Origami On Computer

David Fisher

Acknowledgements

The author would like to thank his supervisor, Allan Bromley, who thought of doing a project on origami with a pop-up thesis in the first place and who has been very patient during this year. Thanks also to "Fred" (who I now know really exists) for a never-ending supply of jelly beans and for his generous loan of many origami books, to the people on the origami-l mailing list for many interesting ideas (especially Baz and Tom), and to Greg - who wanted to be in the credits. The greatest thanks belongs to God, without whom not a thing would have been possible.

Preface

Origami is the Japanese art of paper folding. Work has been done in the area of simulation of origami in an interactive environment, but most standard folds are very difficult to specify unless some kind of language is used to describe folding operations. Means of recording origami instructions exist, but there is no purely written form which is easily understood and may be used as the basis of a computer language for describing origami models.

This thesis describes a language for describing the folding sequences of flat origami models and a program for entering and executing instructions given in the language. Algorithms and data structures for performing origami operations are presented, and many issues involved in using computers to do origami are raised. The origami language is shown to be a better method of recording origami instructions than existing methods if it is used in conjunction with the program.

Contents

Overview

The thesis is divided into eight sections, as follows :

Chapter 1 is an introduction to origami, and should be read by readers unfamiliar with origami and origami terminology. Its contents are assumed knowledge throughout the rest of the thesis.

Chapter 2 contains a summary of literature relating to the use of computers in origami, means of recording origami instructions and origami mathematics.

Chapter 3 traces the development of a language for describing folding sequences and gives a justification for the syntax developed.

Chapter 4 compares some different methods of giving folding instructions including use of the origami language.

Chapter 5 describes algorithms for folding and tucking in origami models.

Chapter 6 gives the data structures used to represent an origami model and a folding sequence, and shows the effects of parsing different statements in the origami language on these data structures.

Chapter 7 describes problems and issues in the implementation of the program to enter the instructions and do the folding and gives details of what has been implemented.

Chapter 8 reflects on the project and mentions possible research areas.

There are also two appendices containing theorems and a complete syntax for the origami language.

Chapter 1. An Introduction To Origami

This section is intended for readers who are not familiar with origami. The terms and symbols used are assumed knowledge for the rest of the text. It may be skipped by capable folders.

Origami is the ancient Japanese art of paper folding. The paper used for folding is usually limited to a single sheet of square or rectangular paper; occasionally some other shape will be used, but it must always be convex [13]. Use of scissors or glue is forbidden; technically, if the paper is cut then it is kirigami which is being performed, not origami [12]. Within this seemingly limited domain there is a vast number of possibilities, ranging from the traditional crane to biplanes, roses and cuckoo clocks with pop-out cuckoos (see [10,13]).

Dr Sakoda, winner of the origami section of the First International Paper Aeroplane Competition sponsored by Scientific American in 1967, said : "There appear to be some common skills required in computer programming and origami ... and I have come across a number of programmers who are also paper folders. One common requirement is the fitting of parts into a circumscribed area ... ie., to work spatially in a restricted environment." [15].

Origami Terms

There is a standard language for describing origami folds and procedures. When a single fold is made towards the folder, so that if the paper is unfolded again the resulting crease forms a valley in the paper, the fold is called a valley fold. When the fold is done away from the folder and the resulting crease is like a ridge on a mountain, the fold is called a mountain fold.

A raw edge is an edge on the outside of the paper.

Some folds involve a combination of several simultaneous valley and mountain folds. These are known as combination folds. The following are some of the more common ones :

1. Reverse or Hood fold

Start with a sheet of rectangular paper, folded in half.

Inside reverse fold *Outside reverse fold*

2. Squash fold

3. Pleat

4. Crimp fold

5. Petal fold

6. Sink fold

7. Rabbit ear fold

There are several common starting sequences for origami models. These are called bases. Bases are often named after models which they may be used to make, such as a frog, a fish or a bird. A very common one is known as the preliminary base, which is also the starting point for several other bases. Some examples are given below.

Kite base

Preliminary base

Bird base

Waterbomb base

Fish base

Frog base

Origami Instructions

Folding directions for origami models are given as a sequence of steps, each step consisting of a diagram and usually some text. The diagram shows what the model looks like at the beginning of the current step, along with the location of the fold or folds to be done. The text is used to clarify the diagram, and often includes instructions which are difficult or impossible to show on the diagram [13]. Origami instructions are notoriously difficult to follow; most paper folders have learned the basics from other people rather than from books. One approach to making instructions clearer has been to use photographs of half-finished folds, but this is uncommon.

