Image Vectorization, Compression, and Transmission

Brian Wu Dru Ellering

Figure 1. Progressive compression of an image from original to an uncompressed vectorized form, a 90% compressed vectorized form, and a 99% compressed vectorized form (left to right).

Abstract

While many methods of image compression exist, most of the techniques performed today focus on shrinking the size of the image for storage, not for image transfer, which causes problems when attempting to send a detailed image. We introduce a method for vectorizing existing raster images that contain arbitrary geometry. These vectorized images can then be dynamically downscaled while retaining an approximation of the initial geometry while also significantly reducing the amount of data needed to be stored.

Because of the vectorized structure behind these images, any compressed version of them can be sent and then restored to the original image by applying a sequence of inverse transformations. Each transform can be sent in an optimized order as an individual packet, restoring important parts of the image first. The exact ordering of reconstruction also does not need to be maintained, and the receiver is able to determine if an inverse transform is valid at the current image state. This allows for the transforms to simply be buffered until they are able to be restored.

1 Introduction

With the advent of the internet, the transmission of images and other visual media has become commonplace. Even newly created forms of media, such as image macros, have become an integral part of communication in the modern age. As such, research into the various ways to store an image more efficiently (a process known as image compression) has long been explored. However, these methods are often used for image transmission as well as a general purpose tool. This leaves an interesting niche, focusing on bettering image transmission rather than general image compression.

This concept has been explored in other domains such as in the rendering of three-dimensional scenes. For objects far away from the camera, low-poly versions of the model are loaded to save on memory.

While there are a great many types of images, some of the most important ones contain structure to them in order to display meaningful visual information to the viewer. For this reason, the images that we will focus on in this paper are those with patterns instead of those with random noise.

2 Prior Work

The basis of this work is rooted in the principles of mesh subdivision, specifically a variant of the Catmull-Clark subdivision surfaces algorithm introduced by DeRose et al. in 1998. Catmull-Clark subdivision is a method of subdividing existing geometry into triangles, making for a rather versatile algorithm. Their improved method introduced a method for maintaining sharp edges and creases when subdividing. The underlying mesh representation is a grid-like arrangement of quads, which makes it compact. Taking the limit of the subdivision to infinity results in a smooth surface, which is useful for maintaining quality.

When it comes to reversing any basic subdivision, Hoppe shows a method of progressively representing a complex mesh as a series of edge collapses. This method allows for the simplification of complex meshes to save on space, but also allows for the continuous transmission of the mesh at various levels of quality. Edge collapses are invertible, which allow for the compression of a dense mesh into a simpler mesh; the dense mesh can be retrieved losslessly. Another aspect of progressive meshes is selective refinement; areas of high detail can be selected for refinement, which is useful for saving resources.

The *progressive mesh* representation has certain advantages over traditional image compression algorithms. Structured images can be represented by a mesh with very little storage space. Additionally, meshes are vectorized and can be infinitely scaled. An image in progressive mesh representation can be *progressively* transmitted such that higher-level details are introduced on top, instead of in raster order; this key feature allows the image to be shown as the data is being downloaded, instead of all at once after all the data has been downloaded.

Previous work in image vectorization has also been done. Sun et al. described a method of transforming raster images into vector images while encoding smooth transitions as seen in real life. The mesh utilizes bezier curves and color differentials to ensure the vectorized image appears as realistic as possible. In order to represent sharp lines and boundaries, two slim patches on either side of the line are used. Hoppe goes on to demonstrate that representing the image as an optimized mesh is better at preserving the original image when upscaled.

3 Vectorization

The first step in our algorithm is to convert a raster image into a vector image. Vertex position and color are

represented with an array of three binary32 IEEE 754-2008 floating point numbers. A vertex anchor attribute determines whether a vertex is anchored to the edge of the image; this consideration helps prevent edge collapses from altering the frame of the image. Additional member attributes store information for facilitating the algorithm.

