AUTOMATED EMBROIDERY PROTOTYPE SIMULATION:

AN INDEPENDENT STUDY IN 3D COMPUTER GRAPHICS RENDERING

Jared Wickman
May 14, 2002

Overseen by:
Dr. David Goldman
and Dr. Richard Eckert

OVERVIEW

In the automated embroidery business, the ability to present a client with a tangible design prototype is highly desirable. However, the cost of materials involved with creating such prototypes increases sharply when one takes into consideration the thousands of possible combinations of thread types and colors, stitch patterns, and applications of a particular embroidery design. I define applications of an embroidery design as what the design will eventually be embroidered onto, such as hats, coats, shirts, and other articles of clothing. Thus, the simulation of these prototypes using three-dimensional models can allow for higher efficiency at lower costs.

The goal of this project is to take a bitmap, created in a stitch simulator, as input and apply it as a texture image to a Bezier patch. A user, to create the illusion that it is in the same orientation and shape as an object in the two-dimensional background image, then manipulates the patch, or mesh. A final image is produced by rendering this mesh with the texture map, thereby creating the illusion that the design has been embroidered onto some article of clothing in the background image.

Throughout the course of the project, different students have worked on different portions of the code. Mo Chen fashioned the transformation matrices and many different aspects of manipulating the mesh, such as scaling and distortion. Joseph Pamer worked on the rendering process, including texture mapping and bitmap loading. Last semester I designed the graphical user interface (GUI) for the program and aided Joe and Mo in their portions. This semester, I rearranged and corrected a large amount of the preexisting code, redesigned the GUI, and added new processes for texture mapping; specifically, a scanline algorithm used for obtaining the exact texture values for a given

output pixel. This semester as well, Kenny worked on implementing a different texture mapping filtering routine based upon an article by Feibush and Cook. He also improved the process of converting between (x,y) screen coordinates and (u,v) texel coordinates.

THE RENDERER
(class CRenderer)

The renderer is responsible for the world, view, and projection transformations that display the mesh. This is accomplished by storing the viewing system parameters and updating stored transformation matrices whenever the parameters are altered. A given mesh transforms its own coordinates by sending each vector to the renderer, and then stores the new coordinates in a separate array so that the 3D coordinates are not destroyed.

The renderer can in some ways be likened to a type of specialized device context. A background image, texture image, mesh to be rendered, and the point where the mesh origin will be rendered are selected into the context of the renderer. A user manipulates the mesh, updating the viewing system parameters, and presses the *"Render"* button to begin the rendering process. At this point, the location of the mesh is converted to the appropriate coordinates within the background image and selected into the rendering context. The aptly titled *"Render"* method of the renderer is called, passing a progress bar control to the rendering process so that the GUI will be updated as the image is rendered. The Render method starts a new worker thread and returns control to the GUI thread.

At this point in the project, there are several algorithms that can be used to render a final image: QuickRender, FilterRender, and ScanlineRender. QuickRender is a simple point sampling algorithm that renders very quickly, and is only meant as a draft mode to

allow the user to orient the mesh properly without significant delay; it is by no means accurate, and provides no alpha-blending with the background image. FilterRender is the implementation of the Feibush and Cook algorithm created by Kenny. ScanlineRender is the rendering algorithm that uses a method for obtaining exact pixel color values based upon a weighted average of the area a screen pixel maps to on the texture image surface. Both FilterRender and ScanlineRender require the method *"GetUVCoords"* in the CRenderer class in order to find what the (u,v) texel coordinates are for a given screen pixel location. FilterRender additionally requires the *"GetXYCoords"* method of CRenderer, which does the reverse of the operation *GetUVCoords*. Both *GetUVCoords* and *GetXYCoords* were written by Kenny and utilize barycentric coordinates.

SCANLINERENDER

ScanlineRender is a thread function that renders the currently selected mesh onto the background surface, using the currently selected texture map. *ScanlineRender* begins by setting up the necessary structures for the Scanline Average Color Algorithm *(see below).* The basic approach of *ScanlineRender* is to loop through each x and y value within the bounding rectangle of the transformed mesh. For each (x, y), the bounding rectangle of the pixel is calculated. For each of the four corners of this rectangle, a call is made to *GetUVCoords* to interpolate the texture coordinates (u, v) corresponding to the given (x, y) coordinates *(see Kenny's description of Barycenter coordinates)*. For the time being, if any portion of the pixel lies outside of the mesh, it does not get rendered. If the pixel lies completely within the mesh, the resulting quadrilateral in texture space is then clipped to the texture image using the *GetClippedPolygon* function. *GetClippedPolygon* functions similarly to *GetCornerWeight* except for the fact that it returns a polygon rather

than the area of the resulting polygon. The clipped polygon is passed to the scanline average color algorithm via a call to *GetAvgFillColorScanLn*. This function returns the weighted average of the color values in the texture image that are contained within the input quadrilateral. The resultant color is finally combined with the output surface using alpha values, where an alpha value of 0 represents completely opaque and an alpha value of 255 represents completely transparent. A pixel that occurs completely within the texture map will use the weight given by the returned alpha value. However, for a pixel that occurs on the border of the texture map, the alpha value is multiplied by the ratio of the area of the texture map contained within the pixel to the total area of the pixel. This creates a smooth edge along the border of the texture-mapped image.

