the mesh color data into 2D textures. This is extremely easy to do by defining texture coordinates such that vertices are mapped to the centers of some texels, as shown in Figure 1. This way, we can achieve bilinear texture filtering on hardware by copying mesh colors onto a 2D texture, where vertex and edge colors are duplicated as needed. The number of duplicated colors can be minimized by defining mesh colors using a lower-resolution version of the mesh than the tessellated mesh used for rendering, as explained in Yuksel et al. [2010].

The difficulty, however, is trilinear filtering using mipmap levels. A naïve mapping, as explained above, would require specifying different texture coordinates for each mipmap level, which is infeasible. We solve this problem by converting non-normalized 2D texture coordinates $\mathbf{u} \in [0, S_{0,1}]^2$, where S_0 and S_1 are the texture resolutions (number of pixels) in each dimension, using a 4D texture coordinate that contains two 2D components. Let ℓ be the mipmap level, such that $\ell = 0$ is the lowest resolution mipmap level. The texture coordinate for level ℓ is represented as $\mathbf{u}_\ell = 2^\ell \mathbf{u}_0 + \mathbf{u}_\delta$, where $\mathbf{u}_0$ is the 2D texture coordinate for level $\ell = 0$ and $\mathbf{u}_\delta$ is a constant offset. This way, we only need to store a 4D texture coordinate ($\mathbf{u}_0$ and $\mathbf{u}_\delta$) per vertex. For trilinear filtering, we perform two hardware accelerated bilinear filtering operations using $\mathbf{u}_\ell$ and $\mathbf{u}_{\ell+1}$, and linearly interpolate the result.

While generating the 2D textures, care must be taken for triangle edges that are placed diagonally on the texture map. Consider the four texels labeled $\mathbf{c}_0$ through $\mathbf{c}_4$ in Figure 1. The correct linear filtering along the diagonal edge must only depend on the edge colors $\mathbf{c}_1$ and $\mathbf{c}_2$. However, bilinear filtering near the diagonal edge also uses a texel inside the triangle c_0, and a texel outside of the triangle $\mathbf{c}_3$. Since $\mathbf{c}_3$ is outside of the triangle, it does not correspond to a vertex, edge, or face color, and it is merely used for assisting bilinear filtering near the diagonal edge. Therefore, we can set it as $\mathbf{c}_3 = \mathbf{c}_1 + \mathbf{c}_2 - \mathbf{c}_0$ to make sure that the bilinearly interpolated color along the edge would be a function of $\mathbf{c}_1$ and $\mathbf{c}_2$. Thus, if this diagonal edge is placed horizontally or vertically for another face, the filtered color result along the edge would be consistent and it would avoiding visible seams along edges. One limitation of this formulation is that when using unsigned color channels, we must enforce $\mathbf{c}_0 \leq \mathbf{c}_1 + \mathbf{c}_2$.

Table 1: *Performance of Mesh Color Filtering on Hardware*

	Bilinear Filtering	**Trilinear Filtering**	**Anisotropic Filtering**
Number of Textures	1	Number of Mipmap Levels	Number of Mipmap Levels
Texture Lookups	1	2	2
Software Computation	none	Mipmap level, UV coordinate, and lerp	Mipmap level, UV coordinate, and lerp
Filtering Quality	exact	exact	Possible seams near edges

Limitations and Hardware Extensions

Table 1 shows the performance of mesh color filtering on current GPUs. Bilinear filtering is as simple as a regular 2D texture lookup. Trilinear filtering with mipmaps, however, requires storing each mipmap level using a different texture, because the resolution of our mipmap level ℓ is more than half of the resolution of mipmap level $\ell + 1$ in each dimension, which is not supported by current GPUs. The additional texture lookup does not introduce performance loss as compared to regular 2D texture lookup with trilinear filtering, but the desired mipmap levels and the corresponding texture coordinates must be computed in software. The implementation of anisotropic filtering (with mipmaps) is identical to trilinear filtering. However, anisotropic filtering across seams can produce incorrect results, because current GPUs automatically determine the sampling positions for anisotropic filtering and those samples may fall outside of the texture area of the shaded face. For producing correct filtering results, the texture filtering hardware must avoid barycentric extrapolation of the texture coordinates and only sample the part of the texture that corresponds to the shaded triangle.

For handling trilinear filtering with a single texture lookup, two relatively minor hardware modifications are necessary. The first one is the ability to specify a custom texture resolution for each mipmap level, and the second one is a 2D texture lookup operation using 4D coordinates to compute a different 2D texture coordinate per mipmap level, as explained above. Hardware extensions for these two relatively minor features would permit hardware accelerated trilinear filtering of mesh colors with only a single texture lookup, which would make the implementation of mesh color filtering on hardware even simpler (as simple as regular 2D textures).

References

YUKSEL, C., KEYSER, J., AND HOUSE, D. H. 2010. Mesh colors. *ACM Transactions on Graphics 29*, 2, 15:1–15:11.