We will now introduce several algorithms for converting the raster image into the vector mesh. A visual comparison of these methods can be seen in Figure 2.

The first method creates a one-to-one mapping by creating a vertex for each pixel in the original image. Vertices adjacent in the x and y axes are connected with an edge and the diagonal of each quad in the mesh is split such that all the diagonals are parallel. This formulation is relatively simple and is relatively smaller when compared to the other methods. One of the major drawbacks, however, is that the original image is not preserved and especially on low resolution images, the colors of adjacent vertices bleed into each other; this is because triangles connect the adjacent vertices and the color between them is linearly interpolated.

The second method is exactly the same as the first except that the diagonals alternate directions in a checkerboard manner. This reduces some of the bias from the diagonal directions but still does preserve the image

Finally, the third method creates four vertices for each pixel. This allows each pixel to be represented as two triangles, while linking triangles, in between each pixel, keep the mesh manifold. The advantage of this method is that the shape and boundaries of the original image is preserved but comes at the cost of significantly more triangles to render and vertices to store.

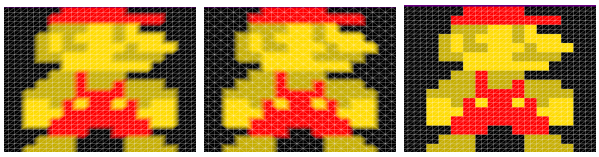

Figure 2. Connectivity of the initial vector image mesh, where the white lines represent triangle edges. The left image shows method one with parallel diagonals, the middle image shows alternating diagonals, and the right image shows a true representation of the image with four vertices per pixel.

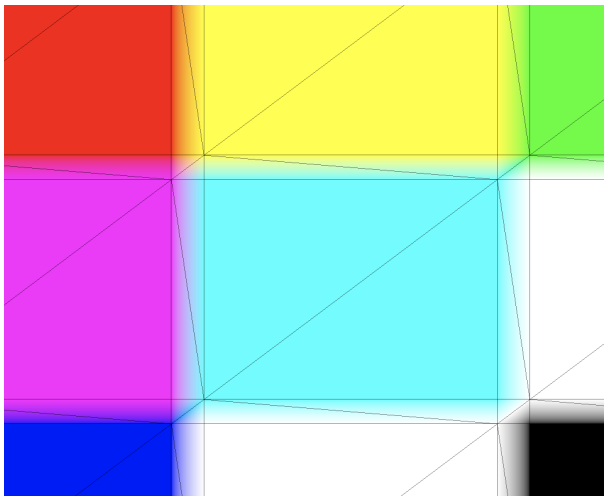

Figure 3. The third method with wide offsets for the edges. The edges between pixels are represented by two slim triangles, which allows for sharp borders between colors.

4 Collapsing Edges

After the image has been vectorized, a basic edge collapse algorithm begins. In order to aid the edge collapse algorithm, a half-edge data structure was created in Rust. The mesh itself stored a vector the the vertices, edges, and triangles. To ensure that the mesh remained manifold, even with highly irregular collapses as might occur in certain images, the mesh was queried for overlapping triangles after each collapse.

In order to determine which edge to collapse, all edges are ranked based on their energy, which is calculated as a function of the length of the edge and the distance between the colors at each of the edge's vertices. The edge with the minimum energy is the best edge to collapse, but only after checking if the edge collapse is valid

Figure 4. Progressive collapse of a mesh's edges. Going from left to right and top to bottom we have the initial vectorized mesh, the same mesh at 25% compression, 50% compression, and 90% compression.

When an edge is collapsed in the mesh, attributes such as position and color are simply averaged between the two incident vertices. The resulting vertex takes on these values, while the anchor attribute is inherited from the most strict parent vertex.

