Eurographics Symposium on Rendering 2003
Per Christensen and Daniel Cohen-Or (Editors)

Computer Generated Celtic Design

Matthew Kaplan and Elaine Cohen

University of Utah

Abstract

We present a technique for automating the construction of Celtic knotwork and decorations similar to those in illuminated manuscripts such as the Lindisfarne Gospels. Our method eliminates restrictions imposed by previous methods which limited the class of knots that could be produced correctly by introducing new methods for smoothing and orienting threads. Additionally, we present techniques for interweaving and attaching images to the knotwork and techniques to encapsulate knot patterns to simplify the design process. Finally we show how to use such knotwork in 3D and demonstrate a variety of applications including artwork and transforming the designs into 3D models for fabrication.

Categories and Subject Descriptors (according to ACM CCS): I.3.0 [Computer Graphics]: General I.3.3 [Computer Graphics]: Picture/Image Generation I.3.6 [Computer Graphics]: Methodology and Techniques

1. Introduction

Celtic decoration refers to abstract non-imitative artwork originating with Celtic tribes dating from about 500 B.C. One of the hallmarks of Celtic art is elaborate knotwork, consisting of entangled threads which maintain a strict over-under alternating pattern between every crossing of the threads. This knotwork is analogous to closed loops of rope that cross over and under one another, becoming entangled. The loops of rope are called *threads* that, when entangled, form the *knot*.

Although similar artwork has been found in many cultures, the specific period of artwork whose style we wish to reproduce is contained in the illuminated manuscripts of the British Isles, such as the Lindisfarne Gospels, the Book of Kells, the Book of Durrow and others, created between the 7th-9th centuries. It is in these works that Celtic art reached its zenith, as shown in Figure 16 inset.

At present, Celtic artwork is enjoying a renaissance; its popularity has manifested itself in design, fine arts, jewelry, body art, decoration of sculpture, and architecture. As pointed out by Wong *et al.* [17] there has been little work in the area of computer generated ornamentation despite the fact that ornamentation has historically played a critical role in architecture, decoration and art.

Because of the complexity of Celtic art, creating hand drawn designs is tedious and time consuming and often requires a significant amount of training to do well. Large designs are difficult to change and experiment with since local changes affect the entire knot. Our technique is presented as a tool to design such artwork intuitively, quickly and easily. It augments human talents for design with a computers skill at repetition and drudgery. It offers considerable time savings over designing such decoration by hand and allows users to experiment with global and local changes of both style and form quickly. Automating the fabrication of such knots for jewelry or decorative use also offers significant time savings over crafting comparable objects by hand.

Our method offers the following contributions:

- We improve over prior automated methods by allowing the creation of all possible Celtic knots based on planar graphs (rather than just a limited subset based on grids). Our method demonstrates how to construct such knots automatically, cleanly and without errors.
- We introduce techniques to automatically orient the threads around any configuration of user defined breakpoints or graph angles. This is one of the major improvements over prior research which severely limited the class of knots which could be correctly produced.
- We present the first method for computers that allows images to be interwoven and connected to the knot.
- We introduce several smoothing techniques to help draw

the threads more "artistically" and "naturally" and show how to stylize the threads.

- We show a new method of encapsulation and a background template builder to facilitate the design process.
- We show how to use Celtic knots in 3D by applying our algorithms to 2D manifold meshes.
- Finally, we generalize the algorithms to support manufacturing physical models of the knots created with this program.

2. Related Work

At some point after the 9th century the techniques used to create Celtic art were lost. George Bain [1] reinvented many of the artistic techniques necessary to create Celtic designs. His son, Iain Bain [2], simplified his father's methods to be surprisingly algorithmic. His method was based on a tri-grid system, which, while constrained by its inability to create knots outside the basic grid pattern, produced beautiful results quickly and easily. Meehan has published a series of books [11, 12, 13, 14] extending I. Bain's work. Mercat [16] gave rules for manually producing arbitrary knots by interpreting a planar graph as the basis for the construction of the threads.

Mercat [15] and Sloss [19] created programs that allowed users to create knots by connecting images of thread crossings. Complexity was added to the knots by connecting more images to the knot set. The use of predefined images means that these programs are restricted to a limited set of angles and patterns with which to define threads. This does not reproduce the general knot algorithm presented by Mercat [16] but rather a simple form of grid knot.

Glassner [5, 6, 7] showed how to compute threads, using I.Bain's method (as do [3, 8]), that are useful as a guide for creating hand drawn artwork. He concluded that the computer is not useful in automatically creating the knotwork because a human is able to interpret the thread pattern much more artistically, though he does show computer generated results for grid patterns. Glassner's technique required manual adjustment of curve drawing parameters in complex regions to produce reasonable results. Recognizing the limitations of the grid pattern, Glassner allowed the user to deform the grid shapes in order to output a larger class of knots. Using the deformable grids, Glassner proposed using the output only as a guide for hand drawn art. Glassner also showed 3D knotwork by unfolding the sides of simple objects into 2D objects and then refolding after the knot had been computed. This created 3D knots in the shape of the original object, but the results were not smooth. Moreover, his method seems difficult to use for all but the simplest of objects. Several knot programs such as those by Abbott [0] and Guionnet [9] use the Mercat technique.

These programs all suffer from a concern raised by Glassner: the inability of the computer to choose and draw a thread smoothly and "artistically". Previous work on graph based systems were all able to correctly determine how to connect the graph to form threads but were not able to contruct valid or smooth threads in any but simple grid cases. The usage of breakpoints (see section 3.6) changes the basic ordering on the graph and is vital to creating visual interest in the knot. In any but the simplest cases, previous programs failed to correctly handle most configurations of breakpoints and were not able to draw threads correctly around graph configurations whose angles weren't explicitly hard coded into the system. This is why Glassner proposes using the deformable grids as only rough guides for hand drawn art. We have developed a general solution to this problem which works for any graph configuration.

Many Celtic knot programs use the basic B-Spline curve method to draw the threads, using the crossing locations as control points for the thread curves. This is unsuitable because splines are not able to direct threads correctly using only position information (see section 3.3). Interpolatory spline methods may also introduce undulation artifacts into the curve. To account for these problems, an elaborate series of extra points are inserted for several predefined graph configurations to straighten out the lines and to direct thethreads around corners and breakpoints. They fail to correctly orient threads in any situation that does not fall into one of the predefined cases. This results in an inability either to draw straight, smooth lines with well formed graphs or to guide threads correctly around angles or breakpoints that are not predefined. Furthermore, threads may overlap other threads in an incorrect manner and exhibit strange discontinuities around sets of breakpoints and irregular graph structures. We show solutions to these problems by using a variety of curve smoothing and directing techniques that work for arbitrary graphs.

Knot theory is an area of mathematics that deals with the definition, structure, equivalence and minimization of knots. Scharein [18] implemented a program for display and manipulation of such knots. These results are not directly applicable to Celtic knotwork because the minimization, optimal display and balancing of knots transform the basic visual structure of the knot and the position of its constituent elements, a result that is unacceptable for our purposes.

