Abstract

We investigate the mesh colors method for per-face parameterization for texture-mapping of geometry, implemented in the game engine Frostbite 3, for the purpose of evaluating the method compared to traditional texture-mapping in a real-time application. Traditional UV-mapping often causes discontinuities which commonly results in visible seams in the end results. If any change is done to the vertex positions or the topology a remapping of the UV-map has to be done. Mesh colors aims to avoid these problems by skipping the transformation to 2D space as in UV-mapping, and associating color samples directly with the geometry of a mesh. The implementation was done in Frostbite 3 in C++ and HLSL shader code, and rendered with the DirectX graphics API. The results show that mesh colors is a viable alternative in a real-time renderer. Though not as fast as regular UV-mapped textures due to lack of hardware accelerated filtering operations, mesh colors is a realistic alternative for special cases where regular texture-mapping would be cumbersome to work with or produce sub-par results. Possible areas of future research are efficient data structures suitable to handle data insertion dynamically, compression of mesh colors data, and mesh colors in the context of mesh LOD generation.

Förord

Jag vill tacka DICE för förslaget om examensarbete, ett speciellt tack till Torbjörn Söderman som gjorde examensarbetet möjligt till att börja med och Andreas Tarandi vars arbete lade grunden för det här projektet. Till Johan Åkesson, Charles de Rousiers, Sebastien Hillaire, och Jan Schmid på Frostbite för deras hjälp och tålamod med mina frågor och funderingar. Till Mark E. Dieckmann och Stefan Gustavson vid LiU. Tack till alla på *Mirror's Edge* på DICE för att ha tagit med mig som en del av teamet. Det har varit en otrolig upplevelse och väldigt lärorikt. Ett särskilt tack till min familj och mina vänner för erat stöd och er uppmuntran.

Preface

I would like to thank DICE for presenting the thesis proposal, a special thanks to Torbjörn Söderman for making this thesis possible to begin with and Andreas Tarandi whose work laid the foundation for this project. To Johan Åkesson, Charles de Rousiers, Sebastien Hillaire, and Jan Schmid at Frostbite for their help and their patience with all of my questions. To Mark E. Dieckmann and Stefan Gustavson at LiU. Thanks to everyone on the *Mirror's Edge* team at DICE for making me a part of your team. It has been a great experience and I learned so much. A special thanks to my family and friends for their support and encouragement.

Contents

List of Figures

List of Tables

1 — Introduction

This report starts with an introduction going over some basic concepts and techniques relevant to this report. A problem description is presented as well as a short introduction to the mesh colors approach, and finally we review related work. In the background section we go into more detail in describing the problems with the current common method for texturing geometry and provide some more detail concerning mesh colors. The implementation section details the implementation done in the game engine Frostbite 3. Finally results are presented, and we discuss our findings and possible future work.

This report assumes that the reader has a basic understanding of 3D computer graphics and is familiar with the graphics pipeline.

All models and textures in this report are courtesy of DICE.

1.1 Texture Mapping & UV-coordinates

Texturing of meshes has been the industry standard to explicitly add the illusion of geometric detail where there is none. The currently most widely used method in the games industry for mapping a texture onto a mesh is to explicitly parameterize the 3D geometry to 2D space and then paint the texture onto this 2D representation. The painted texture is then projected back onto the mesh. This 2D representation of the mesh is commonly referred to as a UV-map, where U and V represent the two axes of the 2D representation, and the technique as UV-mapping. UV-coordinates are assigned per-vertex and are usually interpolated in the pixel shader to obtain per-pixel UV-coordinates. When creating a UV-map decisions have to be made concerning texture resolution distribution on the mesh, since the texture resolution for a given triangle on the mesh corresponds to that triangle's alloted area coverage in the UV-map. This allows for the artist to prioritize and e.g. assign important faces a higher resolution. Since UV-mapping has been around for so long there is established support for compressing the texture data, as well as hardware accelerated filtering on modern graphics hardware. MIP-maps [9] can be generated automatically by the commonly used graphics API:s such as DirectX and uses hardware filtering since the MIP-map resides in video memory [1].

1.2 MIP-mapping

MIP-mapping is a technique used to lessen the filtering operations needed if a textured mesh occupies a smaller screen-space area than that of the texture's resolution. For a mesh with a given texture resolution copies of that texture is created filtered to lower resolutions. Since common practice dictates that a texture's orignal resolution is a power of 2 it follows that the MIP-maps are filtered to resolutions of decreasing powers of 2. An original texture of resolution 512x512 pixels might have corresponding MIP-maps of resolutions 256x256, 128x128, and so on all the way down to a single pixel. By convention we refer to the orignal texture as being of MIP-level 0 and the following MIP-maps of decreasing resolution are of increasing MIP-level.

1.3 Tessellation

The hardware tessellator is a relatively new addition to the programmable shader stages in the graphics pipeline. Hardware tessellation allows for geometric detail to be modified dynamically in runtime, allowing for algorithms to be run on a coarse mesh with geometric detail added later by the tessellator, thus reducing the load on the memory bus to the GPU. A few applications of tessellation is displacement mapping and subdivision surfaces. The tessellation pipeline consists of three stages, two of which are programmable. The three stages are the hull shader stage (programmable), the tessellation stage (fixed), and the domain shader stage (programmable). The patch constant function in the hull shader provides a per-primitive context, which allows us to access e.g. primitive index of the primitive currently being processed by the hull shader.

For the implementation of mesh colors in this report the tessellation stage is used because the hull shader gives this per-primitve context in its patch constant function, no actual tessellation needs to be performed for mesh colors. For more detailed information about the tessellation stage we refer the reader to [6].

1.4 Problem Description

The process of creating a UV-map is tedious, even if there exists automatic mapping tools they often require some form of manual input from the artist and it takes quite an effort and a skilled artist to produce a “good” UV-map. The transformation from 3D space to 2D space often introduces discontinuities since an arbitrary mesh can rarely be flattened while keeping the topology intact. This can result in visible seams in the final texture which results in a poor visual result. If we want to change the resolution distribution at a later time the UV-map would have to be modified, causing already painted textures to need modification as well. Also, if any change is done to the mesh's vertex positions or the topology the UV-mapping would have to be redone in the same fashion. These issues often cause considerable time delays in production projects since a given mesh typically passes through different departments on it's way to becoming a finished asset (e.g. concept $\rightarrow$ modeling $\rightarrow$ UV-mapping $\rightarrow$ texturing $\rightarrow$ animation). Because of the issues mentioned above there has been a lot of research in alternatives to explicit UV-mapping. One of these alternative methods is mesh colors.

1.5 Mesh Colors

The method investigated for this thesis is mesh colors and was presented by C. Yuksel et al. [12] in 2010. The idea with mesh colors is to remove the transformation from 3D space to 2D space and back to 3D used in traditional UV mapping, and instead associate color values directly with the geometry of the mesh. This would remove many of the cons of UV mapping mentioned above since no transformation to 2D is needed and no explicit mapping takes place, as well as allowing artists to paint models directly in 3D if implemented as a paint tool.

A one-to-one correspondence is achieved, meaning that color samples do not have to be duplicated to achieved color continuity across edges or vertices. Since we store color indices per-triangle the color resolution can be adjusted locally for arbitrary triangles without any global effect or re-mapping needed. This also means that mesh editing operations (e.g. face deletion or creation, subdivision, changing of vertex positions etc.) does not require re-sampling as it would with UV mapping, but is handled inherently with mesh colors.