4 Splitting Vertices

In order to undo an edge collapse, an inverse operation called a vertex split must be performed. Taking inspiration from Hoppe's work, any time we do an edge collapse, we store the index of the v_l, v_r, and v_s, vertices shown in Figure 5 along with the two deltas required to move v_s to its original positions before the edge collapse. Using smarter encoding, as Hoppe has shown, this information can be reduced, but a more straightforward and understandable approach was taken here to demonstrate the concept.

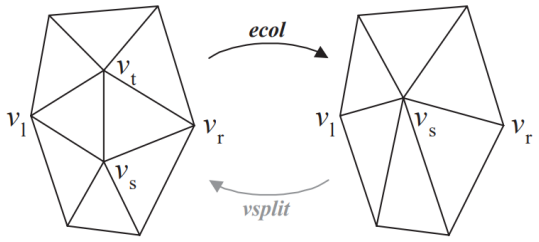

Figure 5. Visualization of a basic edge collapse operation and its inverse, a vertex split (Hoppe, 1996).

To know when a vertex can be split, we simply need to check if v_s, v_l, and v_r exist in the current mesh. Since the indices of the vertices have been saved, this check is a simple O(1) lookup. It is worth noting that when the initial simplified mesh is sent, the number of vertices also

needs to be sent along with vertex position information to preserve this property.

Finally, in order to implement selective refinement of the mesh, we need a quick way to gather all valid vertices to split near a given point. For this, we add yet another kd tree into the mesh structure. Anytime we add an entry to the history, we also append the position of v_s along with its index in the history list to the tree. When looking for a vertex to send, we can then quickly find the nearest vertex to the point of interest, then send that vertex split if it is valid for the receiving mesh.

5 Optimizations

This algorithm is expensive due to a number of factors. The initial creation of the mesh is essentially linearly upscaling the image to create the triangles. As such, millions of vertices need to be rendered for a high definition image. In addition, ensuring the mesh remains manifold through arbitrary collapses requires accessing the triangles, of which a naive implementation would require O(n) access time. In addition, uncollapsing the edges in a traditional half edge mesh would also require a naive search through all edges, once again requiring O(n) access time.

Because of these factors, the reduction of vertices that our first two vectorization methods obtain highly impacts the performance. For this reason, all images generated in this paper used the first method unless otherwise stated.

To further optimize the algorithm, a kd-tree containing each triangle's index and centroid was stored in the mesh. With this, we could gather each altered triangle in an edge collapse and find a radius encompassing all possible triangles that those triangles also affected, even if they were not directly connected. This reduces the need to check for overlap in every triangle in the mesh to only those possibly affected. In our testing, this change alone reduced the time needed to make a significant amount of collapses drop from hours to minutes.

The other optimization was once again to modify the traditional half edge structure such that each vertex was able to instantly access one of its corresponding edges. We could then simply search the neighborhood around that vertex for the proper edge, rather than manually sorting through each possible edge when splitting a vertex.

Lastly, the render pipeline was optimized to reduce duplication of data. Along with a buffer of vertices, an index buffer is passed along to the pipeline. The index buffer contained the index of the associated vertex in the vertex buffer. Now, a sequence of three indices can be passed to the renderer to define a triangle in the mesh.

6 Results

The initial results of this method are extremely promising, with a 90% reduction of edges producing vectorized images that are sometimes difficult to tell apart from the

initial rasterized image as shown in Figure 5. In other examples, the vectorized image looks better than the input raster image, especially in cases where the raster image was initially rather noisy and aliased. This effect is largely due to the imperfect method of vectorization chosen. By not respecting the initial image's edges, we instead interpolate between those edges, making soft edges look much more natural, like in the case of Figure 4.