Akira Yoshizawa, acknowledged to be a world expert in origami, has developed a set of symbols which have become standard in origami diagraming. Some of these are given below:

Valley Fold

Mountain Fold

Crease

Raw Edge

X-Ray View

Cut

Fold from A to B

Fold from A to B and return

Next diagram enlarged

Turn the paper over

A good introduction to origami including some traditional models may be found in [12]. There was an article on origami in the Mathematical Games column of Scientific American in 1960 [5], and a more recent article including instructions on how to fold Albert The Experimental Rat in New Scientist [14]. The latter includes a good introduction to what Robert Lang calls "Technical Folding", which makes subjects such as lobsters and jack-in-the-boxes no longer an impossibility.

Chapter 2. Literature Survey

Little work has been done involving the use of computers to record and replay instructions for paper folding. Agui, Tekeda and Nakajima [1,2] have created a system for animating certain types of fold which requires the coordinates of the starting and finishing locations of vertices to be given; the computer does not actually do the folding. Some of the most recent work involving the application of computers to origami was done in 1992/1993 by Miyazaki, Yasuda, Yokoi and Toriwaki [16,17], who have developed a program in which a restricted group of fold types may be done interactively using a mouse. This work also involves folding in three dimensions. Details of these systems will be given later.

Research into data structures suitable for describing paper folding in terms of crease patterns of origami models has recently been done by Uchida and Itoh [20]. Unfortunately this is in Japanese and a translation was not available at the time of this writing.

In 1994 Robert Lang created a program to aid in the design of origami called TreeMaker, which optimizes a design of an origami base to take up as much space as possible so that there is a minimal amount of wasted excess paper in the model. However, this program does not tell you how to find the base that corresponds to the design created; it just guarantees that such a base exists.

Recording of origami instructions has been done since at least as far back as 1797, when "Sembazuru Orikata" ("Folding 1000 Cranes") was published in Japan (see [13]). In the 1950s and 60s a standard set of symbols for diagraming origami models was developed by Akira Yoshizawa. This system was adopted by western authors Harbin and Randlett in the early 1960s and has been used as an international language for communicating origami instructions since that time. A standard terminology for describing basic folds and starting points for models is also used in origami instructions.

In 1975 Alice Gray produced a written origami language called O.I.L., which stands for Origami Instruction Language [6]. This language allows for descriptions of arbitrary points on or off the paper, but is awkward to use since every point on the paper is renumbered after each step, and the language includes instructions which are not intuitively understandable, such as "1---3". The booklet describing O.I.L. is no longer available, and so a complete description of the language cannot be given.

Another method for recording origami models is to store the complete model's crease pattern - that is, the locations of all of the mountain and valley folds after the model has been completed. However, only the final state of the model will be recorded, not the steps involved in folding the model. A crease pattern by itself may also be ambiguous.

The work by Miyazaki, Yasuda, Yokoi and Toriwaki [16,17] does not include a language for describing folds; the kinds of folds which can be made are limited by what may be done using the mouse. There do not appear to be any programs for performing origami instructions with the input consisting of text.

There was a discussion on an origami mailing list called origami-l about aspects of origami languages. Archives of the mailing list are available at the time of this writing from the anonymous ftp site rugcis.rug.nl in the directory /origami/archives.

Quite a lot of work has been done on mathematics associated with paper folding [3,4,7,9, 10,11] including some work on using origami to teach mathematics [8] and the discovery that it is possible to trisect an arbitrary angle using origami, which is impossible in the Euclidean system [18]. Huzita has enumerated what he considers to be the unit operations of origami (such as folding a point to a point or an edge to an edge). These are reproduced in [18].

The aim of Agui, Tekeda and Nakajima was to determine the intermediate pictures between key frames of an origami model for the purposes of animation of the folding procedure. Linear interpolation of vertices does not work, because a property of paper is that distances between adjacent vertices must remain constant. An origami model is considered to be made of flat polygons; curves and thickness of the paper are not taken into account. Three kinds of fold are considered : folds parallel to a side (horizontal or vertical folds), diagonal folds, and composite folds in which a small number of parallel and diagonal folds happen simultaneously. The number of models which are made up of only such folds is relatively small; however, they are able to produce intermediate pictures of a "flapping bird". The data structures used are not described in detail, although it is clear from the article that an origami model is stored as a set of polygons with vertices in clockwise order with respect to the front of the face.