For this thesis mesh colors has been implemented in the game engine Frostbite 3. The primary focus for this report has been in comparing mesh colors to traditional UV-mapping in terms of performance in real-time rendering, and also to highlight possible optimizations and improvements for the method to be a realistic alternative to texture-mapping. For this thesis we only consider triangular meshes of type two manifold, although a possible extension to support quadrilateral primitives are briefly covered by Yuksel in [12].

1.6 Related Work

Ptex presented by B. Burley et al. [7] in 2008 is a method for per-face texture-mapping. It was developed for use on Catmull-Clark subdivision surfaces [8] for use in animated feature films. It is similar to mesh colors in that the method does not require explicit UV mapping and it guarantees color continuity across edges by using adjacency data for filtering. Due to the somewhat wasteful storage of texture data along with only supporting quadrilateral primitives makes Ptex, in it's original form, not very well suited for real-time rendering.

There are applications in use that allow for the user to paint directly on 3D geometry, like MARI [5] and Deep Paint 3D [3]. Both applications still require explicit UV mapping of the geometry and essentially only give the illusion that no 2D mapping takes place when painting.

A previous thesis work by A. Tarandi [11] implements mesh colors in a stand-alone project in the context of surface data on dynamic topologies. Tarandi provides details of the implementation that were lacking in the original paper by Yuksel. We expand further on this work by adding support for 1-manifold edges concerning adjacency sampling when generating MIP-map data for these boundary edges, as well as implementing the method in a game engine.

1.7 Frostbite 3

The thesis is implemented in the game engine Frostbite 3. The engine uses deferred rendering which is a rendering technique that separates the shading into several passes [4]. The first pass renders screen-space geometry data (positions, normals, materials etc.) into a geometry buffer. The subsequent passes then perform the actual shading operations such as lighting computaions on the stored geometry buffer from the first pass. This decoupling between the scene geometry and shading allows for rendering many light sources without any significant decrease in performance. The engine is used by studios within Electronic Arts. Some notable titles using Frostbite 3 include *Battlefield 4* developed by Digital Illusions CE and *Need for Speed, Rivals* developed by Ghost Games and Criterion Games.

2 — Background

2.1 Texture Mapping

As mentioned in the problem description the mapping of 3D geometry onto a 2D surface comes with some inherent problems. For complex geometry as is often the case in games these problems become even more apparent, especially when discontinuities causes visible seams which disrupts the immersion for the user. In figure 2.1 the UV-coordinates, detailed in the introduction section, are visualized as color values on the mesh surface with the U and V coordinates represented by the red and green color channels respectively. Discontinuities in the UV map are visible around the ears and from the top of the head down along the neck area.

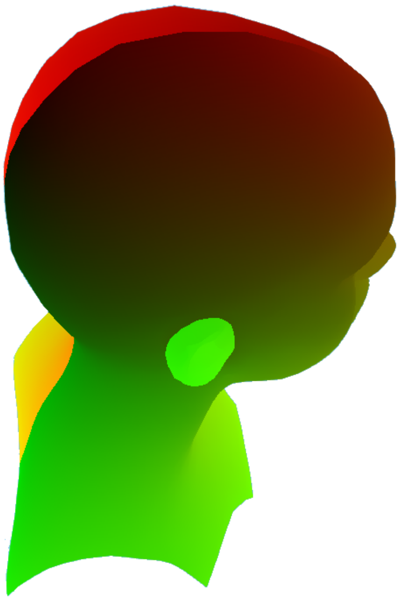


Figure 2.1: Head model with UV-coordinates projected onto it's surface. Visible seams appear at the ears and at the top of the head and down the neck area.

In this instance it would be up to the artist painting the texture to match color values across the discontinuities to avoid visible seams in the end result. This is both time consuming, unintuitive and not guaranteed to produce good results.

2.2 Mesh Colors

Mesh colors is for the most part described well in [12] and [11] and for simplicity's sake we provide a summary of the method here.

2.2.1 Overview

Mesh colors does not require any explicit mapping. Evenly spaced color values are distributed on the vertices, edges, and faces of the mesh using the barycentric coordinates for each face, as seen in figure 2.2.

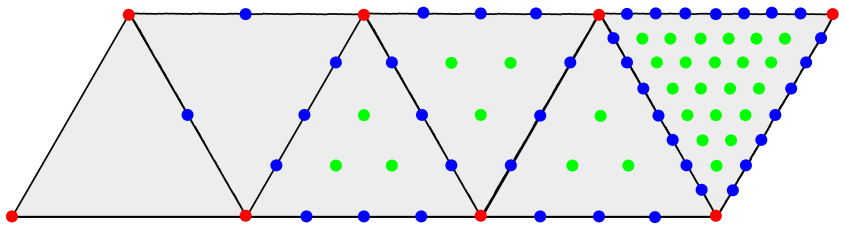


Figure 2.2: Evenly spaced color samples distributed on the vertices (red), edges (blue), and faces (green) of the geometry with increasing resolution from left to right.

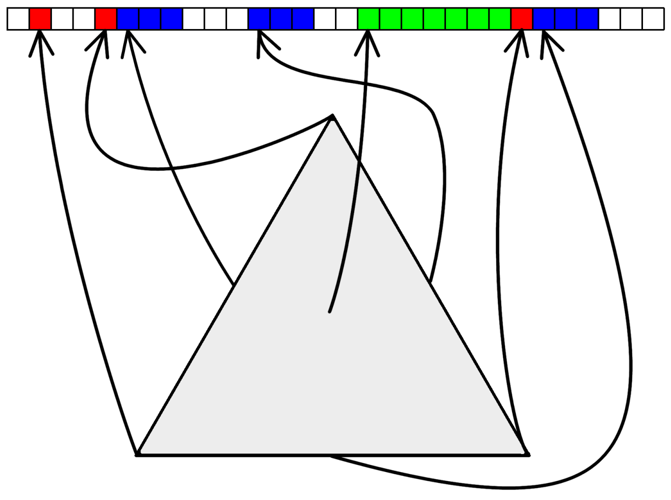


Figure 2.3: Illustration of an example of how the per-face buffer relates to the color buffer memory layout for a face.

Given a resolution R for a face the number of samples distributed for that face are given in eq. 2.1a-c.

$$Vertex = 1 \tag{2.1a}$$

$$Edge = R - 1 \tag{2.1b}$$

$$Face = \frac{(R-1)(R-2)}{2} \tag{2.1c}$$

Eq. 2.1c is defined in [12] but a derivation is lacking and will be provided here. The equation is derived from the formula for calculating the area of a triangle, $bh/2$, where bh is the area of a parallelogram with a base of length b and a height of h. For the discreet sample distribution used in mesh colors we have $b = h = (R-1)$ resulting in $(R-1)^2/2$ number of face samles. But the samples along the line dividing the parallelogram in question would be on the edge of the triangle and must therefore be subtracted when calculating the number of face samples. We adjust the formula accordingly as follows

$$\frac{(R-1)^2}{2} - (R-1) = \frac{(R-1)(R-2)}{2}$$

to arrive at eq. 2.1c.