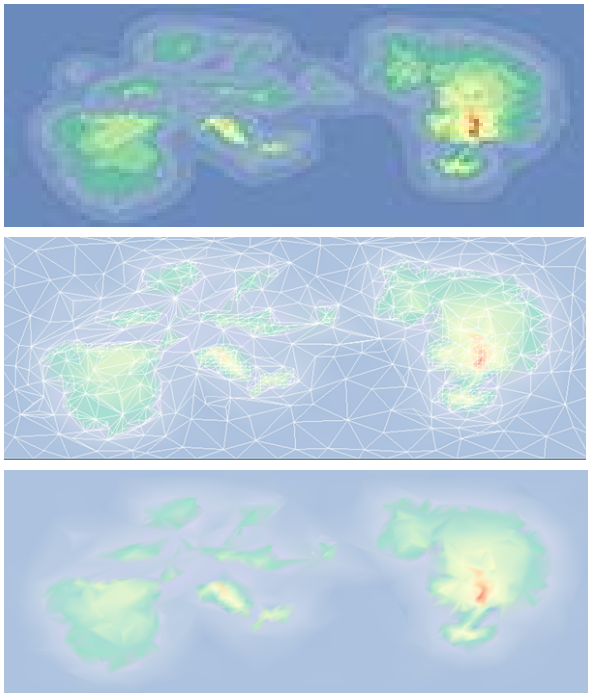

Figure 6. An example of a 120x60 map compressed by 90%. The top image is the input image, the middle displays a visualization of the compressed mesh and the bottom is the resulting compressed output.

However, this does have its drawbacks. For small, purposefully pixelated images, such as in Figure 2, the soft edges look extremely out of place. In fact, any time there is a hard edge, the simplification of the edge produces very noticeable triangular artifacts as seen in Figure 7. While this is acceptable in noisy images, the triangular artifacts in some images may not be preferable.

Figure 7. The progressive collapse of a mesh's edges along a circular curve. Going from left to right we have the initial vectorized image, the same image at 90% compression, and a visualization of the mesh at 90% compression.

Allowing support for arbitrary polygons may help minimize such artifacts by providing a more controllable edge to fit the shape, however, no matter what shape is used, there will be a linearization of the simplified images. Instead of polygons then, perhaps looking into the work of Sun et al. to incorporate bezier curves into the mesh would serve to reduce the effects of linear artifacts.

The last major issue is with the alterations of the colorspace from the input images. While the RGB values are respected, there is a significant desaturation effect on some file formats. This is likely due to an unfaithful transformation from the image colorspace to our colorspace, and while this issue is beyond the scope of our work, should be simple to fix.

7 Conclusion

In this paper we have shown a method for converting raster images into vectorized meshes and then

transforming the mesh with a series of edge collapses to create a compressed representation. We can then recover the original image through a sequence of vertex splits. One application of this technique is the loading of images. The compressed base mesh is quickly loaded first due to its small size. The sequence of vertex splits is then loaded and performed on the mesh until the final mesh is recovered. Loading images in this way can help prevent popping artifacts when images load all at once.

8 Future Work

The runtime of this algorithm can be improved with various optimization techniques. More vertex attributes can be explored such as keeping track of color derivatives to maintain a smoother image.

Another approach to this problem could utilize quad meshes or Ferguson Patches like in Sun et al. Quads could potentially lend itself to vectorization especially since they correspond better with the pixels in the rasterized image.

9 Acknowledgements

We would like to thank both Dru Ellering and Brian Wu for their hard work on this project. Specifically, we would like to thank Mr. Ellering for implementing a half edge data structure in Rust along with the basic vectorization, edge collapse, and vertex splitting algorithms along with general optimizations; and Mr. Wu for implementing the rendering pipeline in Rust, shaders, the edge collapse energy function, vertex anchors, additional mesh initialization methods, and the various other extensions to basic edge collapse.

References

DeRose, T., Michael, K., and Tien, T. 1998. Subdivision surfaces in character animation. In *Proc. of the 25th annual conference on computer graphics and interactive techniques*, 85-94.

Hoppe, H. 1996. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 99-108.

Sun, J., Lin L., Fang W., and Heung-Yeung S.. 2007. Image vectorization using optimized gradient meshes. In *ACM Transactions on Graphics (TOG) 26, no. 3*, 11-es.