A NEW COMPUTER MODEL FOR ORIGAMI

by

HAMILTON LINK

A THESIS

Presented to the Department of Computer
and Information Science
and the Honors College of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Bachelor of Arts

June 1997

Approved: 

Professor Eugene Luks

An Abstract of the Thesis of

Hamilton E. Link for the degree of Bachelor of Arts

in the Department of Computer and Information Science to be taken June 1997

Title: A NEW COMPUTER MODEL FOR ORIGAMI

Approved: [redacted]

Professor Eugene Luks

There is no general method for modeling folded structures, and in particular the sole example of a program for folding paper is incapable of simulating complex manipulations. The new Ada program described in this report correctly folds mountain and valley folds, as did its predecessor, but it takes an additional step towards becoming a complete model by folding tucks as well. The existing program by Robert Lange was unable to do this, which limits its usefulness, whereas the new model can be used to simulate the folding of many complex traditional origami forms. This report describes a program written for folding origami and explains the structures and methods used. The appendices include the specification and complete implementation of the routines for manipulating the paper's abstract data type, as well as a listing of Ada package specifications for some of the libraries used.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

The age-old art of origami has become a hobby for many people here in the United States. Several computer programs have been made to help people learn to fold certain origami patterns, but only one program could be found that actually allows the user to arbitrarily manipulate a virtual piece of paper (Lange). This program appears to be too limited to fold origami forms other than simple paper airplanes. After reviewing some of the mathematical studies of origami, I developed a data structure with the functionality needed to create many complex traditional origami designs. In addition, I proposed a new representation that significantly advances the existing methods. This new representation appears to provide a suitable underlying model upon which a fully functional paper folding program may be built. The central problem in both the old and new model is that they impose order upon the folded structure that is not always present in the real world, and in doing so they destroy the ideal 1:1 relationship between forms in the real world and structures in the modeled world. This creates problems with verification, which became apparent as I tried to

create a more functional system based in part on my understanding of the old. Although I was able to extend the functionality of the older model to a more reasonable level, the implementation I had chosen was inherently limited. This led me to the idea of using linked structures to represent overlap relationships. Although the model has not been extended to include this concept, it promises to remove the ambiguity in the model, making verification easier as well as removing fold-ordering dependence.

Although little has previously been done with origami in computer science, it has been of interest to the mathematics community for quite some time. Various investigators have analyzed the problem of determining the foldability of crease patterns (Hull, 1994 and 1996) and the nature of origami tessellation, the regular repetition of a fold pattern across a piece of paper (Kawasaki, 1988). Others have discussed regularities in the crease patterns of complex objects (Engel, 1989) and proposed a set of origami "primitives," fundamental operations from which all possible geometric constructions could be derived. Although it does not seem possible to incorporate any of these results in the current model, some may prove useful in the future.

Origami as a computer problem has several features that can be seen individually in other computer science problems. As an example of modeling foldable structures, origami models are of practical interest. Folding sheet metal, for example, is important for some manufacturing processes. Although currently it is almost certainly done based on application-specific knowledge of the folds that are needed and can be made, such examples of experimental knowledge provide little guidance when attempting to solve related problems. Collapsible structures are useful in many applications where portability or efficient storage are needed. The virtual paper in my model is infinitely thin, and of course does not completely represent reality in many other ways -- it is not flammable and cannot bend or tear -- but it represents a useful simplification of a structure that can be arbitrarily folded. In principle, several of the methods used in this program could be used in folding other objects.

ORIGAMIMANIP (Lange)

Pre-filmed origami-teaching programs, of which there are several, and the Mathematica tools by Ludmila Zamiatina (Zamiatina, 1994), all use predefined folding patterns. The user is bound to a predefined sequence of folds, instead of being free to fold the simulated piece of paper in any possible way. Only one program that allows the user this freedom could be found. This program, Origamimanip, was written for the Macintosh by Robert Lange. Origamimanip does not provide a model capable of folding complex origami, but within certain limits the user has freedom to manipulate the model in any manner. Origamimanip is thus a reasonable attempt at creating a useful origami model, but it is in fact not developed enough to be useful. Its first problem is immediately apparent when the enthusiast tries to fold a traditional crane: Origamimanip can only make book folds (Fig. 1), it can neither tuck nor invert (Fig. 2) nor make press folds (Fig. 3). None of the traditional origami bases, such as the crane base (e.g., Fig. 4), can be created. Very few origami can be folded with this limitation.

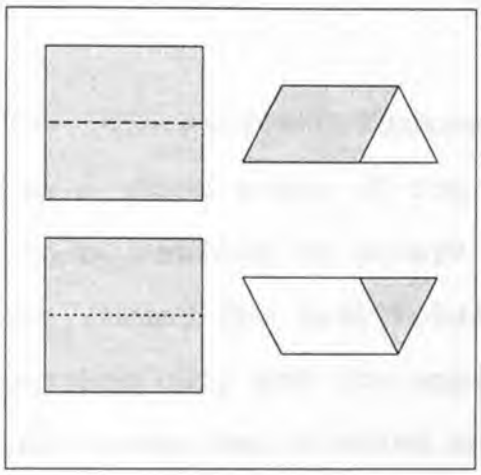

Fig. 1: Book Folds

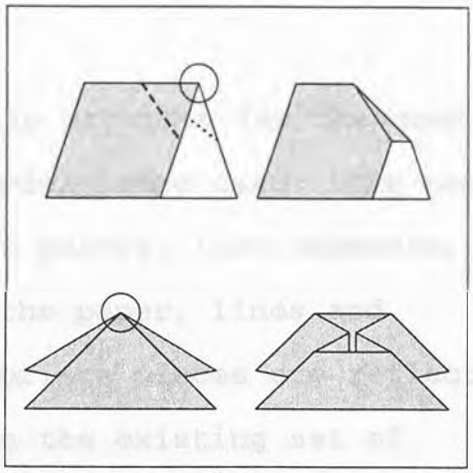

Fig. 2: Tuck and Invert

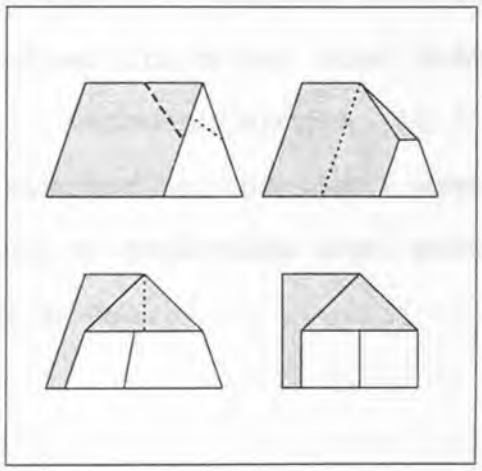

Fig. 3: Simple Press

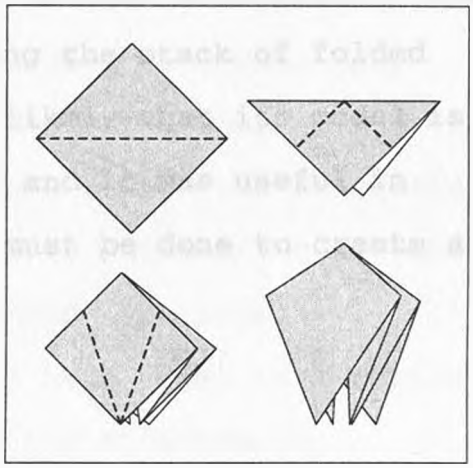

Fig. 4: Crane Base

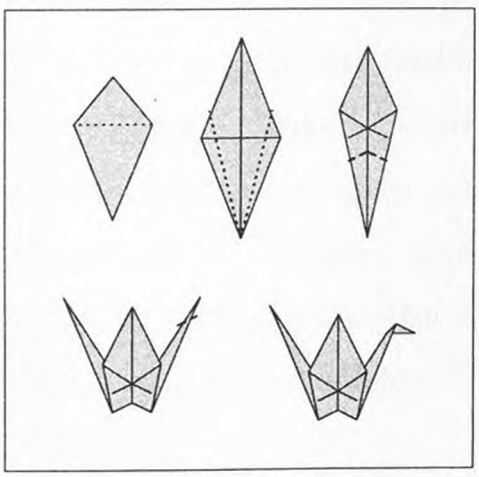

Fig. 5: Crane (from 4)