Using color indices we define a coordinate system for a face as described in [12]. The coordinate system is illustrated in figure 2.4. A color sample C at position ij on the triangle is denoted C_{ij} where $0 < i < R$ and $0 < j < R - i$. In figure 2.4 we see that color samples C_{00}, C_{0R}, and C_{R0} correspond to vertex color samples. Furthermore, color samples C_{k0}, C_{0k}, and $C_{k(R-k)}$ for $0 < k < R$ are edge samples. All other samples are on the face itself.

$$\begin{aligned} C_{00}, C_{0R}, C_{R0}&\text{: Vertex} \\ C_{k0}, C_{0k}, C_{k(R-k)}&\text{: Edge} \\ other&\text{: Face} \end{aligned} \tag{2.2}$$

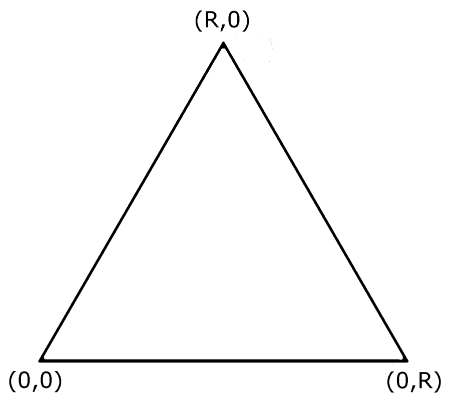


Figure 2.4: Triangle coordinates defined by color indices.

We can retrieve the 3D surface position of any color sample C_{ij} by using the barycentric coordinates and the index of the color sample. The barycentric coordinate $\vec{P}_{ij}$ for the color sample C_{ij} is computed as in eq. 2.3 where $0 < i < R$ and $0 < j < R - i$.

$$\vec{P}_{ij} = (\frac{i}{R}, \frac{j}{R}, 1 - \frac{i+j}{R}) \tag{2.3}$$

When generating the data two buffers are stored. One buffer contains the actual color values and one buffer contains data for each face. The per-face data buffer stores the resolution, the MIP-level, and indices into the color buffer for the face's edges, vertices, and the color samples on the face itself. A vertex color index points to a single color value in the color buffer since a vertex can only have one color regardless of the resolution, but for edges and faces it is a bit more complicated. Since an edge or a face typically have more than one color sample we store all color values for a given edge or face sequentially in memory. A color index for a given edge or face then points to the first color sample for that edge or face in the color buffer. The relationship between the per-face data buffer structure and the color buffer is illustrated in figure 2.3.

2.2.2 Non-uniform Resolutions

Non-uniform face resolutions is one of the main advantages of mesh colors. Since a vertex only has one color value regardless of resolution, and since face color samples are not shared between faces, these cases do not need to be handled explicitly. Edges however can be shared between faces. In this case we choose a resolution so that, from the face's point of view, no discontinuity appears between the edges of different resolutions. To ensure consistent mapping between edges of different resolutions we constrict ourselves to only allow resolutions that are of a power of 2. This will also simplify the process of generating MIP-maps. More details about non-uniform resolutions are given in the implementation section.

2.2.3 Filtering

For 2D filtering we use the barycentric coordinate $\vec{P}$ computed as in eq. 2.3 multiplied by the resolution R. This gives us the indices for the color sample nearest to $\vec{P}$. For linear filtering we need to find the indices of the three nearest color samples and weights to linearly combine them. We do this by extracting, from the multiplication above, the fractional part $\vec{w}$ and the integer part $\vec{B}$ defined as in eq. 2.4 where the brackets is an integer floor operator.

Consider the three sought color samples as a sub-triangle on the face. The fractional part $\vec{w}$ is considered as the barycentric coordinate with respect to the sub-triangle defined by samples at position $((i+1)j, i(j+1), ij)$. The sum of the barycentric coordinates w_x, w_y, and w_z will provide information about where the point $\vec{w}$ is located with respect to the sub-triangle [2]. A sum of 1 means the point is on the triangle, a sum of 0 means the point is on an edge or a vertex of the triangle, and lastly any other sum means that the point lies outside of the triangle.

$$\begin{aligned} \vec{B} &= \left\lfloor R\vec{P} \right\rfloor \\ \vec{w} &= R\vec{P} - \vec{B} \end{aligned} \tag{2.4}$$

As described in [12] there are three cases. If $\vec{w} = 0$ we are on a sample at point $\vec{B}$ and thus that is the color. If $w_x + w_y + w_z = 1$ the point is on the sub-triangle and the nearest color values are $C_{(i+1)j}$, $C_{i(j+1)}$, and C_{ij} and they can be linearly combined using $\vec{w}$ as weight. Lastly if $w_x + w_y + w_z = 2$ the nearest color values are $C_{i(j+1)}$, $C_{(i+1)j}$, and $C_{(i+1)(j+1)}$. These can be linearly combined by $(1,1,1) - \vec{w}$.

For nearest filtering we simply pick the color sample with the highest weight.

2.2.4 MIP-maps

Separate MIP-maps are generated for each face as detailed in [12]. These are created by a weighted sum of color samples from a higher MIP-level. Note that a MIP-level of 0 corresponds to the finest resolution.

The vertex color for MIP-level n is computed using samples from MIP-level $n-1$ from all incident edges to the vertex. In eq. 2.5 v is the valence of the vertex and C_{e_i} is the edge color sample from MIP-level $n-1$ closest to the vertex, and C'_v is the resulting vertex color for MIP-level n. C_v is the vertex color sample from MIP-level $n-1$.

$$C'_v = \frac{2}{2+v}\left(C_v + \frac{1}{2}\left(\sum_{i=1}^{v} C_{e_i}\right)\right) \tag{2.5}$$

A sample for MIP-level n on an edge or a face is computed by weighting samples from MIP level $n-1$ in the immediate surrounding area. For a face sample of level n a total of six values from level $n-1$ are used as in eq. 2.6.

$$C'_{ij} = \frac{1}{4}\left(C_{(2i)(2j)} + \frac{1}{2}\begin{pmatrix} C_{(2i)(2j-1)} & + \\ C_{(2i)(2j+1)} & + \\ C_{(2i-1)(2j)} & + \\ C_{(2i-1)(2j+1)} & + \\ C_{(2i+1)(2j-1)} & + \\ C_{(2i+1)(2j)} & \end{pmatrix}\right) \tag{2.6}$$

For an edge sample of MIP-level n an equation similar to eq. 2.6 is used, though we have to take into account the case of a boundary edge. This is done by adjusting the weighting of the of color samples used from MIP-level n by the number of samples taken.

2.3 Memory usage

A UV-mapped texture seldom uses all available space on the UV-map. This is most often due to the UV-mapping space being square shaped, and the geometry being mapped does not have the topology to allow for fitting and covering the entire mapping space in a usable way. An example of this can be seen in fig. 2.5 where the black areas of the map are not used for holding any color data. Mesh colors does not waste space in this manner since all color samples are directly associated with the geometry. We can therefore achieve visual equivalence with mesh colors even though fewer color samples are used. Since UV-mapping has been the industry standard for some time there has been lots of research done on algorithms for compression the color data. Lossless compression methods such as PNG and lossy compression methods like JPG are widely used and allow for the textures to be stored in disk in a compressed format. Although there is some ongoing reseach in the area, no such standard compression methods exist for mesh colors.