In order to speed up the rendering process, (u, v) coordinate calculations are stored and used as the algorithm progresses. After each column of pixels is rendered, the progress bar, passed to the renderer at the beginning of the rendering process, is updated to reflect the portion of the bounding rectangle of the mesh that has been rendered. When no pixels are rendered in a column, the rendering process is finished, and the final image is outputted to a 32-bit ".bmp" file.

SCANLINE AVERAGE COLOR ALGORITHM

The entry point of the scanline average color algorithm is the function *GetAvgFillColorScanLn*, located in *ScanlineRender.cpp*. Below are descriptions of the data structures used by the algorithm.

VERTEX LPVERTEX	A single point in texture image space. This holds two coordinate values, *x* and *y*, which are double precision numbers. Additionally, each vertex contains pointers to the next (*pNext*) and previous (*pPrev*) vertices of the input polygon. The only restrictions on this are that the polygon is convex and that the vertices are listed in clockwise order; in

	other words, *pNext* points to the next vertex in the polygon in clockwise order, and *pPrev* points to the next vertex in counterclockwise order. LPVERTEX is merely a pointer to a VERTEX object.
ACTIVEEDGE	An edge of the polygon between two VERTEX objects. This structure contains a double precision x-intercept value (*x*) that represents the location at which the edge intersects the current scanline. The inverse slope of the edge is stored in the *dx* member to avoid computing it multiple times. Additionally, the endpoints of the edge are stored in the *pVertex* and *pVertexTop* member variables. *pVertexTop* is the vertex with the lesser y-value, and *pVertex* is the other.
SCANLINE	This structure holds much of the information needed for the algorithm. The purpose of this structure is to avoid the frequent allocating new memory for each of its member variables in various functions. Instead, this structure is passed as a reference to the necessary functions. Additionally, this method decreases the number of parameters necessary to call the subfunctions. Below are explanations of the member variables. The member variables *pdWeightBuf* and *pVSub* must be allocated before the SCANLINE structure is passed to *GetAvgFillColorScanLn*. *VERTEX vBoundRect[4]* vBoundRect stores the coordinates of the corners of the current texel location. This is useful so that these locations do not have to be recalculated many times to find intersections. *double* pdWeightBuf* pdWeightBuf is a pointer to an array that holds the current values of the weights calculated for the pixels of the current scanline. As the algorithm progresses, the weights anywhere in pdWeightBuf can be updated to reflect the area of the texel that is within the input polygon. *int nLeftPixel and int nRightPixel* The boundaries of valid weights within pdWeightBuf for the current scanline. The values between nLeftPixel and nRightPixel are those which contain at least some of the input polygon. *bool bDirection* This variable represents the direction in which the algorithm is currently progressing: clockwise or counterclockwise. *(See the description of the algorithm below)* *bool bTopBottom*

	This variable represents whether the algorithm is currently processing using the active edge list or the previous edge list. If the algorithm is using the active edge list, the edges intersect the bottom of the scanline, and bTopBottom is false. Otherwise, the algorithm is processing the edges intersecting the top of the scanline (previous edge list), and bTopBottom is true. *LPVERTEX pVSub* The purpose of this variable is to hold vertices that were found to be part of a subpolygon. This is merely so that new memory need not be allocated for each iteration.

The main idea of this algorithm is to divide the image surface into a number of scanlines, each with a height of 1. Each pixel within the scanline is given a default weight of 1.0. If an edge occurs within a given pixel in the current scanline, the area outside of the polygon within the pixel is subtracted from the weight buffer. As the algorithm progresses, the outside weights are subtracted, until all of the edges intersecting the scanline have been processed, leaving the correct weights for the scanline in the weight buffer. There are four subcases: when zero edges intersect the bottom or top of the current scanline; when two edges intersect the bottom of the current scanline and zero edges intersect the top of the scanline; when two edges intersect the bottom of the scanline and two edges intersect the top of the scanline; and when zero edges intersect the bottom of the scanline and two edges intersect the top of the scanline. In the first case, many of the techniques, to be discussed in more detail later, of the rest of the cases are used. In the second case, or "top corner case", the active edge list is used, and the algorithm progresses clockwise from the leftmost edge to the rightmost edge. In the remaining cases, only the previous edge list is used. In the third case, the leftmost edge in the edge list is traced counterclockwise until it exits the scanline at the bottom. Similarly, the rightmost edge in the edge list is traced from the top of the scanline to the

bottom; however it is traced in a clockwise direction. In the last case, or "bottom corner case", the leftmost edge is followed around until it intersects the top of the scanline.