Miyazaki, Yasuda, Yokoi and Toriwaki created an interactive system in which the operator may view the current state of the origami model from any direction (using the keyboard) and perform three kinds of folding operations : bending, folding up and tucking in. In each of these, a vertex is selected with the mouse and the position of the vertex is changed to a different three dimensional location (using the mouse for two dimensions and the buttons for the third dimension) while the model is dynamically updated on the screen. If the result is a fold which is very close to a standard fold (folding a vertex to a vertex, an edge to an edge, folding through two vertices or at a 30°, 45° or 60° angle or a multiple of these) then the fold is automatically corrected.

Bending involves folding a vertex to any point in space, and corresponds to doing a mountain or valley fold in traditional origami terminology. Folding up is a special case of bending where the angle is 180°. Tucking in corresponds to an inside reverse fold in traditional origami terminology; the vertex selected is the one which will go down between the two layers of paper involved in the fold. The reverse fold is assumed to be symmetrical. The line along which the folds occur is the intersection of the folded face and the plane equidistant from the original position and the new position of the vertex.

These three operations cover a limited amount of origami; standard folds such as sink folds and petal folds cannot be done. Another disadvantage is that all folds except for valley, mountain and inside reverse folds will take many steps, whereas in standard origami instructions standard folds take a single step.

The data structure consists of faces, edges and vertices. The faces are stored in a binary tree, with the children of a node representing the two faces which the parent was previously split into. The leaves represent the current state of the paper and the root represents the original unfolded piece of paper. Each face has an associated group number; faces which are in the same plane are in the same group.

Edges are stored in a binary tree in a way similar to faces, except that several trees may exist because a fold may generate entirely new edges as well as split existing edges in two.

Vertices are stored in lists in a lookup table. Each vertex has an associated three dimensional location.

There are links between vertices, edges and faces, such as the edges which are associated with a face.

A small description is given on how to determine which faces will move along with other faces : the selected face moves, faces connected to the moving part of a moved face move, in bending and folding up operations faces overlapping moved parts of other moved faces on a certain side will move, and in tucking operations faces which overlap moved parts of the two other moved faces between them will also move.

Chapter 3. Syntax Of The Origami Language

3.1. Issues In The Design Of The Language

In order to be useful, the origami language must be simple for a human reader or writer to use. For this reason, there should be no mention of absolute coordinates or directions in the language. The existing origami terminology is perfectly able to describe all basic kinds of folds, so it is possible to keep the origami language similar to what is used in traditional origami instructions. This would certainly be helpful for understanding the language and for recording existing folding sequences.

Ideally, it should be possible to follow the commands given in the origami language without the aid of a computer program. In order to make this as simple as possible for the reader, locations on the paper would have to be described in terms of directions and other parts of the paper. This is possible to do using natural language and/or diagrams but is very difficult using a computer language, and doing so goes against the requirement that there should be no mention of absolute directions. The alternative to doing this is to have named locations on the paper with which it is simple to describe any other parts of the paper as required. This may be done by assigning names to the vertices (the corners of the paper). It is possible but more difficult to follow the commands in the origami language without using a computer if locations on the paper are specified in this way; however, writing the commands becomes near impossible for anything but simple models without using a computer, for reasons which will be explained later in the section on assigning labels.

The syntax for the origami language is outlined in the following sections. A complete syntax is given in Appendix B, which also describes the syntax conventions used.

3.2. General Format Of Origami Instructions

The standard method of giving origami instructions is as follows :

The size and shape of the paper is given first, along with the side of the paper which is initially "up" (the plain or the coloured side). Sometimes there is some pre-creasing, such as "start with a square of paper folded diagonally in half". The starting point of a model may also be another model (such as a base) or a particular stage of another model (eg. "start with step 12 of the camel").

Following the description of the paper a sequence of numbered steps is given. Each step is given a new diagram. More than one fold may be made in a single step.

In the origami language, a corresponding format would be :

Model ::= *Header StepList*

The header consists of some way to indicate that it is an origami model and a description of the initial piece of paper :

Header ::= *origami Identifier UsesPart*

UsesPart ::= *uses Expression* : *Expression paper*

The expressions after "uses" represent the relative width and the height of the paper. It seems awkward to specify which is the width and which is the height, and there is no standard in origami books, so the convention adopted is width first then height. The use of the ":" symbol to indicate the ratio is common in origami books [12, 13]. The initial piece of paper must have names for its four vertices. A simple convention is to name the corners a, b, c and d, where length(a,b) and (c,d) are equal to the width and going clockwise with respect to the plain side of the paper.