3. Knotwork

The basic algorithm for artists as presented by Mercat [16] for creating a Celtic knot is conceptually very simple and we generalize it for our purposes:

Following algorithm 1 produces a complete three dimensional Celtic knot. When viewed from above, the result can be displayed in two dimensions. Figure 1 shows an example of this process.

3.1. Defining a Graph

The power of the method presented by Mercat is that every planar graph defines a knot. While methods based on

The Mercat algorithm (Alg.1):

1. Define a planar graph.
2. Find the midpoint of each edge. Put *crossings* at each midpoint.
3. Compute the *threads* that compose the knot by connecting the crossings.
4. *Inflate* the threads.
5. Calculate the overlap order of the threads and offset their height values based on the overlap order.

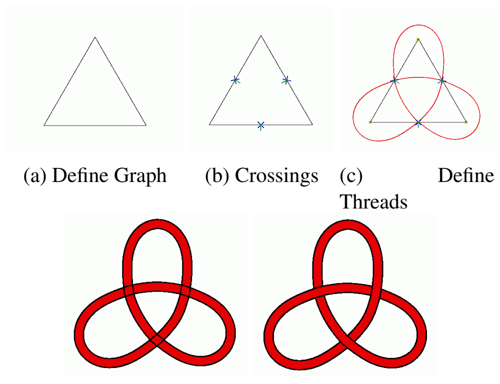


Figure 1: *The knot algorithm*

grid systems can be used to define a large number of knots, they are constrained by the basic topology of grids. A graph-based system can produce all possible knots. While it may seem counterintuitive to use graphs to define knotwork, simple geometric patterns often produce knots of striking complexity. Also, the underlying graph structure provides an easy, intuitive method for altering thread order via breakpoints, which we discuss in section 3.6

In our system, graph edges are represented by *strokes* that are drawn by the user and vertices are represented by *junctions*. Users are allowed to draw strokes with the mouse using either free-form or straight line styles. The system separates strokes where they intersect and culls tiny overlaps that occur due to the hand drawn nature of the strokes. Next, a set of junctions is automatically created at the endpoints of every stroke. A junction records a location, the strokes that have endpoints near that location and the counterclockwise ordering in which the strokes connect to that junction. Junctions that are close to one another are combined. This ensures that a user doesn't have to draw a graph "perfectly"; stroke endpoints just need to be close. We have defined functions that automatically produce several types of graphs such as rectangular grids, hexagonal grids and circular patterns.

3.2. Midpoint Information

The threads will cross each other at the midpoint of each stroke. We identify four vectors that lie at 45 degree angles to the stroke, based on the stroke and the stroke normal and label them as left or right as shown in Figure 2. The utility of this format is that from either endpoint, looking down the length of the stroke, both left and right directions are consistent. This property proves useful in calculating the threads. A *node* is defined to exist in conjuction with each of the four midpoint direction vectors. A node consists of a midpoint position and direction vector and has either a left or right orientation.

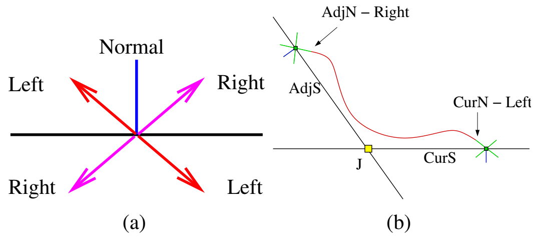


Figure 2: *a) Stroke geometry. b) Thread construction example.*

3.3. Thread Construction

At this stage, we can think of each thread as a curve in space that we will build based on the graph. First, a control set is constructed that defines each thread, linking the stroke midpoints. Second, the thread curve is calculated based on the control set.

Our thread construction algorithm proceeds as follows: Each control set consists of a circular linked list of nodes. The system selects a node from the set of unused nodes and marks it as used. Then it creates a new control set with this node as the first element. Nodes are added to the control set and marked as used until the starting node is reached. Since only one thread can pass through both nodes of either left or right orientation, both left or right nodes are marked as used in pairs. This creates an individual thread. The algorithm continues selecting unused nodes and creating threads until no nodes are left unused. A result of this can be seen in Figure 1c.

Now we discuss the algorithm that selects the next node to add to the control set, using a current node, $CurN$, a member of stroke $CurS$, as a starting position. First, the system finds the junction, J, that connects to $CurS$, in the direction that $CurN$ is pointing. It selects a stroke adjacent to the $CurS$ around J called $AdjS$. If the orientation of $CurN$ is left, $AdjS$ is the next clockwise stroke around J. Otherwise, it is the next counterclockwise stroke around J. The node to add, $AdjN$, is the node of left-right orientation opposite $CurN$ (i.e right if $CurN$ is left, and vice versa) that is pointing away from J on $AdjS$. An example of this is shown in Figure 2 b.

Based on the control set, the system computes the curve which defines the thread path. A curve is calculated for every *thread segment* in the linked list. A segment is a curve that runs between two adjacent nodes in the linked list. Note

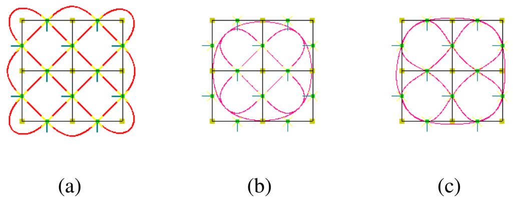


Figure 3: *Threads calculated with a)Cubic hermite curves, b)B-spline curves, c)Interpolating B-splines.*

that the last node in the linked list connects to the first node. Since each node in the thread list contains a position and direction vector, it is natural to use the cubic hermite representation to calculate the curves. Figure 3a shows threads obtained in this way. Because the B-spline curve method uses positional data, but not tangent direction and magnitude, methods that have attempted to generate thread paths using the basic B-spline curve approach have created auxilliary "meta" control points to guide the threads into correct paths around known corner situations and breakpoints. This results in undulations, unintended overlaps and irregularly shaped threads. Another problem with this is that it only allows the user to create corners and breakpoints without artifacts around a few predefined cases, thus limiting the benefit of the Mercat technique. A comparison of threads obtained using the cubic Hermite, the B-spline and the interpolating B-spline representations is shown in Figure 3. In both spline curves, the entire control polygon is used to calculate each thread rather than individual thread segments. It reveals the problem with using splines without modification; B-splines maintain the convex hull property and do not correctly orient the threads. While interpolating splines do slightly better at directing threads than B-splines, they introduce undulations into the curve. An automatic thread creation technique should remove undulations due to curve drawing errors. On the other hand, it is necessary to observe that some undulations occur because of the underlying graph and should not be removed. See Cohen *et al.* [2001] for a reference on how to implement the basic curve representations.