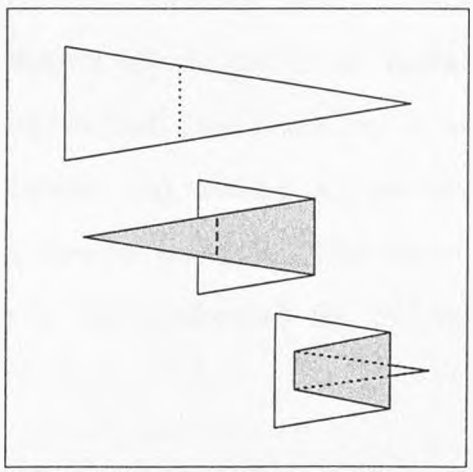

Fig. 6: Impossible Fold

The information Origamimanip provides for the user supports a rough guess of the model Lange used. The paper seems to be defined by arrays of points, line segments, and polygons. When the user folds the paper, lines and polygons are cut, and the appropriate pieces are reflected over the crease and stacked upon the existing set of polygons. In order to replicate two- sided origami paper, each polygon is marked with its up/down orientation, which determines its color when drawing the stack of folded paper. Although simple, it is likely that its model is very similar to the one I used, and it was useful in helping me determine what more must be done to create a useful model.

DISCUSSION AND RESULTS

A useful model may not need to be complete as long as it captures the most important characteristics of the system in question. In this case, the model must allow at least some origami shapes to be made. Adding tuck folds is a critical step that enables the user to create many traditional origami patterns. The traditional crane (Fig. 5) is one example of the many models that can be created using only book and tuck folds. No interesting patterns use only book folds, so by extending the model's functionality to include tucks I have taken an important step beyond the previously existing solution in Origamimanip.

I attempted to create a list of folding primitives and, if possible, to determine which ones could be made from the others. This process revealed that making a model that would support tucking and inverting would allow the user to make many origami shapes, even some of the more complex ones. The problems that I encountered in folding suggested several questions:

How do I tuck and invert without folding the entire stack, but rather by limiting each fold to only those layers which must move to make the fold?

How do I check for obstructions for a fold, to see if the fold is possible?

How do I make a model that is not only unambiguous, but has a unique representation for a particular origami?

How do I make a model that will easily be extended into three-dimensional origami, and should I?

Solutions or potential solutions to most of these problems are explained below.

In preparing to create a new model, I first created several potential folding algorithms, in order to better understand what information the model must capture. This section describes three of the landmark cases. The package containing the implementation of the paper data type and the code for folding it are in Appendix A, and the library packages used are listed in Appendix B.

Method I: Naive Stack, Global Naive Tuck

The most naive representation of a folded piece of paper is as a stack of smaller sheets, which in this case are always convex polygons. No attention is paid to the connections between layers (i.e., the folds proper), and

with every fold every layer overlapping the crease is folded. Although this representation allows some patterns to be made (with great difficulty), a model based on it oversimplifies the concept of the paper and is limited and very difficult to use as a result. The method for folding such a representation is shown in the following pseudocode:

```
given a stack, fold, and layer at which to tuck:

in reverse through the layers above the insertion point for the tuck,
      if the layer crosses the fold line, then
            cut the polygon
            reflect the portion being folded over
            insert this new polygon into the stack below
                  those polygons that have already been folded
      end if
end loop
in reverse through the layers below the insertion point for the tuck,
      duplicate the above process for each polygon crossing the crease
end loop
```

Method II: Connected Stack, Restricted Naive Tuck

A second, more sophisticated representation of a stack of paper considers the physical connections of edges and vertices, making it possible to restrict the folding process somewhat. Instead of folding every surface that intersects the fold edge, it is possible to select a starting point for the fold (in this example, a vertex) and follow the physical connections until they cross the fold line in order to determine what should be modified. Although this sometimes does not capture nested layers, a

connection-based fold replicates the real process of folding much more closely than the naive stack fold, allowing much more natural use of the model. This model is the basis for the code listed in Appendix A. The pseudocode for the model follows:

```
mark all objects as ones not to be moved
starting with the selected point
      follow the connections until they cross the fold line,
            marking everything as an object to be moved
for every marked object,
      if the object does not cross the fold line,
            reflect the object over the line
            do not modify its links
      otherwise,
            cut the object with the fold line
            reflect the half of the object being moved
            add this half of the object to the stack
            correct the physical connections of the object's halves
      end if
```

The folding procedure thus makes several passes through the data structure representing the paper. Initially each vertex, edge, and surface that must be modified is marked, either to be cut or to be moved. Then each object that must be cut is split at the fold line, and new vertices, edges, and surfaces are created. The new objects are also marked if they must be moved. Finally, the objects that must be moved are reflected over the fold line and inserted at the proper place within the stack.

Method III: Connected Stack, Obstructed Folding

A third algorithm is nearly identical to the connected folding method because it is a manipulation of nearly the same structure, but it differs in detecting obstructions and folding them along with the layers that are the primary concern. Unfortunately, determining whether a piece of the paper obstructs the fold is very difficult with the information provided by the fairly straightforward paper connections. This is due in part to the non-uniqueness of the representation. In order to more efficiently determine which layers interfere with the desired change, it is helpful to keep track of a new order in the stack, based on which layers overlap each other. This information not only facilitates the identification of obstructions, but also makes it possible to test the structure of a stack and restrict the model to valid states. This, the most sophisticated of the algorithms I developed, is represented by the following pseudocode:

```
mark every object as being undisturbed
starting with the selected point
      follow the connections until they cross the fold line,
            marking everything as an object to be moved
for each layer that is blocked by an unmarked layer,
      use the obstructing layer as a starting point,
            and mark the stack using the same method as above
for every marked object,
      if the object does not cross the fold line,
            reflect the object over the line
            do not modify its links
```

```
otherwise,
      cut the object with the fold line
      reflect the half of the object being moved
      add this half of the object to the stack
      correct the physical connections of the object's halves
end if
```

In this future model, the computational aspects of the tucking process are the same as described in method II above, but insertion into the stack is based on the artificial linearization of the overlap hierarchy tree.

Each of the above methods is significantly more developed than the previous one, and all are more capable than Origamimanip's model. The naive stack and tuck model allows the user to create a few interesting shapes, but very few objects can be folded by manipulating the entire stack at each step. The connected stack, obstructed folding model could potentially solve this problem by folding obstructing layers along with the connected layers, and checking for the realism of the final state; however, updating the overlap hierarchy properly is a difficult problem that remains unsolved. The connected stack, restricted folding model solves the problems of the first model by limiting the fold to those layers that are forced to move because of their physical association with the primary object in motion. With this process, many new fold patterns become possible. The main problems are that

impossible states may still be reached (Fig. 6, p. 5) and that the model's nonunique representation makes certain states possible only when particular representations of identical states are used (Fig. 7). Nevertheless this solution, while still inconvenient in many ways, is comprehensive enough that it can be used to model many common origami patterns. With the extension of the algorithm to allow inversions and reverse tucks, it could be made to do almost anything which can be done with real paper (e.g., Fig. 8).