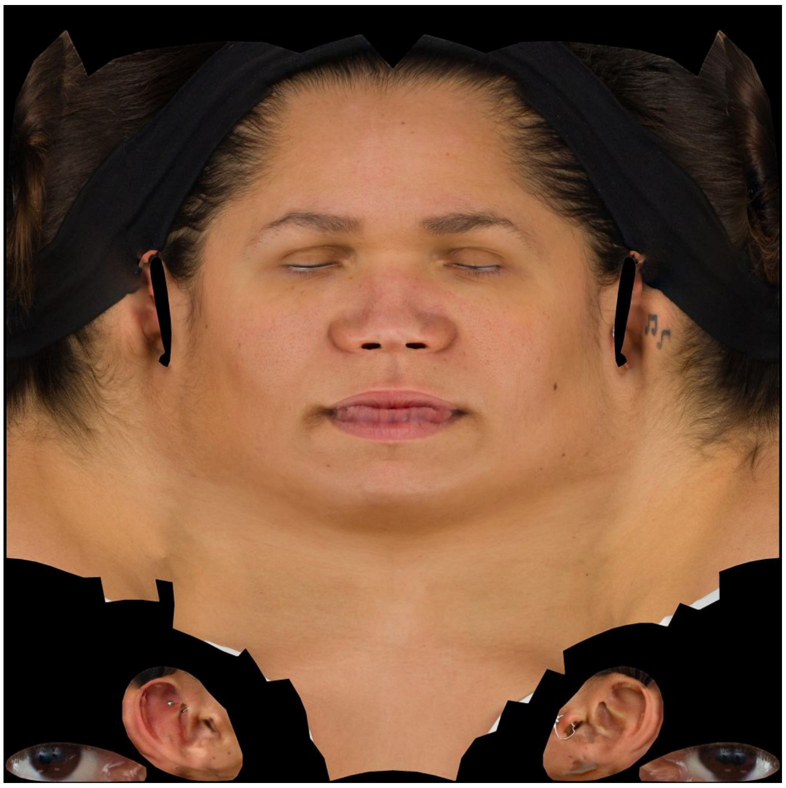


Figure 2.5: UV-map for the head model. An original resolution of 1014x1024 samples.

3 — Implementation

For this thesis mesh colors has been implemented in Frostbite 3. For the sake of showcasing visual results we have used models with existing UV-maps and texture maps to sample when populating the mesh colors buffers [10].

3.1 Data structure and data flow

For storing the face data buffer we define a struct on the CPU (listing 3.1) and a matching struct in the HLSL shaders (listing 3.2) containing resolution, MIP-level, and indices into the color buffer for the face's vertices, edges, and face color samples.

Listing 3.1: CPU data structure

```
typedef Vec4<float> ColorData
struct FaceDataMeshColor
{
  u32 resolution;
  u32 faceColorIndex;
  int32 edgeColorIndices[3];
  u32 vertexColorIndices[3];
  u32 mip;
};
```

Listing 3.2: GPU data structure

```
typedef float4 color_data_t
struct face_data_t
{
  uint face_color_resolution;
  uint face_color_index;
  int3 edge_color_index;
  uint3 vertex_color_index;
  uint mip;
};
```

Color data is simply a linear array of floating points defining RGB colors. We use structured buffers when sending the buffers to the GPU. Because of this the memory layout of the structures must be the same on both the CPU and the GPU.

The implementation is written in C++ and HLSL shader code, and uses the DirectX graphics API for rendering in Frostbite 3. The vertex shader is a pass-through stage and no transformation or projection takes place. In the hull shader we do culling of backward facing patches relative to the camera view direction, and in the patch constant function of the hull shader we fetch the unique patch id by the system semantic *SV_ PrimitiveID* and pass to the domain shader [13]. In the domain shader we use the received patch id to do a lookup in the face data buffer and pass the face resolution, MIP-level, and the color indices for the patch vertices, edges, and face color samples as integers using the *nointerpolation* modifier to the pixel shader. Even though we deal with per-face data we have to send the data per vertex to the pixel shader. We also perform transformation and projection in the domain shader and pass the resulting values needed to the pixel shader.

3.2 Index calculation

Given the coordinate $\vec{B_{ij}}$ from eq. 2.4 on a face we use eq. 2.2 to determine if we are on a vertex, edge or face. For the first case we need to determine which of the face's three vertices to sample with eq. 3.1a. In the second case we need to determine which of the edges we should sample and with what offset. This is determined by eq. 3.1b-c. Eq. 3.1 does not determine indices in a global context, but rather the indices defined locally within the face currently being processed.

For a face sample we need to compute the offset into the face's color buffer for the sample, as in eq. 3.2.

$$V_{index} = \begin{cases} 0 \text{ if } \vec{B_j} = R \\ 1 \text{ if } \vec{B_i} = 0 \text{ and } B_j = 0 \\ 2 \text{ if } \vec{B_i} = R \end{cases} \tag{3.1a}$$

$$E_{index} = \begin{cases} 0 \text{ if } \vec{B_i} = 0 \\ 1 \text{ if } \vec{B_j} = 0 \\ 2 \text{ if } \vec{B_i} \neq 0 \text{ and } B_j \neq 0 \end{cases} \tag{3.1b}$$

$$E_{offset} = \begin{cases} R - B_j \text{ if } E_{index} = 0 \\ B_i \text{ if } E_{index} = 1 \\ B_j \text{ if } E_{index} = 2 \end{cases} \tag{3.1c}$$

The offset for face samples is the sum of all previous indices ij. The sum has initial condition of $k = 1$ and terminating condition of $i - 1$ since the first and last indices belong to the edges of the face.

$$F_{offset} = \sum_{k=1}^{i-1} (R - (k+1)) + (j-1) = \frac{(2R - 2 - i)(i-1)}{2} + (j-1) \tag{3.2}$$

When sampling an edge shared between two neighboring faces the buffer needs to be sampled in reverse. As described in [11] we solve this by denoting the sampling direction with a negative sign on the edge color index and handle these occurrences in the shader code during rendering.

3.3 MIP-mapping

As mentioned in the background section we store separate MIP-maps for each face, and we always create the maximum number of MIP-levels m defined by eq. 3.3 where m is an integer.

$$m = log_2(R) \tag{3.3}$$

MIP-maps are computed for each vertex, edge, and face and stored in a linear fashion sequentially in memory following the previous MIP-map of one level higher. Storing the MIP-data in this fashion allows us to fetch the color index of a given MIP-level by calculating and using an offset from the original color index into the color buffer. In figure 3.1 storage of the MIP-levels for an edge with a resolution of 8 is illustrated. Since a vertex always has exactly one color value regardless of MIP-level, the offset for a vertex is increased by 1 for each MIP-level The calculation of the offsets for edges and faces is the sum of the number of samples from all previous MIP-maps of higher level for that edge or face. Note that by convention a MIP-level of 0 corresponds to the finest resolution.

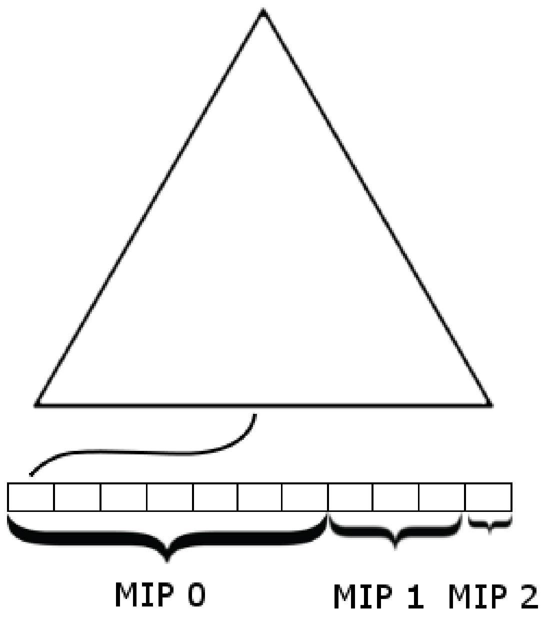


Figure 3.1: Storage of MIP-levels generated for an edge of resolution 8.