3.4. Inflating the Threads

The thread curves must be inflated so they have width. Therefore, we define a left and right curve for each thread. For every position in the thread curve, an offset in the direction of the curve normal is created. The offset is added to the curve position to obtain the right curve and subtracted to obtain the left. While this process can lead to overlaps on sharp curves, as shown in Figure 4a, our system searches for such sections and culls them as seen in Figure 4b.

The style of the threads can be changed by modifying the pattern with which the sides are inflated. An example of this is shown in Figure 4b where corners have been sharpened around breakpoints(discussed in section 3.6). Sharpening is done by altering the outer inflated curve of any thread segment with sufficiently large curvature and is also applied when two breakpoints are adjacent to one another at a junction. In this case, the median position of the outer curve is repositioned at a user-defined distance between it's original position and the junction it is closest to. The other curve positions are defined as a blend of the new median position and the original inflated curve positions based on parameter distance from the median. Figure 4c shows how the sharpening is accomplished.

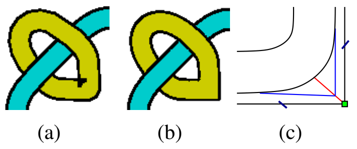


Figure 4: *The interior overlap in a) has been culled in b) and the outer edge has been sharpened. c)This figure shows how we sharpen corners. The thread path is altered to follow the blue lines towards a position at some user defined distance along the vector that lies between the junction and the median of the outside curve.*

3.5. Overlap

To display the proper *over-under* characteristic pattern, we must apply an alternating z-offset value between consective nodes in the thread linked list. Our system uses 1 and 0 as the over and under z-offsets. Due to our use of images, a slightly more complex algorithm is required than that presented by Mercat. Our algorithm traverses each threads linked list and computes the overlap values for every node, switching overlap value between each thread segment. To enforce a consistent ordering, this algorithm recurses to any thread that intersects the current thread, starting with the opposite overlap value. For each image encountered, the overlap order may need to be reversed (as described in section 4.2) after curve calculation, which is why we don't simply add the offset values as inputs to the thread construction algorithm. We linearly interpolate the overlap values between nodes as our z offset. The result is shown in Figure 1e.

3.6. Breakpoints

In section 3.3 the threads are calculated by a constant ordering on the graph. Changing the thread connection order around the midpoints leads to different thread patterns. *Breakpoints* can be used to change the regularity of the knot and add visual interest. The user can define a breakpoint at a stroke. This creates one of two new crossing patterns at the stroke midpoint, both illustrated in Figure 5 b-c. The Breakpoint Connection Rules shown below define how the system calculates thread lists using each breakpoint type, illustrated in Figure 5g.

Because the natural connection location for a given thread

Breakpoint Connection Rules:
Type 1 - Thread linked lists may not add nodes from this stroke. When the recursive algorithm finds a node at a type 1 breakpoint, it applies itself again using the same orientation it started with.
Type 2 - This stroke is ignored as a graph element. The thread construction algorithm crosses the stroke edge and uses the next adjacent stroke to find the node to add to the control set.

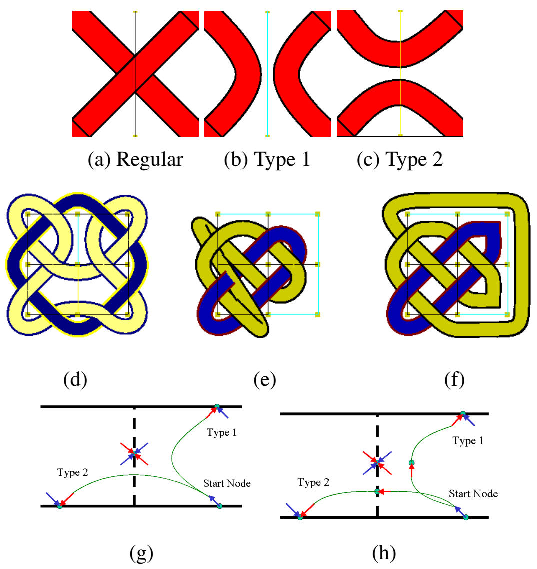


Figure 5: *a) A standard crossing pattern at a midpoint of the stroke, b) A type 1 breakpoint, c) A type 2 breakpoint, d) A knot illustrating the use of both types of breakpoints, e) An example of overlap and thread path errors that may occur when breakpoints are used (in this case due to the multiple adjacent breakpoints around the border), f) The corrected version, using meta points, g) The default handling of both types of breakpoints, h) The handling of both types of breakpoints using meta points.*

segment is skipped when a breakpoint is used, their use may introduce errors in several ways. First, the distance of the thread segment is roughly doubled and second, the direction of the curve is changed halfway through the thread which may introduce thread path and overlap errors as shown in Figure 5e. All previous graph-based systems would have produced something like the knot shown in Figure 5e for the given graph, when what is desired is shown in Figure 5f. We introduce *meta points* to guide the path of the thread around the breakpoint. Meta points allow us to use any set of breakpoints and do not require advanced knowledge about what type of graph structures will be used. This removes a limitation of previous work and allows us to create any possible Celtic knot without thread path errors.

A meta point will be a new node in the thread control set that is automatically inserted by the system whenever a breakpoint is reached. This essentially functions to tie the curve path to the breakpoint position. Therefore meta points are placed relative to the midpoint of the breakpoint stroke. For type 1 breakpoints, the position of the meta points are defined as the midpoint plus or minus the stroke normal while the direction is the strokes tangent vector. For type 2 breakpoints, the position is the stroke midpoint plus or minus the stroke tangent vector and the direction is the stroke's normal vector. These meta point configurations are depicted in Figure 5h. The user is allowed to alter the meta point offset distance from the midpoint. This is an important variable in maintaining smooth knots. Reasonable values for this meta point distance seem to scale linearly with stroke size making it easy for the program to estimate starting values. The meta points are inserted into the thread list and the thread set is computed in the standard way. This allows coherent computation of curves with long sets of breakpoints as in Figure 5f.

3.7. Smoothing

An exact implementation of step 3 in Algorithm 1 may cause undulations in the threads of non-grid graphs. We would like to be able to smooth the threads to get a more natural “artistic” feel.

One extension that we have created supports automatic alteration of the set of directional derivatives. The method alters both the direction and the magnitude (speed) of the derivative. Too large a magnitude leads to internal overlaps of thread segments and unintended kinks and twists. Too small a magnitude degenerates to linear interpolation. We have created a heuristic to alter the magnitude that the system may use at the users discretion. A general equation for this heuristic is:

$$Magnitude = 1 * UserScale * Dist * Angle \tag{1}$$

where $Magnitude$ is the value which is multiplied with the normalized derivative vector, $UserScale$ is determined by the user, $Dist$ is based on the distance between nodes and $Angle$ is based on the angle between strokes over which a given thread segment is defined. The variables in this equation are taken from the factors that most influence the shape of the curve. While we have implemented several functions that define each of these scale factors, we find that setting $Dist$ equal to *distance between nodes / 12.5*, $Angle$ equal to *angle between strokes / 4.0* and $UserScale$ between 5 and 15 works well in practice.