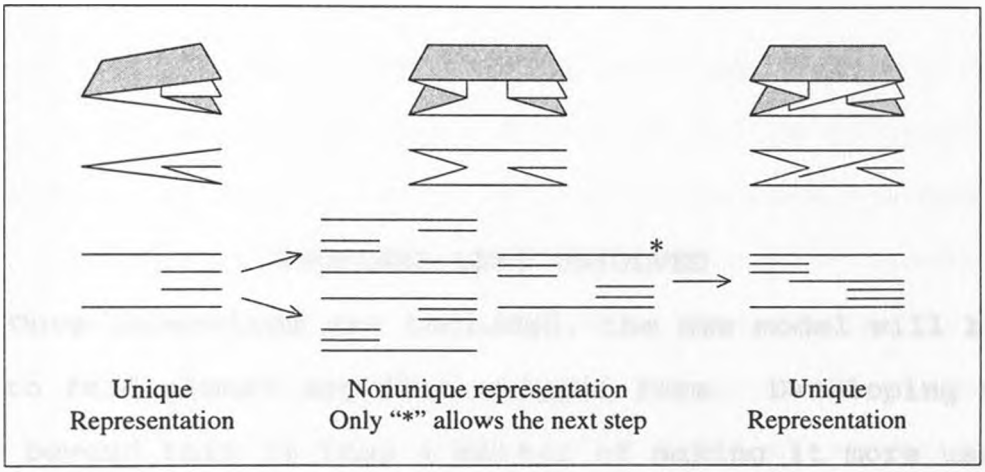

Fig. 7: The Problem of False Order

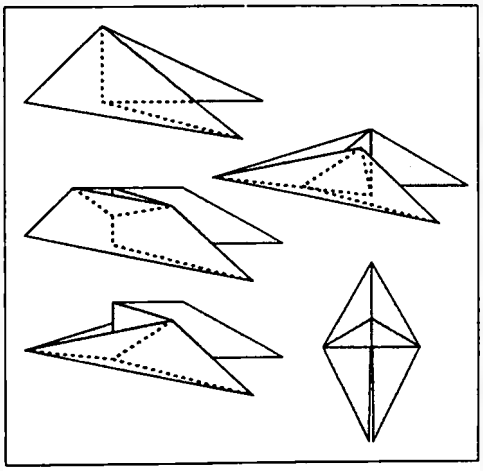

Fig. 8: Composite Fold

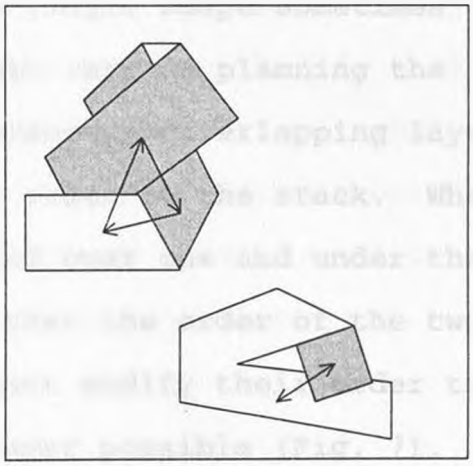

Fig. 9: Circular Overlap

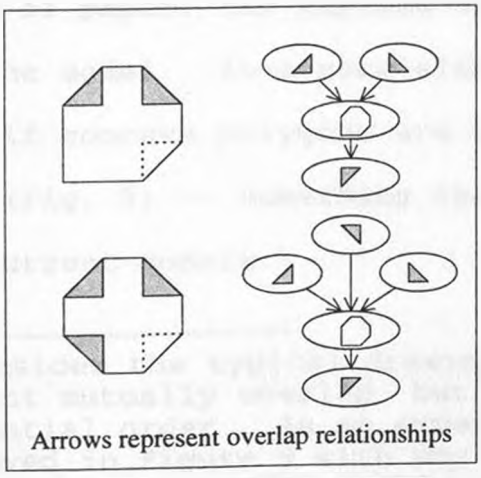

Fig. 10: Overlap Hierarchy

PROBLEMS LEFT UNSOLVED

Once inversions are included, the new model will be able to fold almost any flat origami form. Developing the model beyond this is thus a matter of making it more usable rather than more useful. Currently, the nonuniqueness of the model's representation of a single shape sometimes requires that the user take great care in planning the order of the folds. This is because nonoverlapping layers must still be given an explicit order in the stack. When another layer must then be folded over one and under the other, the model does not know that the order of the two layers is arbitrary, and so cannot modify their order to make the insertion of the new layer possible (Fig. 7). Thus even if the new fold would be possible with a physical piece of paper, the imposed order may make it impossible for the model. In a more elaborate case three layers (or two, if concave polygons are allowed) may each overlap the next (Fig. 9) -- something that cannot be represented with the current models.[1]

---

[1] Consider the typical drawing program. In most, polygons may not mutually overlap, but instead are given a sequential order. As an experiment, try making the forms diplayed in Figure 9 with any drawing program without cropping at least one polygon by hand.

The nonuniqueness and order-based limitations of the current model also introduces a problem in the folding algorithms. Without a 1:1 relationship between the real world and the model, it becomes very difficult to ensure that a particular state is realistic. This problem is partially resolved with the connected obstructed folding method, but this does not address the issue of structural impossibilities. This can lead to anomalous fourth-dimensional folds, for which the paper must pass through itself at a crease (since surfaces all lie flat in these models, they never pass through one another). A model having a 1:1 correspondence with the real world coupled with heuristics to test the realism of a state must both be developed to create a system that is unbreakable and easy to use.

In order to resolve the problem of nonunique representations, it is natural to ask what information the system lacks that makes the model inexact. In this case the problem is not missing data, but rather an unsuitable representation of the data in the model. The problem is that adjacent layers are stored in a way that imposes a fixed order where none exists (Fig. 7). The model does not lack information, but rather the model creates more information than is necessary and cannot identify this

misleading data when making later decisions. Rather than an array of sequentially ordered layers, a new means of representing the stack order must be found which does not associate layers that are not dependent upon one another. Once this is done, it will be possible to address the issue of identifying possible folds.

FUTURE DEVELOPMENT

Before attempting to implement state tests, future work on an origami model will have to implement a new method of storing layers that does not impose a false order on the system. This may best be done by representing overlap relationships with links from each layer to the set of layers immediately above and below it (Fig. 10, p. 14). The arbitrary order imposed upon non- overlapping layers by representing the stack as an array of layers will be removed, and the model will have a unique representation of any real flat origami.

Once such a representation has been used to replace or augment the existing model, the problem becomes one of implementing the algorithms for folding the augmented structure. Linked structures are more complicated to manipulate, because at no time does the procedure have complete knowledge of the structure. If the overlap hierarchy is built on top of the existing array structure, much of the existing code could be used to determine what layers are affected by a fold, and the hierarchy could be used to test folds and determine an order in the array after each fold was made. The system may still represent

reality in more than one way, but the tree structure allows the falsely ordered layers to be seen for what they are and rearranged as needed during the folding process. Although this has not been implemented, this design corrects for the deficiencies of the current model by extending it rather than replacing it. Thus, its implementation should be straightforward.