From eq. 2.1 we derive the offset calculations in eq. 3.4ab. Since the sums have the forms of geometric progressions as described in [11] we can calculate the offset for a given MIP-level as in eq. 3.4ab.

$$\begin{aligned} E_{mip} &= \sum_{k=0}^{mip-1} (2^{m-mip+1} * 2^k) - mip \\ &= 2^{m+1} - 2^{m-mip+1} - mip \end{aligned} \tag{3.4a}$$

$$\begin{aligned} F_{mip} &= \sum_{k=0}^{mip-1} (2^{2(m-k)-1}) - 3 \sum_{k=0}^{mip-1} (2^{m-k-1}) + mip \\ &= \sum_{k=0}^{mip-1} (2^{2(m-mip)+1} 4^k) - 3 \sum_{k=0}^{mip-1} (2^{m-mip} 2^k) + mip \\ &= \frac{1}{3}(2^{2m+1} - 2^{2(m-mip)+1}) - 3(2^m - 2^{m-mip}) + mip \end{aligned} \tag{3.4b}$$

In eq. 3.4ab the integer m is the maximum number of MIP-levels possible with the current face resolution, defined as in eq. 3.3, and mip refers to the MIP-level for which we seek the offset for. In the implementation we always generate every possible MIP-level, down to the lowest possible resolution which corresponds to vertex colors.

3.4 Non-uniform face resolutions

As described in the previous section vertices only store a single color value regardless of resolution or MIP-level, and faces do not share color data with other faces, therefore these two cases with non-uniform face resolutions do not need any special treatment. Edges can however be shared between faces.

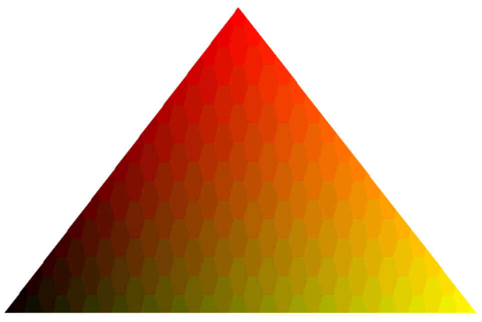
In the case of an edge having neighboring faces of different resolutions we pick the higher resolution for the edge when generating the color data. Since we only allow resolutions that are of power of 2 and since we always create every possible MIP-level as discussed in the previous section, this allows us to pick an appropriate MIP-level of the shared edge when rendering the edge from the perspective of the low resolution face.

4 — Results and Benchmarks

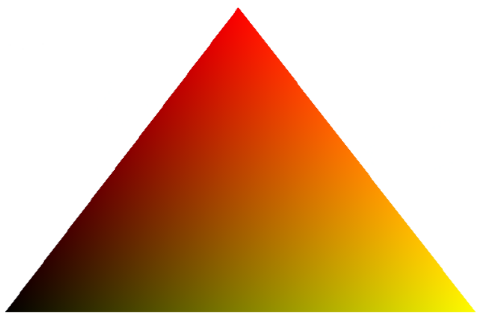
The results presented were run on a desktop computer running Windows 7 64-bit with a Intel X5650 CPU with 6 cores at 2.66 GHz. The graphics cards used separately were an NVidia 670 GTX and an AMD R7950. The color data used to populate the mesh colors color buffer was generated by sampling an existing texture map for the model on the CPU, hence we achieved a visually equivalent result compared to the original texture-mapping. All time measurements were done on a head model consisting of 7473 triangles, originally using a texture map with a resolution of 1024x1024 pixels. Mesh colors renderings and timings were done on the head model with a face resolution of 16 unless otherwise stated. This choice of resolution resulted in an increase of 21% of colors samples used compared to the original texture map. Actually a regular texture map rarely uses all available texture map space so the absolute increase in the number of used color samples is higher than 21% depending in the layout of the UV-map. Renderings were done in the Frostbite 3 engine, and all time measurements were taken from the internal GPU timing tools in Frostbite.

4.1 Filtering

Nearest filtering and linear filtering produces results as expected. Since color samples are effectively placed on the geometry the nearest filtering results in a hexagonal pattern, while the linear filtering results in a blended transition between color samples. The filtering is illustrated in figures 4.1a and 4.1b respectively. Note the clearly visible hexagonal pattern in the result from nearest filtering. Table 4.1 shows render times for mesh colors using nearest- and linear filtering for different resolutions, as well as the regular UV-mapped texture rendered with linear filtering for reference.



(a) Nearest filtering



(b) Linear filtering

Figure 4.1: Triangle illustrating nearest and linear filtering operations respectively

Table 4.1: Render times for mesh colors and regular texture map by filter operation and GPU.

Resolution 16			
Timing (ms)	Nearest	Linear	UV-texture
NVidia 670 GTX	0,14	0,21	0,04
AMD R7950	0,12	0,16	0,04

Resolution 32			
Timing (ms)	Nearest	Linear	UV-texture
NVidia 670 GTX	0,23	0,38	0,04
AMD R7950	0,15	0,17	0,04

4.2 MIP-maps

Timings for different MIP-levels can be seen in figure 4.2. Visual results from MIP-mapping can be seen in figure 4.3, illustrating MIP-levels 0 (native resolution), 2, and 4 rendered with linear filtering.

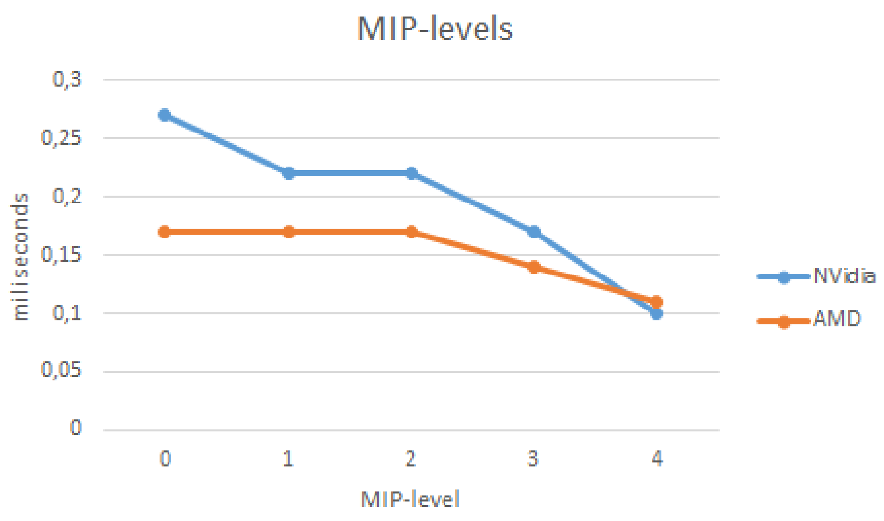


Figure 4.2: Rendering times for different MIP-levels.