To alter the direction of the derivative at a node, a difference vector is created by subtracting the position of the next thread node from the position of the previous node. The final direction vector is a convex blend of the original direction vector and this new difference vector. Currently the blend value is set by the user. Because this method doesn't deal well with corners, we dampen this effect around sharp angles by multiplying the blend value

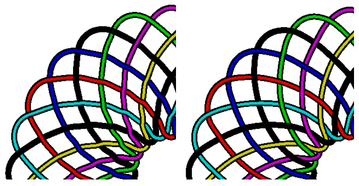


Figure 6: *The circular graph for this knot is ill formed which leads to the kinks evident in a). In b) the knot has been smoothed by altering the cubic hermite derivatives.*

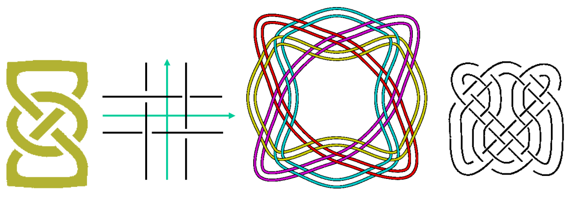


Figure 7: *Alternate knot display styles.*

by $(NextDir.DotProduct(CurDir)+1)/2$, where $NextDir$ is the value of the directional derivative at the next node and $CurDir$ is the value of the directional derivative at the current node. The results of this process are shown in Figure 6.

Though we discussed the reasons why B-Splines were not appropriate for thread construction, B-Splines have several properties we covet, especially smoothness. Smoothness can be derived from B-Splines, yet thread path correctness is reliant on cubic hermites as shown in section 3.3. Therefore, we use Schoenberg variation diminishing splines to vary between how smooth versus how correct we would like the final knot to be. First, the cubic hermite curve is calculated. Then, the cubic hermite curve is sampled at regular intervals to form a control polygon. Finally, a B-Spline is constructed from the control polygon for our final thread curve. The interval size of the sample determines how smooth versus how correct the final thread curve will be. We find values of .125, .25 and .5 to be useful sample intervals to use with this method, with .125 producing very correct, and .5 producing very smooth curves. Though full smoothing can be enabled by selecting a single check box, an example of each step in the smoothing process is shown in Figure 16.

3.8. Displaying the Knot

We allow the user to control the stylistic options of the thread nodes individually. Style information, such as width and color and stylizations are stored with the individual nodes in the graph so thread segments that contain the node can inherit style information. It is useful to store this information with the graph rather than the threads since thread patterns invariably change and any style changes would be wiped out each time the knot is recomputed.

A thread is displayed as a triangle strip in OpenGL that connects both left and right inflated thread lines. The border outline is displayed as a line strip in OpenGL of the left and right inflated thread lines. The border is offset by some small z value to avoid z-fighting with the interior.

Our method computes the intersections of thread lines which allows the display of several other styles frequently found in Celtic art. One of the styles stops drawing an under section before it reaches an over section, as shown in Figure 7a. Another is the use of border elements as threads themselves. We have discovered that this “border as thread” style has the property that each thread has a constant overlap pattern as shown in the template in Figure 7b. Therefore we can draw the left and right inflated thread lines with the template pattern for each thread segment without worrying about overlap information. The results of this are shown in Figure 7c. A combination of these two methods yields a style as seen in Figure7d.

4. Using Images With the Knotwork

One of the essential elements of Celtic Design in illuminated manuscripts is the use of images as part of the knotwork. Images are used in two ways: as terminus locations for the threads and as separate elements interwoven into the threads. By *terminus*, we mean that the thread connects to an image and ends. We present a completely new method for the use of images, differing from that described by Mercat [16]. The rules we describe are simpler, local, and tailored to our technique, making them more easily implemented and incorporated into the program. Mercat showed one way to use images as terminus objects, but his method constrained the type of knots that could use images. Our method is more general and can incorporate images into arbitrary knots. Moreover, we add the ability to entangle images with the threads.

We have implemented an image markup program that allows users to edit images and add data, saving the extra information in a separate file. This allows the user to add an alpha channel, to define *joints* and *spines* and *forces* as described below, and perform many standard image editing operations. The image is displayed as a texture mapped quadrilateral. The quadrilateral is given a z-value between the *over* and *under* values so that threads will pass cleanly above or below it.

4.1. Terminus

For each image, the data file contains a set of valid nodes. A node on an image is referred to as an image *joint*. We allow the user to set connections between graph nodes and joints by manually connecting joints and nodes with the mouse, as shown in Figure 8. A *connection* is a user specified link between nodes that forces the thread construction algorithm to follow a path between the linked nodes.

Because there is no such thing as a thread with only one

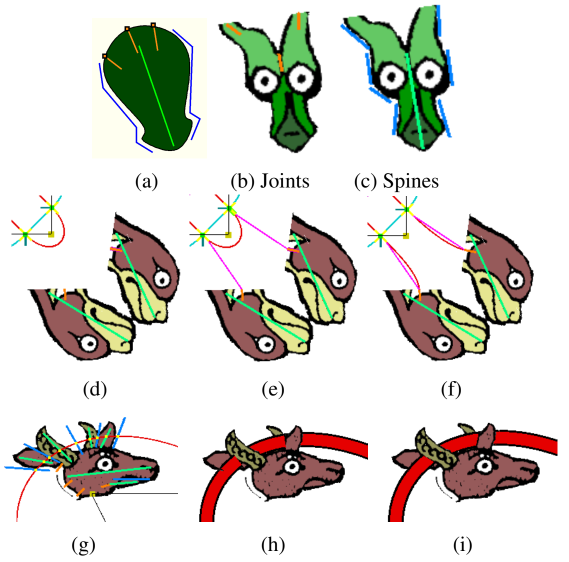


Figure 8: *a) An example of the image data structure. The green area represents the section of the image whose alpha value is one. Joints are orange, spines are light green and forces are blue b-c)Green dragon image with joints and spines. d) Thread set and images, e) Connections defined by the user, f) The threads have been recomputed and now connect to the images. g) Here, the thread curve intersects the image. Intersections are shown in yellow. h) Interweaving result; note that the spots where the overlap values switch don't appear to line up with the image features. i) Here, we have modified the inflate function to force inflation near image features to follow the* force. *Now the thread appears to switch overlap value in relation to the image features.*

end, terminus connections must be added to a thread in pairs. The second terminus object can occur as part of the same thread segment or an even number of segments away from the first terminus. Other configurations will not preserve the overlap order and will introduce a discontinuity into the knot.

After defining two terminus connections, we have associated two joints with two nodes. The thread control set is calculated as normal and as a post process, the joint nodes are inserted into the control set. The thread curve is then computed as usual. However, thread segments that occur between the joints are no longer displayed. Examples of images as terminus objects are shown in Figure 9.