SUMMARY

A practical model for simulating origami has been developed and implemented in Ada even though closely related problems like determining the foldability of a crease pattern have not been solved. An origami program might have practical applications in industry, for people who lack the intricate coordination required for folding paper, or for professionals who wish to diagram new patterns more easily. Origamimanip's simple book folds cannot create complex layering in the paper. With a small extension of the solution to include stack-wide tucking (as in Method I), very simple objects could be created, but the lack of any restrictions to the fold limits the usefulness even in this case.

In many cases, it is taken for granted that arrays or linked nodes should be used as a foundation when developing a program to solve a particular problem. In this situation, however, the best solution was not the obvious one. What appeared to provide a suitable structure and reasonable means of manipulation (the connected folding methods) was revealed to be too limiting to provide a complete solution. Only by reevaluating the nature of the

problem once the failures of the old model had been observed was a more appropriate model found. For my experiment, I implemented a physically restricted algorithm that gave me the capability to fold many origami patterns while preserving the natural connected representation of the paper. Origamimanip clearly contains information about the physical connections among the objects, because it is possible to restrict folds to the top layers if deeper layers are not connected. The new model uses this concept to its full potential, and it uses the connectivity to fold only the things that are pulled by their association with the point being pulled by the user of the system. With the new connected tucking scheme, the model is not a stack of unrelated polygons, but rather a set of interconnected layers of a whole piece of paper.

With the more natural process of folding now implemented, it is not difficult to begin looking for patterns that may be folded. The most widely known origami pattern may very well be the traditional crane, which is possible to fold only with book folds and tucks. With combinations of tucks and book folds, simple press folds may also be made (Fig. 3). A method for inverting corners should not be difficult to implement. I was unable to find any flat origami that could not be folded if inversions were also allowed. Although at this time the code does not

include a user interface, the model is now developed enough that an interface will not be difficult to create as I work towards a complete program for folding origami.

How this new model might be extended to support three-dimensional origami is not clear. For now, many aspects of the problem of flat origami have not been addressed even after an implementation of the overlapping-layer model has been created. Creating a user interface that allows a user to manipulate the paper as easily as if they were folding a real piece of paper will also be a substantial project. Efficiency issues might also be addressed, along with making the paper have real properties such as thickness and stiffness, both of which are considerations when folding real paper. The model is now developed enough that an interface may be developed in order to facilitate further work towards creating a complete program for folding origami.

APPENDIX A

GLOSSARY

Origami: A folded sheet of paper, and the ordered set of folds which creates it. If an origami can be laid flat after each fold has been made, it is may also be called flat or two-dimensional. If an origami may not be laid after any fold has been made during it's construction, it is called three-dimensional.

Atomic Fold: A manipulation of a sheet that can not be achieved as a series of simpler folds.

Composite Fold: A manipulation of a sheet that can be achieved as a series of simpler folds.

Book Fold: A manipulation that folds one portion of a sheet over the remainder along a straight line. These are traditionally called mountain and valley folds.

Tuck (1): In traditional origami, a manipulation that folds one portion of a folded sheet between two layers of the remainder along a single fold line. Note that some, but not all, tucks may be composed of book folds.

Tuck (2): In my program, the computational procedure that simulates this manipulation through the process of marking, cutting, and reflecting parts of the data structure across a fold line. Book folds are thus a special case of tucks.

Inversion: A manipulation of a folded sheet that reverses the orientation of each edge radiating from a vertex, out to a closed path of edges surrounding that point. This has the effect of turning the vertex's corner inside out, and folding each layer involved up against itself.

Press: Typically, a manipulation that involves a folded portion of the sheet being raised, opened, and pressed flat. Many presses may be composed of inversions or tucks, but these all may be considered as atomic folds for the sake of determining the dimensions of an origami.