In order to accomplish the subcases described above, the tasks are broken down further into subfunctions. Each subcase above is handled by a call to the *GetEdgeScan* function. *GetEdgeScan* takes a starting vertex that is just outside the current scanline when following the edges in the appropriate direction. *GetEdgeScan* alternates handling edges using *DoEdge* and handling corners using *GetCornerWeight*. The *DoEdge* function updates the weights for those pixels along the current edge that do not contain endpoints of that edge. This consists of triangles formed by the edge intersecting the top or bottom of the current scanline and trapezoids in between. *GetCornerWeight* handles the other case, where there are one or more endpoints within a pixel. The function follows the edges in the proper direction, clockwise or counterclockwise according to the subcase, adding vertices to a buffer in order to form a subpolygon. *GetCornerWeight* calculates and adds the intersections between the bounding rectangle of a pixel and the edge, and simply adds all other vertices in between. Because it progresses in a clockwise or counterclockwise direction while adding these subpolygon vertices, *GetCornerWeight* then calls *TrapeziumArea* to find the area of this subpolygon. The area outside of this subpolygon is then subtracted from the weight buffer in *GetEdgeScan*. *GetCornerWeight* additionally checks to make sure that all vertices in the input polygon are not contained within a single pixel. In this case, all vertices are added and *TrapeziumArea* is called. *GetCornerWeight* and *DoEdge* call the function *SegSegIntersect* in order to find the intersection between two line segments determined by four input vertices. If there is no

intersection, *SegSegIntersect* returns false. Additionally, *GetCornerWeight* and *DoEdge* return an updated current vertex for the next iteration of the loop in *GetEdgeScan*.

In this manner, the algorithm calculates the weights of individual pixels within a current scanline in each iteration of the main loop in *GetAvgFillColorScanLn*. For each scanline, the weights are multiplied by the RGBA values and added to accumulator variables, accumulating the total weight as the algorithm progresses. The final RGBA values to be returned are found by dividing the component colors by the total weight.

FUTURE CONCERNS, IDEAS, AND SUGGESTIONS

In order to accomplish as much as we have given our time constraints, we have tolerated a certain lack in functionality that should be implemented in the future. For example, adding scrollbars to the 3DViewCtrl window would be very helpful. A useful design consideration would be to add functionality to be able to "grab and drag" the background, effectively scrolling in the opposite direction that the mouse was dragged.

One major inhibiting factor in this program has been that it is difficult to manipulate the control points of the mesh. In order to facilitate such manipulations, a multi-view approach might be used, as in other 3D modeling programs. Another approach that might prove to be useful would be to allow a user to enter specific world coordinate values. This would allow one to obtain a symmetric and highly customized mesh, but it would not be convenient for a user to type in all sixteen control points' coordinate values. Another method used frequently by 3D modeling programs is to provide buttons to restrict control point movement to particular axes or planes. This allows for more controlled and predictable movements. The next suggestion I have for the manipulation is to not use the scrolling wheel frequently provided with newer mouses at all costs. This

is not a predictable use of the scroll wheel, as it is not used in this manner in various other programs. As it is used now, the mesh control points move closer to and further from the camera position when using the scrolling wheel; this causes the mesh to change shape, but because of the orientation, this change is not visibly detected. Additionally, holding down the left mouse button while scrolling the wheel is not always an easy contortion for a user's hand, not to mention the obvious problem if a user owns a simple two-button mouse. My suggestion is to use the scrolling wheel for scrolling as it is intended and use the right mouse button instead for control point manipulations. Adding functionality to the GUI for changing between scaling, rotating, and translating modes would also be an improvement, as it allows for less user error. My last suggestion for the manipulation of the mesh is to allow a user to drag a control point in the 3DViewCtrl to rotate the mesh in various directions. However, if this is implemented, restriction of rotation axes should be used, because 3D rotation using dragging of the mouse can be very difficult to control otherwise.

In the various rendering algorithms we have developed, none as of yet provides alpha blending for the edge of a mesh. Additionally, the texture map is currently limited to having its origin at a particular corner of the mesh. In the future, it would be convenient to have a menu choice to select the texture image origin on the mesh by some method. The mesh would then be generated with new texture coordinates by starting in the middle of the texture map and approximating the length of the Bezier curve to determine the (u, v) values of adjacent vertices. It would also be practical to add a scaling factor, by which these approximated lengths would be multiplied to find the proper (u, v) values.

CONCLUSION

There is still quite a lot of work to be done on this project, such as implementing a lighting model and interface, but I am happy with what we produced this semester. The look and feel of the GUI has drastically changed and the renderering process has been completely reorganized and filled in. Many bugs remaining from previous versions have been found and eliminated. Finally, a few new features such as the ability to save meshes and scenes have been added to enhance the usability of the program.

I'd like to thank Dr. David Goldman and Dr. Richard Eckert for their guidance and the opportunity to work on this project, and my partners in the project for their hard work and dedication.

– Jared Wickman