4.2. Interweaving

Images also appear as elements that are interwoven into the knot, i.e. threads cross over and under them at appropriate locations. The user is allowed to define a set of *spines* that is associated with each image. Spines are line segments that roughly correspond to images features with which a thread may intersect. The intersection of a spine and thread is com-

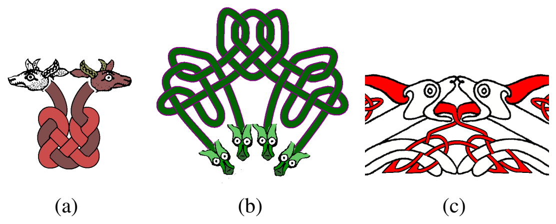


Figure 9: *Images as a-c) terminus and c) interwoven.*

puted after the thread curve has been calculated, but before the overlap order has been derived. Any intersection between a spine and thread causes a change in overlap value at that location. We do not add a node to the thread list at the spine location because spines offer no additional directional data. This has the effect of leaving the curve alone while changing the overlap order. To maintain a well ordered knot, two image crossings or one image crossing and two terminus objects are required.

In order to ensure that the threads overlap the image correctly, the user is optionally allowed to define a set of *forces*. A force is a line segment with which the thread may intersect. Forces roughly correspond with the edges of features that contain spines. A force causes the current thread to immediately switch overlap value at the intersection location. Use of forces is optional and is usually only required for complex interweaving situations. Because forces tend to align with image features, we would like the offset of the inflate function to match those features. Since the thread normal may not match the force direction at the intersection location the inflated thread may overlap the image features improperly, as in Figure 8h. We have added an option that modifies the inflate function and blends the thread normal to match the force direction at the crossing location. This makes the thread appear to cross more naturally where forces have been aligned to image features, as in Figure 8i.

5. Illumination

5.1. Escapes

Often, it is useful to connect separate graphs without physically joining the graphs with a stroke. *Escapes* are user-defined connections between graph nodes. An escape is an arbitrary connection between any two nodes, A and B. When the thread construction algorithm encounters a node A, it immediately adds node B as the next node in the control set and continues from node B. Escapes must also come in pairs. Often, we use escape connections to direct the thread set to cross an image as shown in Figure 11 and the corners of Figure 15.

5.2. Encapsulation

Encapsulation is the concept of creating small knots and using them as pieces with which to make larger knots. This

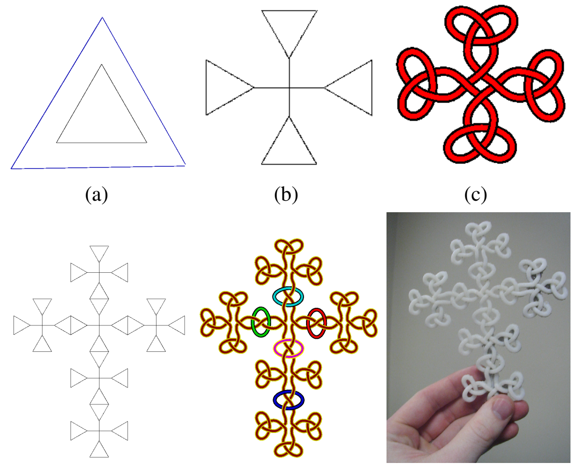

(a) (b) (c) (d) (e) (f)

Figure 10: *The encapsulation process. a) A triangle capsule with bounding volume. b) A larger capsule constructed by connecting 4 triangle capsules. c) The knot resulting from the graph of the capsule in b). d) 6 capsules from b) joined in a high cross pattern. e) The knot resulting from the graph in d). f) The high cross from e) has been fabricated.*

is similar to methods described by Bain [1], Meehan [14] and Mercat [16] for artists. Mercat uses the dual of the graph as the capsule. This is not possible in cases where images are present, undesireable since we want to store style data as well and unecessary since the dual is simply a bounded version of the same knot. In our system, all graph, style, image and connection information is saved to a file. This file defines a *capsule* object. We can read in capsules and perform operations on them such as rotations, transformations, scaling and reflection and change styles en masse. This capsule system is more intuitive than that of Mercat or Meehan because we maintain the original representation of the system and no transformations are required to reconstruct the original graph.

There are two methods of connecting capsules. First, strokes connecting capsule graphs can be drawn in manually by the user, as shown in Figure 10b. Second we can align edge strokes of capsules to create a single border, as shown in Figure 10d. Strokes that overlap one another are merged, resulting in a unified graph. We have created a mouse function which snaps capsules whose edges are close to each other together automatically.

5.3. Template

For creating backgrounds patterns, we have created a template builder that mimics the background styles used in illuminated manuscripts. The guiding principle is the use of thick bands of color to separate sections of knotwork. Our program allows users to define bands of color and their borders with offset Bézier curves whose control points can be

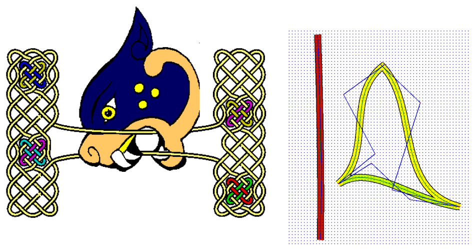

(a) Escape example (b) Template builder

Figure 11:

snapped to a predefined grid. A flood fill tool can be used to fill areas between the borders with a selected color. These background designs function as templates whose interior spaces can be filled with knotwork by drawing strokes or using capsules to fill the space. A fully realized template is shown in Figure 15 (bottom).

6. 3D Knots

We can apply the algorithms presented above to arbitrary 2D-manifold meshes resulting in 3D knots. This works becuse a 2D-manifold mesh can be viewed as locally planar. The original inflation and display algorithms now make no sense since we inflated in 2D. In 3D inflation and display, inflated threads are tubular neighborhoods around the thread displayed using cylinders between points in the thread curves. When we offset the threads based on the overlap order, we use the normal to the mesh at the node position as the offset direction. Node derivative values are in the tangent plane at the midpoint. Examples of knotwork produced using meshes are seen in Figure 15.

7. Results

Graph creation is done in realtime. Users draw strokes and select a *Create Knot* button when finished. Most knots compute at interactive rates, but depending on the complexity of the graph (the combination of effects applied and the density of the samples for each thread curve) a knot can take up to 15 seconds to compute. One of the most important features is the ease of use of our system. User intervention is only really required for graph creation. However, the facility exists for controlling almost every aspect of the system enabling the artist to have as much control over the appearance of the knots as desired. With smoothing enabled, reasonable results are automatically produced 99% of the time though, rarely, a user may need to adjust a smoothing parameter that may be inappropriate for a given graph.