APPENDIX B

PAPER ADT IMPLEMENTATION

```
with Ada.Finalization; use Ada.Finalization;

with points; use points;
with lines; use lines;
with polygons; use polygons;

package papers is
     type paper is private;

     Offsides : exception;

     procedure Put(stack : paper);

     function newSheet return paper;
     procedure fold(stack : in out paper; p : point;
                    fold_edge : line; layer : natural);
private
     type flag is (undisturbed, cut_me, move_me, modified);

     type linked_point;
     type linked_line_segment;
     type linked_polygon;

     type point_access is access linked_point;
     type line_access is access linked_line_segment;
     type polygon_access is access linked_polygon;

     type point_set is array (Integer range <>) of point_access;
     type line_set is array (Integer range <>) of line_access;
     type polygon_set is array (Integer range <>) of polygon_access;

     type pointset_access is access point_set;
     type lineset_access is access line_set;
     type polyset_access is access polygon_set;

     type linked_point is record
          p : point;
          state : flag := undisturbed;

          lineset : lineset_access;
          polyset : polyset_access;
     end record;

     type linked_line_segment is record
          l : line_segment;
          state : flag := undisturbed;

          pointset : pointset_access;
          polyset : polyset_access;
     end record;
```

```
     type linked_polygon is record
          A : polygon;
          state : flag := undisturbed;

          pointset : pointset_access;
          lineset : lineset_access;
     end record;

     type paper is record
          n_pts, n_lines, n_polys : Integer;
          pointset : pointset_access;
          lineset : lineset_access;
          polyset : polyset_access;
     end record;
end papers;

with Ada.Finalization; use Ada.Finalization;
with Ada.Unchecked_Deallocation;
with Text_IO;

with points; use points;
with lines; use lines;
with polygons; use polygons;
with shape_math; use shape_math;
with display; use display;

package body papers is
     ------------------------------------------------------------------
     -- methods for managing memory used by linked structures
     ------------------------------------------------------------------

     procedure Free_pointset is
          new Ada.Unchecked_Deallocation(point_set, pointset_access);

     procedure Free_lineset is
          new Ada.Unchecked_Deallocation(line_set, lineset_access);

     procedure Free_polyset is
          new Ada.Unchecked_Deallocation(polygon_set, polyset_access);

     procedure Free_polygon is
          new Ada.Unchecked_Deallocation(linked_polygon, polygon_access);

     ------------------------------------------------------------------
     -- information extractors for paper:
     --
     -- numpoints, numlines, & numlayers(stack) return # of appropriate unit
     -- coner, crease, & layer(stack, i) return the ith approprate unit
     -- nearest_point(in, pt, on_side, of_line) returns the nearest point
     --   on this side of the line
     ------------------------------------------------------------------

     function numpoints(stack : paper) return Integer is
     begin
```

```
     return stack.pointset'Length;
end numpoints;

function numlines(stack : paper) return Integer is
begin
     return stack.lineset'Length;
end numlines;

function numlayers(stack : paper) return Integer is
begin
     return stack.polyset'Length;
end numlayers;

function corner(stack : paper; i : Integer) return point_access is
begin
     return stack.pointset(stack.pointset'First + i - 1);
end corner;

function crease(stack : paper; i : Integer) return line_access is
begin
     return stack.lineset(stack.lineset'First + i - 1);
end crease;

function layer(stack : paper; i : Integer) return polygon_access is
begin
     return stack.polyset(stack.polyset'First + i - 1);
end layer;

function nearest_point(in_stack : paper;
               pt : point;
               on_side : direction;
               of_line : line) return point_access is
     best, try : point_access;
begin
     best := corner(in_stack, 1);
     for i in 2 .. numpoints(in_stack) loop
          try := corner(in_stack, i);
          if Distance(pt, best.p) > Distance(pt, try.p) and
             onsides(try.p, on_side, of_line) then
               best := try;
          end if;
     end loop;

     if onsides(best.p, on_side, of_line) then
          return best;
     end if;
     raise Offsides;
end nearest_point;

-------------------------------------------------------------------
-- paper display routine
-------------------------------------------------------------------

procedure Put(stack : paper) is
     package T renames Text_IO;
```

```
        package Int_IO is new T.Integer_IO(Integer); use Int_IO;
    begin
        T.Put("(paper ");Put(stack.n_polys, 0);
        for i in 1 .. stack.n_polys loop
            T.New_Line;
            T.Put("  "); Put(layer(stack, i).A);
--          for j in 1 .. stack.polyset(i).pointset'Length loop
--              T.New_Line; T.Put("    ");
--              Put(stack.polyset(i).pointset(j).p);
--              for k in 1 ..
--                  stack.polyset(i).pointset(j).lineset'Length loop
--                  T.New_Line; T.Put("      ");
--                  Put(stack.polyset(i).pointset(j).lineset(k).l);
--              end loop;
--              for k in 1 ..
--                  stack.polyset(i).pointset(j).polyset'Length loop
--                  T.New_Line; T.Put("      ");
--                  Put(stack.polyset(i).pointset(j).polyset(k).A);
--              end loop;
--          end loop;
--          for j in 1 .. stack.polyset(i).lineset'Length loop
--              T.New_Line; T.Put("    ");
--              Put(stack.polyset(i).lineset(j).l);
--              for k in 1 ..
--                  stack.polyset(i).lineset(j).pointset'Length loop
--                  T.New_Line; T.Put("      ");
--                  Put(stack.polyset(i).lineset(j).pointset(k).p);
--              end loop;
--              for k in 1 ..
--                  stack.polyset(i).lineset(j).polyset'Length loop
--                  T.New_Line; T.Put("      ");
--                  Put(stack.polyset(i).lineset(j).polyset(k).A);
--              end loop;
--          end loop;
        end loop;
        T.Put_Line(")");
    end Put;

    ------------------------------------------------------------------
    -- paper folding routines
    --
    -- clear flags, mark for change, pull, and relink
    ------------------------------------------------------------------

    procedure clear_flags(stack : paper) is
    begin
        for i in 1 .. numpoints(stack) loop
            stack.pointset(stack.pointset'First + i -1).state:= undisturbed;
        end loop;
        for i in 1 .. numlines(stack) loop
            stack.lineset(stack.lineset'First + i - 1).state := undisturbed;
        end loop;
        for i in 1 .. numlayers(stack) loop
            stack.polyset(stack.polyset'First + i - 1).state := undisturbed;
        end loop;
```

```
end clear_flags;

-----------------------------------------------------------------------

-- if not already marked, and on the correct side or spanning the line,
-- mark this and recurse.  Note that this probably does more work than
-- is really necessary, because points, and on the wrong side of the
-- line will be checked for onside-ness, but the markings should always
-- stop at the correct point, with any entities crossing the line
procedure mark(this : line_access; side : direction; of_line : line);
procedure mark(this : polygon_access; side : direction; of_line : line);

procedure mark(this : point_access; side : direction; of_line : line) is
begin
     if (this.state = undisturbed) and onsides(this.p, side, of_line) then
          this.state := move_me;
          for i in 1 .. this.lineset'Length loop
               mark(this.lineset(this.lineset'First + i - 1),
                    side, of_line);
          end loop;
          for i in 1 .. this.polyset'Length loop
               mark(this.polyset(this.polyset'First + i - 1),
                    side, of_line);
          end loop;
     end if;
end mark;

procedure mark(this : line_access; side : direction; of_line : line) is
begin
     if (this.state = undisturbed) and onsides(this.l, side, of_line) then
          if (Intersect(this.l, of_line)
          and not On(A(this.l), of_line)
          and not On(B(this.l), of_line)) then
               this.state := cut_me;
          else
               this.state := move_me;
          end if;
          for i in 1 .. this.pointset'Length loop
               mark(this.pointset(this.pointset'First + i - 1),
                    side, of_line);
          end loop;
          for i in 1 .. this.polyset'Length loop
               mark(this.polyset(this.polyset'First + i - 1),
                    side, of_line);
          end loop;
     end if;
end mark;

procedure mark(this : polygon_access; side : direction; of_line : line) is
begin
     if (this.state = undisturbed) and onsides(this.A, side, of_line) then
          if Intersect(this.A, of_line) then
               this.state := cut_me;
          else
               this.state := move_me;
```

```
          end if;
          for i in 1 .. this.pointset'Length loop
               mark(this.pointset(this.pointset'First + i - 1),
                    side, of_line);
          end loop;
          for i in 1 .. this.lineset'Length loop
               mark(this.lineset(this.lineset'First + i - 1),
                    side, of_line);
          end loop;
     end if;
end mark;

-- mark everything connected to the nearest point, staying on this side,