It is also necessary to specify whether the paper starts with the plain side up or down. The default in origami books is for the plain side to be up. To save the user having to remember whether to say "plain side up/down" or "coloured side up/down" (all of which are used in origami instructions), all combinations are allowed :

UsesPart ::= *uses Expression* : *Expression paper* [*Viewside*]

Viewside ::= (*plain* | *coloured*) *side* (*up* | *down*)

To allow the user to use another model or up to a certain step of another model as the starting point for a new model, a possible syntax is :

UsesPart ::= *uses* [*step Number of*] "*String*"

This way there can be a library consisting of bases, paper of various sizes and other starting points of models.

Sometimes a non-rectangular convex shape is desired as a starting point, such as an equilateral triangle. It would be complicated to specify such a shape using angles and distances, and limiting to offer only a fixed number of predefined shapes. A simple solution is to allow the user to fold the shape and then "cut out" a single face of the model, discarding the rest of the paper. This ensures that the shape will be convex (by theorem 5). The syntax for restarting with a given face is :

RestartStatement ::= *restart with Face* [*Viewside*]

The user should not be forced to use numbered steps in the description of the model; the steps exist only as reference points (for the user and for the "uses step Number" command). There should also be a section where pre-creasing commands can be given before the first step. For these reasons the syntax for the list of steps to perform is :

StepList ::= [*StatementList*] { *step Number StatementList* }

The number after the keyword "step" is a positive floating point number. The steps will usually be integers, but for the user’s convenience any floating point number may be used. The step numbers must be in ascending order.

The kinds of statement which may appear in a *StatementList* apart from the "restart" statement are : folds, tucking statements, viewpoint changes and assignment statements.

3.3. Kinds Of Fold

There are two different kinds of fold : simple mountain or valley folds, and multiple folds involving several simultaneous mountain and valley folds. The reason why simple folds are treated separately rather than as a special case of multiple folds will be made clear in the section on multiple folds.

3.3.1. Simple Folds

During a simple fold, different parts of the paper are aligned and the curved part of the paper is pressed down to form a crease. This crease will always be straight by Theorem 1. The paper may be pressed down everywhere along the line, or only along a segment. Occasionally the paper is pressed down at only one point to make a mark rather than a crease.

There are two possible constraints which may be given to specify a simple fold : two parts of the paper may be brought to the same location, or the fold may be made to pass through a specified point. The possible combinations of these constraints are as follows :

1. Align two points
2. Align two edges
3. Align an edge with two points
4. Align an edge and a point such that the resulting fold passes through a specified point
5. Align one edge with one point and another edge with another point
6. Fold through two points
7. Fold an edge to itself such that the resulting fold passes through a specified point (the fold will be perpendicular to the original edge)
8. Make the fold go along the same line as an existing edge

All except for the last one of these are the same as the ones arrived at by Huzita [18]. If a pair of points is thought of as a line then the second and third types of fold may be combined together into "align an edge with a line" and the sixth and last may be combined together into "fold along a line". Sometimes an instruction will be given to do a more approximate fold, such as :

9. Fold parallel to an edge such that the resulting fold passes through a specified point
10. Fold at a specified angle to an edge such that the resulting fold passes through a specified point

A point may be a vertex, the intersection of two non-adjacent edges or an approximate location. Examples of approximate locations are : a relative distance along an edge (such as 1/3), the centre of a face, or an absolute distance along a line (given in terms of distances between points).

It should never be necessary for the user to specify actual coordinates of points; vertices should be named, and everything else should be given in terms of the vertices. This avoids the problem of specifying points in terms of directions and relationships with other points such as "the point just below the one on the far left next to the vertical crease". Edges may be given as a pair of vertices (for example, A-B), and faces may be specified by using several vertices (for example, face(A,B,C,D)). Arbitrary points may be specified in terms of relative distances along lines (for example, 2/3 along A-B). A complete syntax for points, distances and other expressions will be given later.