In sets as follows : Figures 13a, c, e, g, Figures 13b, d, f, h, Figures 14a, d, e, and Figures 14 c, g and h, we see variations on themes. This is accomplished by the using capsules to vary repeatable patterns in new and interesting ways, and by changing stylistic options and breakpoints. Because of the utility and ease of use of encapsulation most of these designs took between 1 and 2 minutes to produce. Figure 13i shows a teapot shape embedded in a knot as a thread, demonstrating how logos and other symbols may be incorporated into Celtic knots produced with our system.

The reproductions of art from the Book of Kells (Figures 16 top, inset, 13j, 14j) and the Lindisfarne Gospels (Figure 15) using our system validate the utility of our techniques for reproducing the artwork of the masters of Celtic illumination. These examples mark the first time a computer has accurately reproduced the knotwork from full carpet pages and illuminated letters from the great works of the Celtic illuminated manuscripts. All of the techniques presented were required to create these images and no prior knotwork creation program could have produced them. These results are important because it shows that not only can our system reproduce the complexity of the original works, but we can create new works with the same style and beauty of the originals. Moreover, we can do it much more quickly and can experiment with alternate designs, styles and patterns with almost no additional effort. The only classical patterns we have not been able to reproduce are those that contain images which interweave with each other which is a problem we have not addressed here. The upper border in Figure 16 took about a half hour to make. The images took about 5-10 minutes to mark with an alpha channel and relevant information. Design of such a work proceeds as follows: filling the empty spaces in the images with graphs, making connections from the graphs to the images and finally replicating the two capsules (one for each dog image) and arranging the capsules in a border pattern. The central *T* pattern in the illuminated letter in Figure 16 was drawn freehand in about 5-10 minutes. Style changes to these designs are quick as regions can be selected with the mouse and changes applied en masse.

The zoomorphic borders shown in Figure 16 are after the style of Meehan [14] which in turn take their inspiration and figures from the Book of Kells. Inset (at bottom) is an example of the smoothing process which is vital to achieving visually pleasing results. The smoothest knot is automatically produced when full smoothing is enabled yet we show the steps for instructional purposes. The lower border demonstrates thread width variation to full advantage.

In Figure 12a, a user has designed body art using our program and a temporary tattoo has been applied with henna. In Figure 12b, our system has created a 3 dimensional knot model for a CAD program. The CAD program has defined circular tubes of revolution around the piecewise cubic hermite thread curves. Using the CAD model as an input, a physical 3D model was fabricated using a Stratasys rapid prototyping system, as shown in Figure 12c and Figure 10f. A user can quickly test many designs on the computer, then create physical results for jewelry or ornamentation automatically, a process that used to be extremely time consuming.

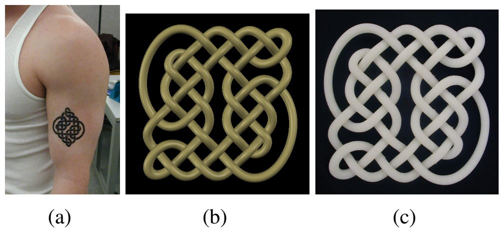


Figure 12: *Uniform design: a) A tattoo design, b) CAD model and c) jewelry created by fabricating the CAD model*

8. Conclusion and Future Work

We have presented techniques for automating the construction of Celtic knotwork and for creating large complex designs with such knotwork. We have introduced novel orientation and smoothing techniques which allow us to compute and display knots in both 2D and 3D, eliminating restrictions of prior work in this area. We have shown how to incorporate images into Celtic knots both as terminus objects and interwoven elements. We have introduced a new method of encapsulation to facilitate the design process and have shown how to construct background templates in the style of the ancient manuscripts. Further, we have shown how to automatically fabricate physical models from such knots.

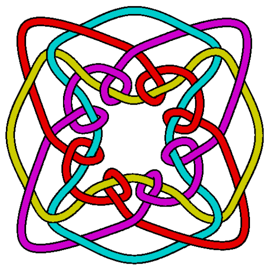
In the future, automatic knot generation using autonomous capsule placement might be a useful feature to explore. Because of the connection to woven materials (the bunny knot in Figure 15 appears crotcheted), cloth simulation and rendering seem a natural extension. Ultimately, we view this system as an artistic playground, where artists, designers and novices alike may quickly and easily explore the creation of advanced Celtic knotwork designs.

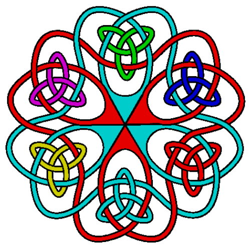
9. Acknowledgements

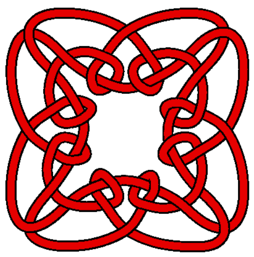
This work was supported in part by ARO (DAAD19-01-1-0013) and NSF (EIA0121533). All opinions, findings, conclusions or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the sponsoring agencies.

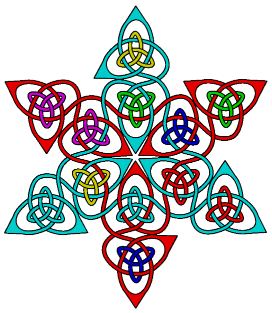
References

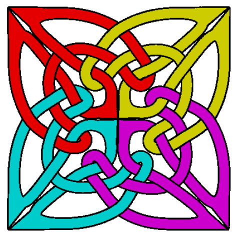
0. ABBOTT, S. Knots3d. Computer Program, www.abbott.demon.co.uk/knots.html. 2
1. BAIN, G. 1951. *Celtic Art, The Methods of Construction*. William MacLellan and Co., Glasgow,Scotland. 2, 8
2. BAIN, I. 1966. *Celtic Knotwork*. Sterling Publishing Co. 2
3. BROWNE, C. 1998 Font Decoration by Automatic Mesh Fitting. *Proceedings of EP-RIDT '98*, 23-43 2
4. E. COHEN, R.F. RIESENFELD, G. E. 2001. *Geometric Modeling With Splines*. AK Peters.
5. GLASSNER, A. 2002. *Andrew Glassner's Other Notebook*. AK Peters. 2
6. GLASSNER, A. November/December 1999. Andrew glassner's notebook : Celtic knotwork, part 2. *IEEE Computer Graphics and Applications*, 78–84. 2
7. GLASSNER, A. September/October 1999. Andrew glassner's notebook : Celtic knotwork, part 1. *IEEE Computer Graphics and Applications*, 78–84. 2
8. GODFREY, R. Celtic knots designer. Computer Program, www.celticdesigner.com. 2
9. GUIONNET, T. Les noeuds celtiques. Computer Program. 2
11. MEEHAN, A. 1991. *Celtic Design: A Beginners Manual*. Thames and Hudson. 2
12. MEEHAN, A. 1991. *Celtic Design: Knotwork, The Secret Method of the Scribes*. Thames and Hudson. 2
13. MEEHAN, A. 1992. *Celtic Design: Illuminated Letters*. Thames and Hudson. 2
14. MEEHAN, A. 1999. *Celtic Borders*. Thames and Hudson. 2, 8, 9
15. MERCAT, C. Celtic knots. Computer program. 2
16. MERCAT, C. 1997. Les entrelacs des enluminures celtes. *Dossier Pour La Science, No.15*. 2, 6, 8
17. MICHAEL T. WONG, DOUGLAS E.ZONGKER, D. S. 1998. Computer generated floral ornament. *Proceedings of SIGGRAPH 98* (August), 423–432. 1
18. SCHAREIN, R. 1998. *Interactive Topological Drawing*. PhD thesis, University of British Columbia. 2
19. SLOSS, A. 1995. *How to Draw Celtic Knotwork: A Practical Handbook*. Brockhampton Press. 2