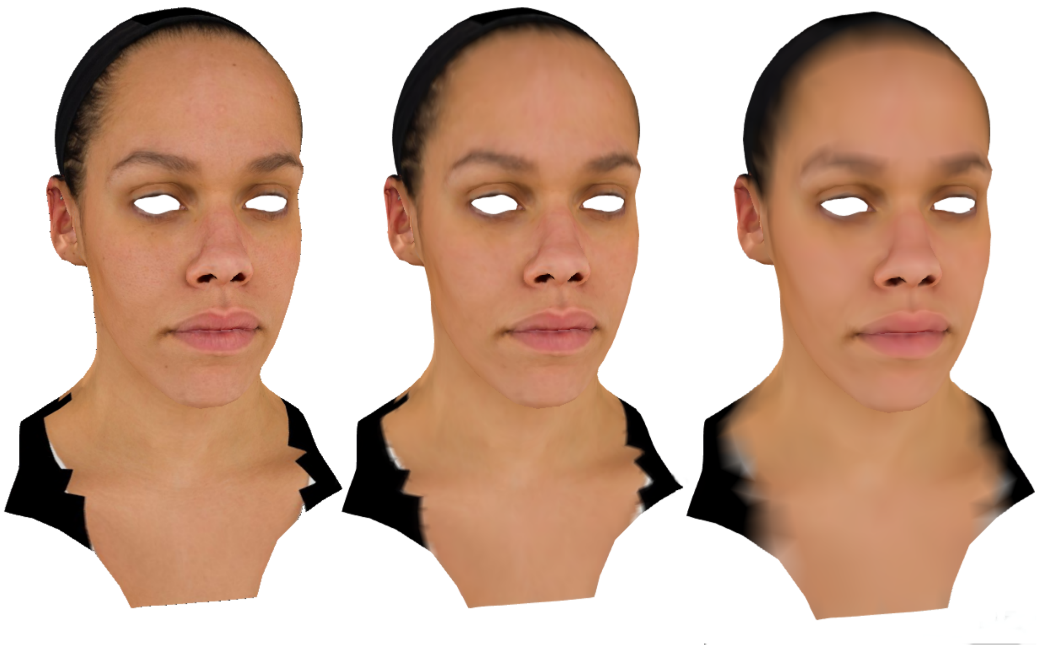


Figure 4.3: From left to right; Native resolution, MIP-level 2 , and MIP-level 4.

4.3 Non-uniform face resolutions

Non-uniform face resolutions is one of the main advantages of mesh colors. Figure 4.4 illustrates a closeup of an example of non-uniform face resolutions rendered with nearest filtering in order to make the difference in resolution more apparent, and figure 4.5 shows rendering timings for different face resolutions.

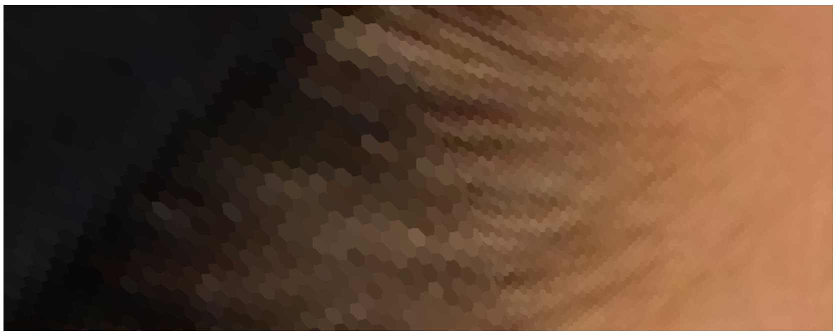


Figure 4.4: Non-uniform face resolutions rendered with nearest filtering.

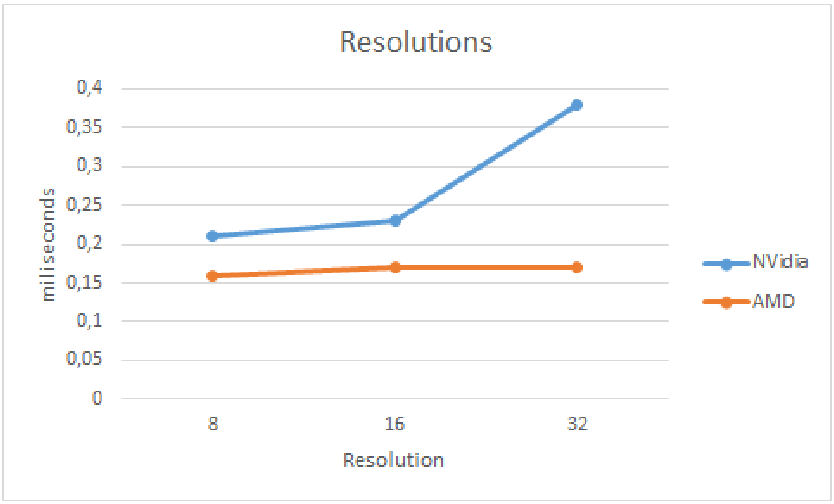


Figure 4.5: Rendering times for different face resolutions.

4.4 Memory usage

In table 4.2 we have compared the size of the original texture used for the head model to the size of the mesh colors color buffer using a resolution that gives a visual equivalence to the original texture when rendered.

Table 4.2: Memory usage comparison between a regular texture map of 1024x1024 pixels and mesh colors using a resolution resulting in visual equivalence.

Method	Samples	Uncompressed (bytes)	JPG compressed (bytes)
Regular	1 048 576	4 194 304	128 170
Mesh Colors	765 564	3 062 256	-

From the table we see that fewer color samples are needed for mesh colors to achieve a visual equivalence compared to rendering with regular textures (a decrease in the number of samples by approximately 27%). Since there are no established methods for compressing the mesh colors data the data is stored in it's uncompressed form. This results in the mesh colors color data taking up memory space of approximately a factor of 24 compared to the JPG compressed regular texture.

4.5 Issues

There are a few issues with mesh colors that Yuksel raises in [12] and most of these issues are reproducible in our implementation.

The lowest possible resolution for mesh colors is that of vertex colors, e.g. the lowest resolution is dictated by the geometric detail of the mesh. This could become a problem if the camera is far enough from the mesh for several faces to fall within the same pixel, since mesh colors filtering cannot sample across multiple faces in it's current implementation. This would involve traversing the geometry to find the neighboring face data. Most modern renderers including Frostbite use sub-pixel sampling and will thus weigh samples from several faces and produce a correct final pixel color.

Since a face has uniform resolution and the number of samples per edge and face are derived from this resolution in eq. 2.1, the use of elongated triangles will cause a disproportionate spatial distribution of color samples along edges and on faces. A degenerate case is illustrated in figure 4.6 where a low resolution and nearest filtering is used to clearly illustrate the issue. The original UV-map and UV-coordinates of the mesh were used to populate the mesh colors buffer. In the figure we see skewed hexagonal shapes running down the middle of the side of the church mesh caused by the elongated triangles in the geometry. An equilateral triangle would render with symmetrical hexagonal shapes.

4.6 Images

Figure 4.7 shows a larger image of the head model rendered with mesh colors with a resolution of 16. In figure 4.8 a house model with 236 triangles is rendered with mesh colors with a resolution of 256 to illustrate that the method works well with a high number of color samples per triangle as well. No UV-coordinates were used when rendering. The mesh colors color buffer was populated in a preprocessing step by sampling the mesh's original texture maps with existing UV-coordinates, therefore we achieved a visual equivalence with mesh colors compared with the original texture-mapping.