A mountain or valley fold is always done relative to the viewpoint of the folder : if the paper is reversed, the meaning of a mountain and a valley fold is reversed. In normal origami instructions, the upward-facing side of the paper is always specified by the diagram. Another means is needed to specify the "up" direction in the origami language. It would be inconvenient if the user was required to specify the direction after every fold was made; it should be implicit in the instructions for the fold. One way to do this is to specify which part of the paper moves when each fold is done. The most common kinds of simple fold, aligning two points or two edges, have a natural syntax in which the moving part of the paper is implicit :

fold A to B and *fold A-B to C-D*

Given a single point or edge which moves (A or A-B), the other parts of the paper which move in a mountain or valley fold are implicit and unambiguous.

In the fold "fold A to B", the fold line (that is, the line upon which the crease will appear) will be the perpendicular bisector of the line which A and B both lie on (by theorem 1), as shown in the diagram below.

In the fold "fold A-B to C-D", the fold line will bisect the angle which A-B makes with C-D (unless they are parallel, in which case the fold line will be parallel to and equidistant from A-B and C-D). The angle bisected is the one formed where A-B and C-D intersect; this means that if the lines (A,B) and (C,D) intersect at a point on segment (A,B) or segment (C,D), the fold is ambiguous, and so is illegal. The three cases are shown in the diagram below.

It is also necessary to state whether the simple fold is a mountain or a valley fold. As stated by Samuel Randlett in [19], the most common kind of fold in origami is the valley fold, and so the word "valley" is often left out of the instructions. So a possible syntax for simple folds is :

SimpleFold ::= [*mountain* | *valley*] *fold* (*Point to Point*) | (*Edge to Edge*)

Using similar syntax, here are some of the other kinds of simple fold :

fold A to B-C through D

fold B-C to A through D

fold along A-B moving C

fold along line(A,B) moving C

The first two are the same fold, except that A is moving in the first fold and B-C is moving in the second fold. The second two are different folds : "fold along A-B" is a fold along an existing edge, while "fold along line(A,B)" is a fold through two points, A and B. The difference is that in the last fold, A and B may be any points, and it is not necessary for there to be an edge between them.

The fold "fold A to B-C through D" is equivalent to the fold "fold A to E", where E is a point on edge B-C such that distance(E,D) = distance(A,D). "fold B-C to A through D" is similar. This will sometimes be ambiguous or impossible, as the diagrams below show.

Ambiguous situation *Impossible situation*

Up to this point it has been assumed that the points specified in each fold may be any arbitrary points. However, when moving points are being specified, they must be at specific locations on the paper. Conceptually, a "point" is any point in space, and is not on a particular face; for example, if a rectangle is folded in half, the point at the centre of the top face is the same point as the centre of the bottom face. Similarly, a "line" is any line in space. However, vertices and edges are at specific locations on the paper. Therefore, all moving parts of the paper must be vertices or edges, unless some way of identifying a point with a face is introduced. Doing so would make the syntax unnatural, and is not necessary for most origami models, and so has been avoided.

The syntax for simple folds becomes :

SimpleFold ::= [*valley* | *mountain*] *fold FoldType*

FoldType ::= *Vertex to* [*Point* | *Vertex*]

FoldType ::= *Edge to* [*Line* | *Edge*]

FoldType ::= *Vertex to* [*Line* | *Edge*] *through* [*Point* | *Vertex*]

FoldType ::= *Edge to* [*Point* | *Vertex*] *through* [*Point* | *Vertex*]

FoldType ::= *along* [*Line* | *Edge*] *moving Vertex*

This does not include all of the simple folds originally given. Aligning one edge with one point and another edge with another point is quite uncommon and so has been left out. Folding at an angle to an edge is a frequent enough fold to be included. Folding an edge to itself to form a perpendicular fold and creating a fold which is parallel to a line or edge may be done by folding at a 90 degree or 0 degree angle, respectively.

The syntax for folding at an angle to an edge is as follows :

FoldType ::= *at Angle to* [*Line* | *Edge*] *through* [*Point* | *Vertex*] *moving Vertex*

As it is, this fold is ambiguous, so some conventions are needed to specify which direction the angle is relative to. See the diagrams below for the possible cases.