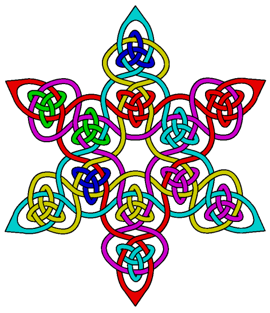

(a)

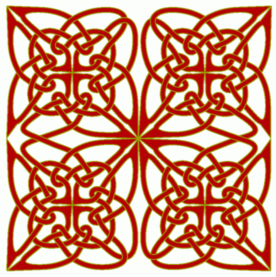

(b)

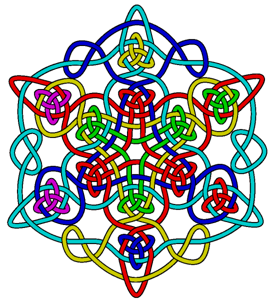

(c)

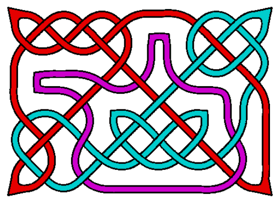

(d)


(e)


(f)


(g)


(h)


(i) Teapot knot

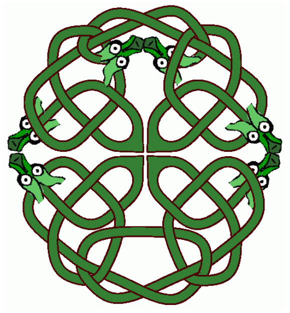
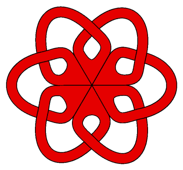
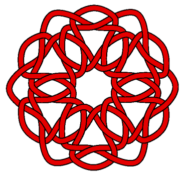
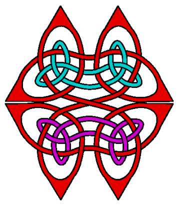
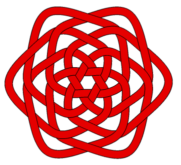

(j) Dragon pattern from the Book of Kells reproduced with our system.

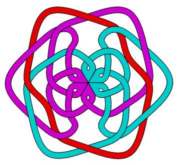
Figure 13: *Examples of Celtic knotwork created with our system.*

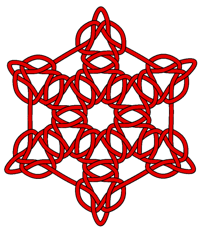

(a)

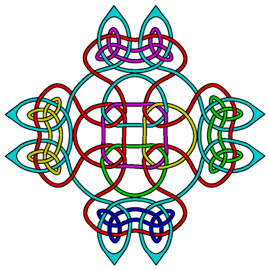

(b)

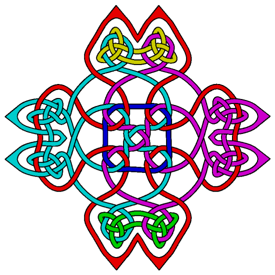

(c)

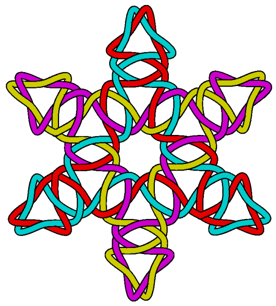

(d)

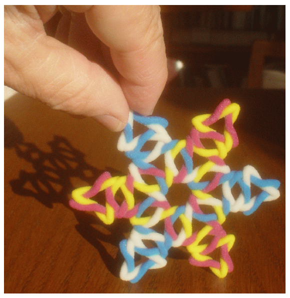

(e)


(f)


(g)


(h)


(i)


(j) physical model of i

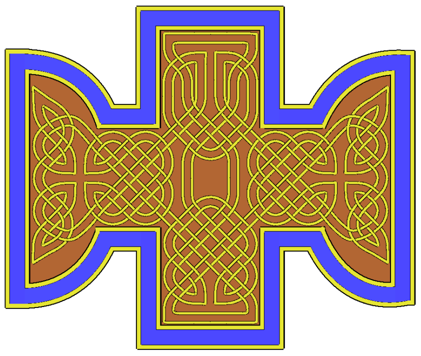

(k) Reproduction of a knot from the Book of Kells using our system.