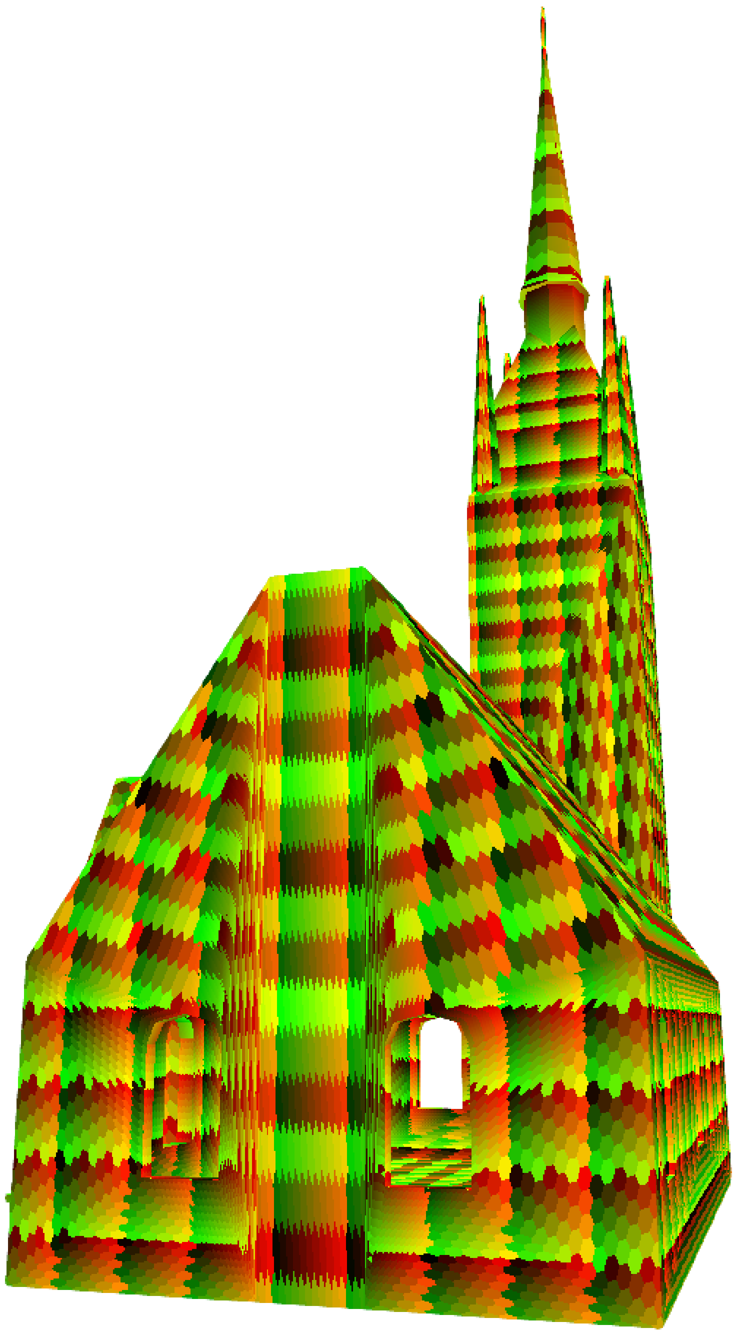


Figure 4.6: Skewing of color samples for elongated triangles.

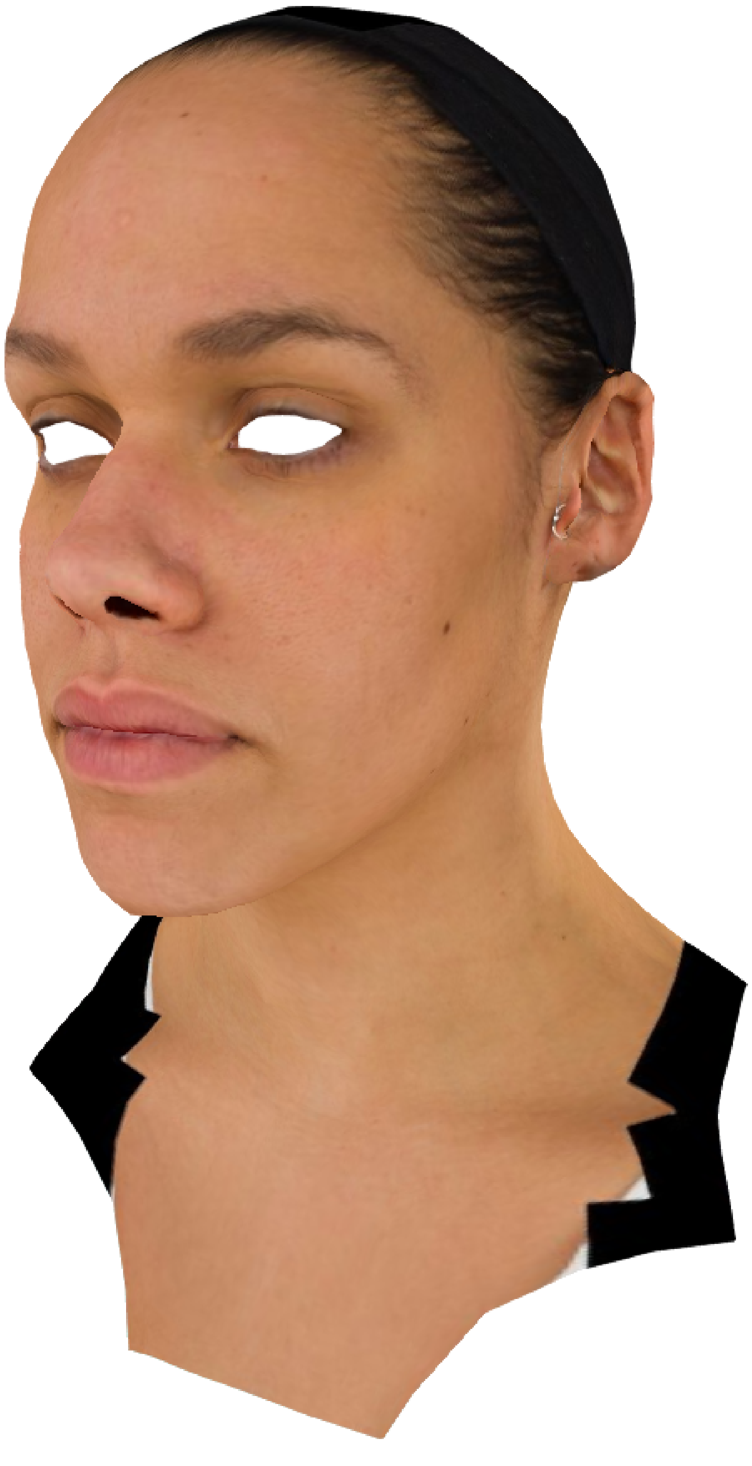


Figure 4.7: Head model rendered with mesh colors.

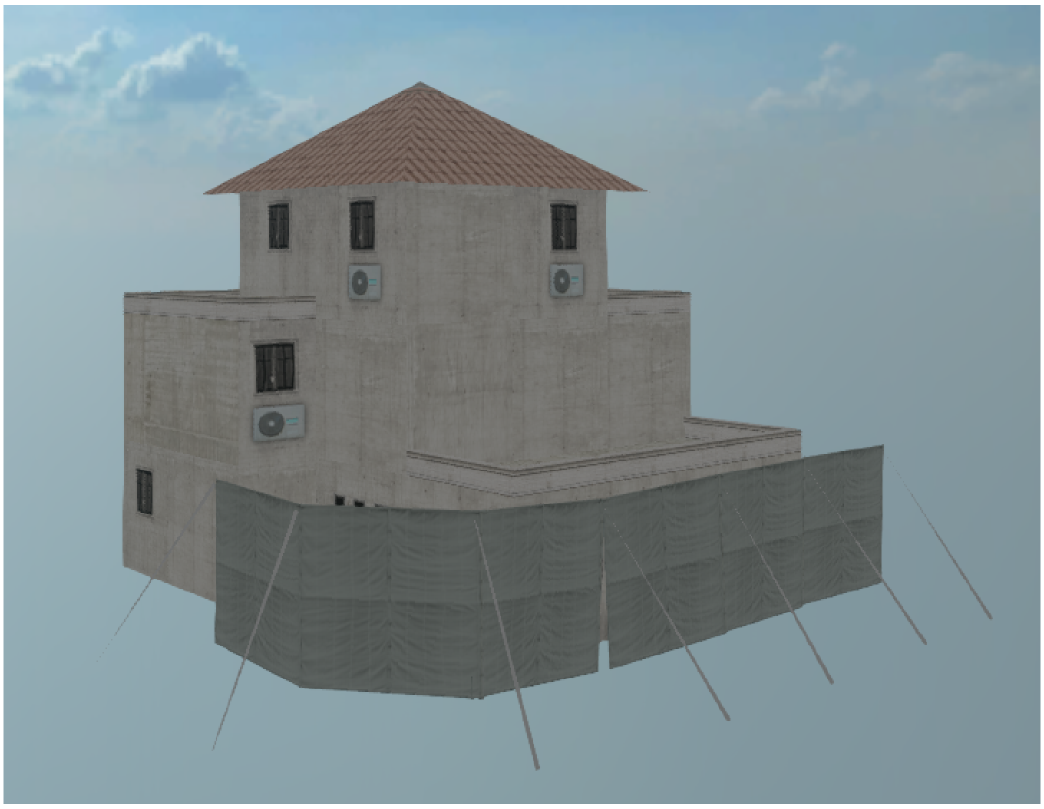


Figure 4.8: House model with a large number of color samples per triangle.