-- as an entity to be modified in the cut-&-reflect pass
procedure mark_everything(stack : paper; side : point; l : line) is
     start_point : point_access;
begin
     if Left(side, l) then
          start_point := nearest_point(stack, side, leftside, l);
          mark(start_point, leftside, l);
     elsif Right(side, l) then
          start_point := nearest_point(stack, side, rightside, l);
          mark(start_point, rightside, l);
     end if;
end mark_everything;

-------------------------------------------------------------------------

-- count the number of new points and line segments that need to
-- be created, equal to the number of line segments and polygons
-- that cross the fold edge
function count_new_points(stack : paper) return Integer is
     segment : line_access;
     newpoints : Integer := 0;
begin
     for i in 1 .. numlines(stack) loop
          segment := stack.lineset(stack.lineset'First + i - 1);
          if (segment.state = cut_me) then
               newpoints := newpoints + 1;
          end if;
     end loop;
     return newpoints;
end count_new_points;

function count_new_polys(stack : paper) return Integer is
     poly : polygon_access;
     newlines : Integer := 0;
begin
     for i in 1 .. numlayers(stack) loop
          poly := stack.polyset(stack.polyset'First + i - 1);
          if (poly.state = cut_me) then
               newlines := newlines + 1;
          end if;
     end loop;
     return newlines;
```

```
end count_new_polys;

procedure prepare_stack(stack : in out paper; layer : Natural) is
     new_stack : paper;
     newpoints, newlines, newpolys : Integer := 0;
begin
     newpoints := count_new_points(stack);
     newpolys := count_new_polys(stack);
     newlines := newpoints + newpolys;
     new_stack.pointset := new point_set(1 .. numpoints(stack)+newpoints);
     new_stack.lineset := new line_set(1 .. numlines(stack)+newlines);
     new_stack.polyset := new polygon_set(1 .. numlayers(stack)+newpolys);

     -- copy old data into the new
     for i in 1 .. numpoints(stack) loop
          new_stack.pointset(i) := stack.pointset(i);
     end loop;
     for i in 1 .. numlines(stack) loop
          new_stack.lineset(i) := stack.lineset(i);
     end loop;
     for i in 1 .. layer loop
          new_stack.polyset(i) := stack.polyset(i);
     end loop;
     for i in layer+1 .. numlayers(stack) loop
          new_stack.polyset(i+newpolys) := stack.polyset(i);
     end loop;

     Free_pointset(stack.pointset);
     Free_lineset(stack.lineset); -- free up old sets of pointers
     Free_polyset(stack.polyset);
     new_stack.n_pts := stack.n_pts;
     new_stack.n_lines := stack.n_lines;
     new_stack.n_polys := stack.n_polys;

     stack := new_stack; -- copy access variables
end prepare_stack;

-----------------------------------------------------------------------

-- add the link arbitrarily, and add the point to the poly's datum
procedure addpoint(to_this : polygon_access;
                   newp, between_this, and_this : point_access) is
     oldpointset : pointset_access := to_this.pointset;
     newpointset : pointset_access
                 := new point_set(1 .. oldpointset'Length+1);
begin
     insertpoint(to_this.A, newp.p, between_this.p, and_this.p);

     for i in 1 .. oldpointset'Length loop
          newpointset(i) := oldpointset(i);
     end loop;
     newpointset(oldpointset'Length+1) := newp;

     to_this.pointset := newpointset;
     Free_pointset(oldpointset);
```

```
exception
     when others => Text_IO.Put_Line("addpoint"); raise;
end addpoint;

-- since lines aren't part of the polygon data, just add the link
procedure addline(to_this : polygon_access; newl : line_access) is
     oldlineset : lineset_access := to_this.lineset;
     newlineset : lineset_access := new line_set(1 .. oldlineset'Length+1);
begin
     for i in 1 .. oldlineset'Length loop
          newlineset(i) := oldlineset(i);
     end loop;
     newlineset(oldlineset'Length+1) := newl;

     to_this.lineset := newlineset;
     Free_lineset(oldlineset);
end addline;

procedure addline(to_this : point_access; newl : line_access) is
     oldlineset : lineset_access := to_this.lineset;
     newlineset : lineset_access := new line_set(1 .. oldlineset'Length+1);
begin
     for i in 1 .. oldlineset'Length loop
          newlineset(i) := oldlineset(i);
     end loop;
     newlineset(oldlineset'Length+1) := newl;

     to_this.lineset := newlineset;
     Free_lineset(oldlineset);
end addline;

procedure addpoly(to_this : point_access; new_poly : polygon_access) is
     oldpolyset : polyset_access := to_this.polyset;
     newpolyset : polyset_access
                := new polygon_set(1 .. oldpolyset'Length+1);
begin
     for i in 1 .. oldpolyset'Length loop
          newpolyset(i) := oldpolyset(i);
     end loop;
     newpolyset(oldpolyset'Length+1) := new_poly;

     to_this.polyset := newpolyset;
     Free_polyset(oldpolyset);
end addpoly;

procedure replace(in_this : point_access;
                  this, with_this : polygon_access) is
begin
     for i in 1 .. in_this.polyset'Length loop
          if in_this.polyset(i) = this then
               in_this.polyset(i) := with_this;
               exit;
          end if;
     end loop;
end replace;
```

```
procedure replace(in_this : line_access;
                  this, with_this : polygon_access) is
begin
     for i in 1 .. in_this.polyset'Length loop
          if in_this.polyset(i) = this then
               in_this.polyset(i) := with_this;
               exit;
          end if;
     end loop;
end replace;

-- adds a point to a line, fixes all the links, and updates the polygons
-- on either side (will temporarily have 3 colinear points, but its ok)
procedure cut_line(stack : in out paper; this : line_access;
                   side : direction; edge : line) is
     left_endpoint, right_endpoint, new_point : point_access;
     left_line, right_line, new_line : line_access;
begin
     stack.n_pts := stack.n_pts + 1;
     stack.n_lines := stack.n_lines + 1;
     stack.pointset(stack.n_pts) := new linked_point;
     stack.lineset(stack.n_lines) := new linked_line_segment;

     new_point := stack.pointset(stack.n_pts);
     new_line := stack.lineset(stack.n_lines);

     if Left(this.pointset(1).p, edge)
     or Right(this.pointset(2).p, edge) then
          left_endpoint := this.pointset(1);
          right_endpoint := this.pointset(2);
     else
          left_endpoint := this.pointset(2);
          right_endpoint := this.pointset(1);
     end if;

     -- set up data for new point and line segments
     new_point.p := point_of_intersection(this.l, edge);
     if side = leftside then
          cut(this.l, edge, new_line.l, this.l);
          left_line := new_line;
          right_line := this;

          for i in 1 .. 2 loop
               if left_endpoint.lineset(i) = this then
                    left_endpoint.lineset(i) := new_line;
               end if;
          end loop;
     else
          cut(this.l, edge, this.l, new_line.l);
          left_line := this;
          right_line := new_line;

          for i in 1 .. 2 loop
               if right_endpoint.lineset(i) = this then
```

```
                    right_endpoint.lineset(i) := new_line;
                end if;
            end loop;
        end if;

        -- set up links for line segments
        new_line.pointset := new point_set(1 .. 2);
        new_line.polyset := new polygon_set(1 .. this.polyset'Length);

        left_line.pointset(1) := left_endpoint;
        left_line.pointset(2) := new_point;
        right_line.pointset(1) := (right_endpoint);
        right_line.pointset(2) := new_point;
        for i in 1 .. this.polyset'Length loop
            new_line.polyset(i) := this.polyset(i);
        end loop;

        -- set up new points links
        new_point.lineset := new line_set(1 .. 2);
        new_point.polyset := new polygon_set(1 .. this.polyset'Length);

        new_point.lineset(1) := left_line;
        new_point.lineset(2) := right_line;
        for i in 1 .. this.polyset'Length loop
            new_point.polyset(i) := this.polyset(i);
        end loop;

        -- and new point to each polygon and fix poly's links
        for i in 1 .. this.polyset'Length loop
            addpoint(this.polyset(i), new_point,
                     left_endpoint, right_endpoint);
            if side = leftside then
                addline(this.polyset(i), left_line);
            else
                addline(this.polyset(i), right_line);
            end if;
        end loop;

        this.state := undisturbed;
        new_point.state := undisturbed;
        new_line.state := move_me;
    end cut_line;

    -- assumes lines have already been cut, with the new points, etc
    -- having been stored somewhere accessible, and linked together
    procedure cut_poly(stack : in out paper; this : in out polygon_access;
                       side : direction; edge : line; dest : Integer) is
        new_line : line_access;
        oldpoly, left_poly, right_poly : polygon_access;
        endpoints_found, numpoints_onleft, numpoints_onright : Integer := 0;
        leftpoints_linked, rightpoints_linked : Integer := 0;
        leftlines_linked, rightlines_linked : Integer := 0;
    begin
        stack.n_lines := stack.n_lines + 1;
        stack.lineset(stack.n_lines) := new linked_line_segment;
```

```
new_line := stack.lineset(stack.n_lines);

stack.n_polys := stack.n_polys + 1;
left_poly := new linked_polygon;
right_poly := new linked_polygon;

oldpoly := this;
if side = leftside then
     stack.polyset(dest) := left_poly;
     this := right_poly;
else
     this := left_poly;
     stack.polyset(dest) := right_poly;
end if;

-- set up line data and all but new poly links back to here