Figure 14: *Further results from our system.*

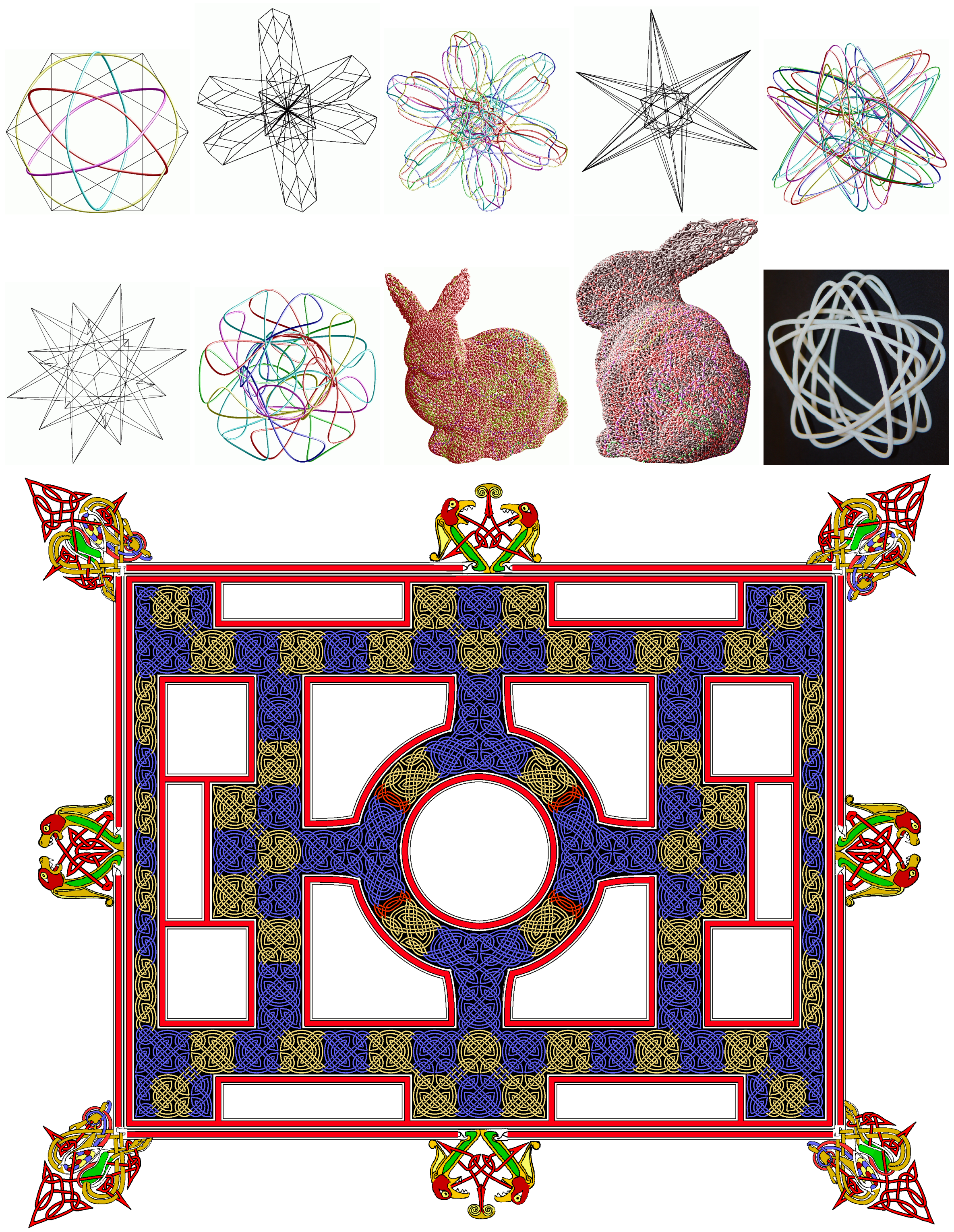


Figure 15: *Top: The knot construction algorithm applied to 3D geometric meshes including a bunny mesh. Also, a physical model of a knot derived from a star shaped mesh. Bottom: Exact reproduction of the great carpet page thread pattern topology introducing the Book of Mark from the Lindisfarne Gospels (ca. 6-7th century AD) produced with our system.*

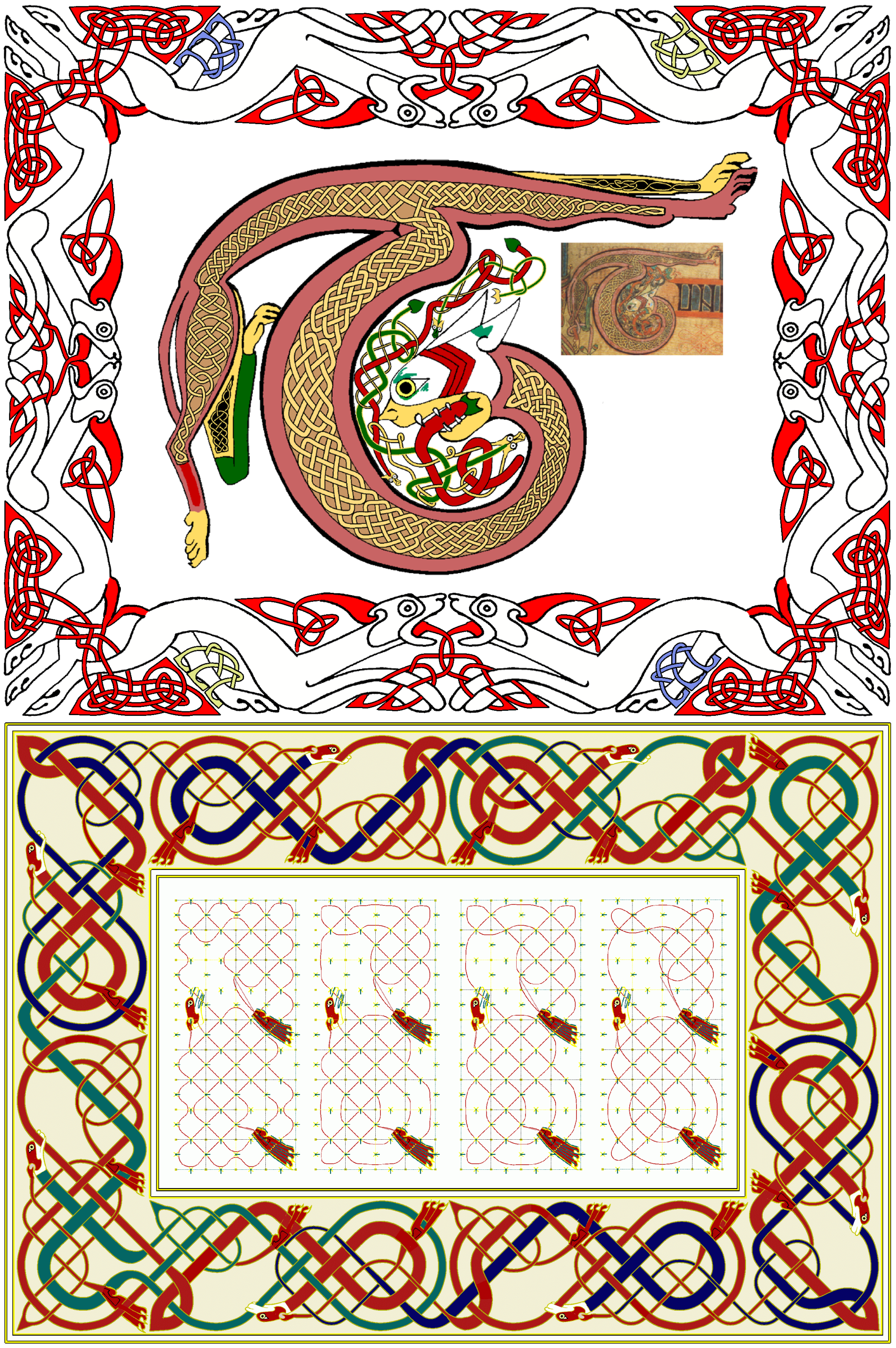


Figure 16: *Top border: A reproduction of a hunting dogs kissing border patterned after Meehan. Top inset: Reproduction of a great illuminated letter design from the Book of Kells: A four legged dragon letter T. Bottom border: The dog eating its own tail border. Our variation on a standard theme that is seen repeatedly in Celtic Art, especially in Meehan. Inset is shown the process of smoothing the threads. From left to right: 1) The default configuration of the threads. Obviously not satisfactory. 2) The threads after appropriate distance values for the meta points have been set. 3) After hermite derivative smoothing has been applied. 4) After Schoenberg smoothing has been applied. 4) is produced automatically when full smoothing is enabled.*