5 — Discussion

The rendering times in table 4.1 and in figure 4.5 indicate that the AMD card handles higher color resolutions better than the NVidia card. This might be due to the higher memory bandwidth of the AMD card (25% higher bandwidth compared to the NVidia card used). This difference in memory bandwidth might also be the reason for the greater relative speedup for the Nvidia card for higher MIP-levels compared to the AMD card as seen in figure 4.2. The number of color buffer lookups in the pixel shader are the same regardless of the MIP-level, but with fewer cache misses for higher MIP-levels the difference in memory bandwidth between the graphics cards might be less noticeable. Since a change in resolution only affects the size of the color buffer this conclusion makes sense. These findings highlights the importance of the ability for mesh colors to use non-uniform face resolutions.

Compression of the mesh colors data is a big issue. The texture map originally used for the head model uses JPG compression when stored on disk. As seen in table 4.2 the mesh colors color data in our use case takes up approximately 24 times more memory on disk than a JPG compressed UV-mapped texture of visual equivalence. For mesh colors to be a realistic alternative to regular UV-mapped texture a solution to this problem would need to be researched. During implementation we saw that a high mesh colors resolution resulted in a color buffer that did not fit in the video memory of the graphics card and resulted in the data being swapped back and fourth between the CPU and the GPU, causing a big performance hit. This further highlights the importance into reseaching compression methods for mesh colors data for the method to be a realistic alternative to traditional texturing methods.

5.1 Comments & reflections

A lot of the time spent on this thesis was dedicated to getting familiar with the Frostbite code base and developing a plan for implementing mesh colors to work with the different pipelines in the engine, all the way from processing mesh data, to generating the mesh colors data buffers, to loading data in runtime, to getting the data to the GPU and finally rendering using the correct shader permutations. We have shown that mesh colors is a realistic alternative to regular UV-mapping for real-time rendering. Although not as fast and efficient as textures with hardware accelerated filtering operations and MIP-map support alike, one might consider using mesh colors for special cases where the method's advantages outweighs it's drawbacks.

5.2 Limitations in implementation

The implementation focused on implementing the core rendering technology of the mesh colors method in the Frostbite engine. For mesh colors to be used in production an implementation of a paint tool would have to researched and developed.

We only consider triangular meshes, although possible extensions of mesh colors to quadrilateral meshes and subdivision surfaces are mentioned briefly in [12].

Rendering functionality that would have to be considered if implemented fully is choosing correct MIP-levels depending on some metric e.g. the screen-space area of a triangle or distance to the camera for a pixel. Since MIP-levels are fully implemented in this thesis the choice in runtime of which MIP-level to use would be trivial to implement.

6 — Future Work

The usage of the tessellation stage in the graphics pipeline seems superfluous since no actual tessellation takes place. Granted, mesh colors could trivially be extended to support tessellated geometry as mentioned in [12], but for this implementation it is only used for the per-patch context available in the hull shader. An avenue of future investigation could be looking into solutions that does not need to pass through the tessellation stage.

Another issue is compression of the color data, primarily on disk but also during runtime. During the writing of this report there is currently a thesis project under way at DICE that is looking at efficient compression algorithms specifically for the data generated and used by mesh colors.

In a real-time application such as games meshes often have several representations of decreasing geometric detail, often referred to as different levels of detail (LOD). A mesh's LOD levels are somewhat a mesh equivalence to a texture's MIP-levels. For this thesis we have implemented MIP-levels for mesh colors, but have not investigated if and how the mesh colors data from the original mesh could be used when generating mesh colors data for the different LOD-levels of a mesh. This is a subject that could be investigated further.

Another interesting area for future work is an implementation of a paint tool using mesh colors to enable artists to use mesh colors in production. For a paint tool one would have to take a number of items under consideration, e.g. usable work flows, suitable data structures etc. A tool for use by artists would have to handle editing operations which are trivial when adding or deleting geometry, but it gets a bit more tricky when for example incrementing the resolution of arbitrary faces. The pipeline for this thesis consists of generating the color data as a preprocessing step on the CPU, and neatly packing both face data and color data in linear arrays ready for rendering. The tool would need to use a data structure which can handle on-the-fly adjustments of e.g. face resolutions, inserting data into the color buffer and updating all affected color buffer indices in the face data buffer. A linear array structure as used in this thesis would entail inserting data into an array, shifting all subsequent elements and adjusting all affected color indices in the face data buffer. One suggestion might be to have an editing mode (e.g. "painting mode") using some form of map structure, and when finished painting again pack all data neatly in linear arrays suitable for rendering.

Bibliography

[1] Automatic generation of mipmaps. http://msdn.microsoft.com/en-us/library/windows/desktop/bb172340(v=vs.85).aspx. Accessed 2014-06-03.

[2] Barycentric coordinate system. http://http://en.wikipedia.org/wiki/Barycentric_coordinate_ Accessed 2014-08-28.

[3] Deep paint 3d. http://www.diskovery.com/. Accessed 2014-06-03.

[4] Deferred shading. http://en.wikipedia.org/wiki/Deferred_shading. Accessed 2014-08-12.

[5] Mari. http://www.thefoundry.co.uk/products/mari/. Accessed 2014-02-24.

[6] Tessellation overview. http://msdn.microsoft.com/en-us/library/windows/desktop/ff476340(v=vs.85).aspx. Accessed 2014-06-05.

[7] B. Burley and D. Lacewell. Ptex: Per-face texture mapping for production rendering. In *Proceedings of the Nineteenth Eurographics Conference on Rendering*, EGSR'08, pages 1155–1164, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[8] E. Catmull and J. Clark. Seminal graphics. chapter Recursively Generated B-spline Surfaces on Arbitrary Topological Meshes, pages 183–188. ACM, New York, NY, USA, 1998.

[9] A. Schilling and G. Knittel. System and method for mapping textures onto surfaces of computer-generated objects, May 22 2001. US Patent 6,236,405.

[10] R. Stuart Fergson. *Practical Algorithms for 3D Computer Graphics.* A K Peters/CRC Press, 2 edition, 2013.

[11] A. Tarandi. Surface data on dynamic topologies. Master's thesis, KTH Computer Science and Communication, 2014.

[12] C. Yuksel, J. Keyser, and D. H. House. Mesh colors. *ACM Trans. Graph.*, 29(2):15:1–15:11, Apr. 2010.

[13] J. Zink, M. Pettineo, and J. Hoxley. *Practical rendering and computation with Direct3D 11.* CRC Press, Boca Raton, 2011. An A.K. Peters book.