new_line.l := edge_of_intersection(oldpoly.A, edge);
new_line.polyset := new polygon_set(1 .. 2);
new_line.polyset(1) := left_poly;
new_line.polyset(2) := right_poly;
new_line.pointset := new point_set(1 .. 2);
for i in 1 .. oldpoly.pointset'Length loop
     if On(oldpoly.pointset(i).p, edge) then
          endpoints_found := endpoints_found + 1;
          numpoints_onleft := numpoints_onleft + 1;
          numpoints_onright := numpoints_onright + 1;

          new_line.pointset(endpoints_found) := oldpoly.pointset(i);
          addline(oldpoly.pointset(i), new_line);
     elsif Left(oldpoly.pointset(i).p, edge) then
          numpoints_onleft := numpoints_onleft + 1;
     else
          numpoints_onright := numpoints_onright + 1;
     end if;
end loop;

-- set up left and right polygon memory and data
cut(oldpoly.A, edge, left_poly.A, right_poly.A);
left_poly.pointset := new point_set(1 .. numpoints_onleft);
left_poly.lineset := new line_set(1 .. numpoints_onleft);
right_poly.pointset := new point_set(1 .. numpoints_onright);
right_poly.lineset := new line_set(1 .. numpoints_onright);

-- scan through old polygon, picking points and lines and replacing
-- pointers to oldpoly with appropriate new pointers; add newline
for i in 1 .. oldpoly.pointset'Length loop
     -- handle point(i): link poly to it, link it to poly
     if Left(oldpoly.pointset(i).p, edge) then
          leftpoints_linked := leftpoints_linked + 1;
          left_poly.pointset(leftpoints_linked)
                                   := oldpoly.pointset(i);

          replace(oldpoly.pointset(i), oldpoly, left_poly);
     elsif Right(oldpoly.pointset(i).p, edge) then
          rightpoints_linked := rightpoints_linked + 1;
```

```
                right_poly.pointset(rightpoints_linked)
                                        := oldpoly.pointset(i);

                replace(oldpoly.pointset(i), oldpoly, right_poly);
           else -- on edge
                leftpoints_linked := leftpoints_linked + 1;
                rightpoints_linked := rightpoints_linked + 1;
                left_poly.pointset(leftpoints_linked)
                                        := oldpoly.pointset(i);
                right_poly.pointset(rightpoints_linked)
                                        := oldpoly.pointset(i);

                replace(oldpoly.pointset(i), oldpoly, left_poly);
                addpoly(oldpoly.pointset(i), right_poly);
           end if;

           -- handle line(i)
           if onsides(oldpoly.lineset(i).l, leftside, edge) then
                leftlines_linked := leftlines_linked + 1;
                left_poly.lineset(leftlines_linked) := oldpoly.lineset(i);

                replace(oldpoly.lineset(i), oldpoly, left_poly);
           else -- on right
                rightlines_linked := rightlines_linked + 1;
                right_poly.lineset(rightlines_linked) := oldpoly.lineset(i);

                replace(oldpoly.lineset(i), oldpoly, right_poly);
           end if;
      end loop;
      left_poly.lineset(numpoints_onleft) := new_line;
      right_poly.lineset(numpoints_onright) := new_line;

      -- free old memory and reset states
      Free_pointset(oldpoly.pointset);
      Free_lineset(oldpoly.lineset);
      Free_polygon(oldpoly);

      new_line.state := undisturbed;
      if side = leftside then
           left_poly.state := move_me;
           right_poly.state := undisturbed;
      else
           left_poly.state := cut_me;
           right_poly.state := undisturbed;
      end if;
end cut_poly;

procedure cut_everything(stack : in out paper; p : point; edge : line;
                layer : Natural) is
      dir : direction;
      old_n_lines, old_n_polys, polys_cut, after_split : Integer := 0;
begin
      if Left(p, edge) then dir := leftside;
      else dir := rightside;
      end if;
```

```
     -- go through the valid lines, splitting and inserting new lines and
     -- points after the current set of valid lines and points
     old_n_lines := stack.n_lines;
     for i in 1 .. old_n_lines loop
          if stack.lineset(i).state = cut_me then
               cut_line(stack, stack.lineset(i), dir, edge);
          end if;
     end loop;

     -- go through the top and bottom sections from bottom to top, so
     -- that we may insert the polygons we cut in a regular fashion from
     -- top to bottom in the gap
     -- new edges are inserted after all the existing valid lines
     old_n_polys := stack.n_polys;
     for i in reverse 1 .. layer loop
          if stack.polyset(i).state = cut_me then
               cut_poly(stack, stack.polyset(i), dir, edge,
                     layer + polys_cut + 1);
               polys_cut := polys_cut + 1;
          end if;
     end loop;
     after_split := layer + numlayers(stack) - old_n_polys + 1;
     for i in reverse after_split .. numlayers(stack) loop
          if stack.polyset(i).state = cut_me then
               cut_poly(stack, stack.polyset(i), dir, edge,
                     layer + polys_cut + 1);
               polys_cut := polys_cut + 1;
          end if;
     end loop;
end cut_everything;

-------------------------------------------------------------------

procedure pull_points(stack : paper; across : line) is
     n : Integer := numpoints(stack);
begin
     for i in 1 .. n loop
          if stack.pointset(i).state = move_me then
               stack.pointset(i).p := reflect(stack.pointset(i).p, across);
               stack.pointset(i).state := modified;
          end if;
     end loop;
end pull_points;

procedure pull_lines(stack : paper; across : line) is
     n : Integer := numlines(stack);
begin
     for i in 1 .. n loop
          if stack.lineset(i).state = move_me then
               stack.lineset(i).l := reflect(stack.lineset(i).l, across);
               stack.lineset(i).state := modified;
          end if;
     end loop;
end pull_lines;
```

```
procedure pull_polys(stack : paper; across : line) is
     n : Integer := numlayers(stack);
begin
     for i in 1 .. n loop
          if stack.polyset(i).state = move_me then
               stack.polyset(i).A := reflect(stack.polyset(i).A, across);
               stack.polyset(i).state := modified;
          end if;
     end loop;
end pull_polys;

procedure pull_everything(stack : paper; across : line) is
begin
     pull_points(stack, across);
     pull_lines(stack, across);
     pull_polys(stack, across);
end pull_everything;

-----------------------------------------------------------------------

function newSheet return paper is
     result : paper;
     a,b,c,d : point;
     ab, bc, cd, da : line_segment;
     abcd : polygon;
     pset : Point_Array(1 .. 4);
     lset : array(1 .. 4) of line_segment;
begin
     a := newPoint(0.0, 0.0);
     b := newPoint(100.0, 0.0);
     c := newPoint(100.0, 100.0);
     d := newPoint(0.0, 100.0);
     pset := (a,b,c,d);

     ab := newLine_Segment(a,b);
     bc := newLine_Segment(b,c);
     cd := newLine_Segment(c,d);
     da := newLine_Segment(d,a);
     lset := (ab, bc, cd, da);

     abcd := newPolygon(pset);

     result.pointset := new point_set(1 .. 4);
     result.lineset := new line_set(1 .. 4);
     result.polyset := new polygon_set(1 .. 1);

     for i in 1 .. 4 loop
          result.pointset(i) := new linked_point;
          result.pointset(i).p := pset(i);
          result.pointset(i).lineset := new line_set(1 .. 2);
          result.pointset(i).polyset := new polygon_set(1 .. 1);

          result.lineset(i) := new linked_line_segment;
          result.lineset(i).l := lset(i);
```

```
            result.lineset(i).pointset := new point_set(1 .. 2);
            result.lineset(i).polyset := new polygon_set(1 .. 1);
        end loop;
        result.polyset(1) := new linked_polygon;
        result.polyset(1).A := abcd;
        result.polyset(1).pointset := new point_set(1 .. 4);
        result.polyset(1).lineset := new line_set(1 .. 4);

        --link them all together
        for unit in 1 .. 4 loop
            for link in 1 .. 2 loop
                result.pointset(unit).lineset(link) :=
                    result.lineset(((unit + link - 3) mod 4) + 1);
                result.lineset(unit).pointset(link) :=
                    result.pointset(((unit + link - 2) mod 4) + 1);
            end loop;
            result.pointset(unit).polyset(1) := result.polyset(1);
            result.lineset(unit).polyset(1) := result.polyset(1);
        end loop;
        for i in 1 .. 4 loop
            result.polyset(1).pointset(i) := result.pointset(i);
            result.polyset(1).lineset(i) := result.lineset(i);
        end loop;

        result.n_pts := 4;
        result.n_lines := 4;
        result.n_polys := 1;

        return result;
    end newSheet;

    -- uses augmented paper with linked points, lines, and surfaces
    -- accounts for physical connections (entities are marked by this)
    -- does not account for interfering/nested but unconnected layers
    -- does not account for impossibile models
    procedure fold(stack : in out paper; p : point;
                   fold_edge : line; layer : natural) is
        new_stack : paper;
    begin
        -- mark what needs to be moved with the fold
        -- this doesn't mark trapped layers, but will
        clear_flags(stack);
        mark_everything(stack, p, fold_edge);

        -- make enough space to hold the new units, and transfer old stack,
        -- leaving space for the cut-off bits
        prepare_stack(stack, layer);

        -- cut all the lines and polygons crossing the fold edge,
        -- and pull everything from one side to the other
        cut_everything(stack, p, fold_edge, layer);
        pull_everything(stack, fold_edge);
    end fold;
end papers;
```

APPENDIX C

LIBRARY PACKAGES USED

```
package points is
     type Point is private;
     type Vector is private;

     Zero_Vector : constant Vector;
     Zero_Vector_Error : exception;

---------------constructors for points and vectors--------------
     function newPoint(a : Vector) return Point;
     function newPoint(X,Y : Float) return Point;

     function newVector(a : Point) return Vector;
     function newVector(X,Y : Float) return Vector;

-----------------selectors for points and vectors---------------
     function X(a : Point) return Float;
     function Y(a : Point) return Float;

     function X(a : Vector) return Float;
     function Y(a : Vector) return Float;

-----------------------point operations----------------------
     function "-" (Left, Right : Point) return Vector;
     function "+" (Left : Point; Right : Vector) return Point;
     function "*" (Left : Float; Right : Point) return Point;
     function "*" (Left : Point; Right : Float) return Point;

     function "=" (Left, Right : Point) return Boolean;

     function Distance(a,b : Point) return Float;
----------------------vector operations----------------------
     function "+" (Left, Right : Vector) return Vector;
     function "-" (Right : Vector) return Vector;
     function "-" (Left, Right : Vector) return Vector;

     function "=" (Left, Right : Vector) return Boolean;

     function "*" (Left : Float; Right : Vector) return Vector;
```

```
      function "**" (Left : Vector; Right : Float) return Vector;
      function "**" (Left, Right : Vector) return Float;

      function Magnitude(a : Vector) return Float;
      function Perpendicular (V : Vector) return Vector;
      function Normalized(a : Vector) return Vector; -- may raise
Zero_Vector_Error
private
      type Point is record
           x,y : Float;
      end record;

      type Vector is record
           x,y : Float;
      end record;

      Zero_Vector : constant Vector := (x => 0.0, y => 0.0);
end points;

with points; use points;

package lines is
     type Line is private;
     type Line_Segment is private;

     No_Intersection : exception;
     Lines_Overlap : exception;

     function newLine(p : Point; v : Vector) return Line;
     function newLine_Segment(a,b : Point) return Line_Segment;

     function P(l : Line) return Point;
     function V(l : Line) return Vector;

     function A(l : Line_Segment) return Point;
     function B(l : Line_Segment) return Point;
-------------------------------------------------------------------------
     function On (a : Point; l : Line) return Boolean;
     function Left (a : Point; l : Line) return Boolean;
     function Right (a : Point; l : Line) return Boolean;
-------------------------------------------------------------------------
     function Intersect (L1,L2 : Line) return Boolean;
```

```
     function Intersect (L1 : Line_Segment; L2 : Line) return Boolean;
     function Intersect (L1 : Line; L2 : Line_Segment) return Boolean;
     function Intersect (L1,L2 : Line_Segment) return Boolean;

     function Point_of_Intersection (L1,L2 : Line) return Point;
     function Point_of_Intersection (L1 : Line_Segment;
                                     L2 : Line) return Point;
     function Point_of_Intersection (L1 : Line;
                                     L2 : Line_Segment) return Point;
     function Point_of_Intersection (L1,L2 : Line_Segment) return Point;
private
     type Line is record
          p : Point;
          v : Vector;
     end record;

     type Line_Segment is record
          a,b : Point;
     end record;
end lines;

with points; use points;
with lines; use lines;
with Ada.Finalization; use Ada.Finalization;

package polygons is
     type Polygon is new Controlled with private;
     type Point_Array is array (Integer range <>) of Point;

     procedure Adjust(poly : in out Polygon);
     procedure Finalize(poly : in out Polygon);

     function newPolygon(point_list : Point_Array) return Polygon;

     function numPoints(this : Polygon) return Positive;
     function vertice(this : Polygon; i : Natural) return Point;
     function edge(this : Polygon; i : Natural) return Line_Segment;

     procedure insertpoint(to_this : in out polygon;
                           newp, between_this, and_this : point);
private
     type Full_Polygon(numpoints : Positive) is record
```

```
        point_set : Point_Array(1 .. numpoints);
    end record;
    type Polygon_Link is access Full_Polygon;

    type Polygon is new Controlled with record
        full_view : Polygon_Link;
    end record;
end polygons;

with points; use points;
with lines; use lines;
with polygons; use polygons;

package shape_math is
    type direction is (leftside, rightside);

    -- test to see if part or all of the shape is on one side of a line
    function onsides(this : point;
                     side : direction; of_line : line) return Boolean;
    function onsides(this : line_segment;
                     side : direction; of_line : line) return Boolean;
    function onsides(this : polygon;
                     side : direction; of_line : line) return Boolean;
-------------------------------------------------------------------------
    function Reflect (this : Point; over : Line) return Point;
    function Reflect (this : Vector; over : Line) return Vector;

    function Reflect (this : Line; over : Line) return Line;
    function Reflect (this : Line_Segment; over : Line) return Line_Segment;

    function Reflect (this : Polygon; over : Line) return Polygon;
-------------------------------------------------------------------------
    function Intersect(this : Polygon; path : Line) return Boolean;
    function Edge_of_intersection(this : Polygon; path : Line) return
Line_Segment;
-------------------------------------------------------------------------
    procedure Cut (this : Line_Segment; path : Line;
              leftside,rightside : out Line_Segment);
    procedure Cut (this : Polygon; path : Line; leftside,rightside : out
Polygon);
end shape_math;
```

```
with points; use points;
with lines; use lines;
with polygons; use polygons;

package display is
    procedure Put(a : Point);
    procedure Put(a : Vector);

    procedure Put(l : Line);
    procedure Put(l : Line_Segment);

    procedure Put(p : Polygon);
end display;
```

BIBLIOGRAPHY

Engel, Peter. Folding the Universe: Origami from Angelfish to Zen. Vintage Books, 1989. Dover, 1994.

Hull, Tom. "A Note on Impossible Paper Folding." American Mathematical Monthly. 103(#3) (1996): 240-241.

Hull, Tom. "On the Mathematics of Flat Origamis." Congressus Numerantium. 100 (1994): 215-224.

Kawasaki, T. and M. Yoshida. "Crystallographic Flat Origamis." Mem. Fac. Sci. Kyushu Univ. 43(1988): 152-157.

Lange, Robert. Origamimanip [Computer software]. Written for Mac OS, not available for general distribution.

Zamiatina, Ludmila. "Computer Simulations of Origami." http://www.wolfram.com/cgi-bin/MathSource/Applications/Graphics/Animation/0207-672. (1994): accessed May 26, 1997.