A-B folded, C not on A-B

A-B unfolded, C not on A-B

A-B folded, C on A-B

A-B unfolded, C on A-B

It is common in origami to do a simple fold and then unfold it straight away, leaving a crease. Replacing the "fold" with "crease" in the syntax would make some statements sound unnatural; for example, *crease A to B* does not make sense. In written origami instructions the words "and return" may be appended to a folding instruction, meaning do the fold and then return to the previous position. This is a satisfactory way of expressing creases. Note that in simple folds which require a moving vertex to be specified, the vertex may be left out if a crease is being done. The syntax for simple folds becomes :

SimpleFold ::= [*valley* | *mountain*] *fold FoldType* [*and return*]

Another frequent operation is unfolding a previously folded edge. This is equivalent to folding along an existing edge, but to make it less confusing to the user there is a separate statement for this situation :

UnfoldStatement ::= *unfold Edge moving Vertex*

Re-folding an unfolded edge does not require a separate statement; the existing syntax is clear enough for this situation.

3.3.2. Multiple Folds

A multiple fold is a fold involving several simultaneous mountain and valley folds. Any origami model may be seen as a single multiple fold. The reason why instructions for origami models are not given as a single multiple fold is that it is extremely difficult to fold a model in one go; there will also usually be ambiguities in the layering.

When origami is done using physical paper, the creases involved in a multiple fold are usually made before the multiple fold is done, since it is difficult to accurately create several folds simultaneously; sometimes one or two creases are left out and appear during the fold when the paper is pressed flat. However, this pre-creasing is not usually mentioned in origami instructions : the creasing and the multiple fold are treated as a single fold. It would be preferable to do the same in the origami language; however, doing so would require the user to calculate angles and locations of vertices in order to specify where the folds occur, which would be too unnatural.

If the creases have already been formed before a multiple fold is done then all of the vertices involved in the fold will already exist when the fold is done. All that is needed to specify the multiple fold is a list of the edges involved, the kind of fold each edge will have (if it is any different from the kind of fold it already has), and a single face which does not move during the multiple fold, so that the subsequent viewpoint (which side of the paper is facing upwards) is unambiguous. Note that the face may be a face which does not yet exist but will be created during the fold.

A possible syntax for multiple folds is :

MultipleFold ::= *multifold* [*EdgeList*] { , (*valley* | *mountain*) *EdgeList* } *holding Face*

EdgeList ::= *Edge* { , *Edge* }

The edges in the first list are assumed to have the desired fold type already (mountain or valley). All edges after the keywords "mountain" or "valley" up to the next keyword have that kind of fold. At least one edge must be specified in a multiple fold.

This syntax leaves out two possibilities. In a multiple fold, a folded edge may become unfolded (for example, in a squash fold). It is also possible for a folded edge to reverse its fold type (for example, in a reverse fold). In both cases a new kind of specifier must be used. The syntax for multiple folds becomes :

MultipleFold ::= *multifold* [*EdgeList*] { , (*valley* | *mountain* | *unfold* | *reverse*) *EdgeList* }
   holding Face

There are some multiple folds in which one of the folds involved is difficult to give (for example, the mountain fold in the rabbit ear fold). In normal folding instructions the other folds will be given and the remaining one appears when the fold is done. The approximate location of this fold must be given, since ambiguities are possible. In order to specify the additional fold, the fold type and the edges it is between must be given. The edges must have a vertex in common. The additional fold must be between two edges involved in the multiple fold which do not have another edge involved in the fold between them. A syntax which allows additional folds is :

MultipleFold ::= *multifold* [*EdgeList*] { , (*valley* | *mountain* | *unfold* | *reverse*)
   (*EdgeList* | *between Edge and Edge*) } *holding Face*

Note that "between" can only come after "mountain" or "valley". It is possible to have several additional folds, but each must be coincident with a different interior vertex.

As has already been stated, the layering after a multiple fold may be ambiguous. To make it unambiguous, one or more layering relationships between faces must be given. A possible syntax is to allow the user to specify which faces are tucked under which other faces during the fold in a way similar to the "tuck" command (which will be given later) :

MultipleFold ::= *multifold* [*EdgeList*] { , (*valley* | *mountain* | *unfold* | *reverse*)
   (*EdgeList* | *between Edge and Edge*) } *holding Face*
   [*tucking Face under Face* [, { *Face under Face* }]]

The pre-creasing involved in multiple folds sometimes involves creases which only extend to a particular point (for example, in the rabbit ear fold). With physical paper, this involves pressing down on only part of the crease line. The part of the paper pressed down may be a segment or a ray (that is, an infinite line with one endpoint). Since simple folds have the same effect except for adding some unnecessary creases, it is possible to avoid having to create a folding command to handle this kind of pre-creasing.

The above syntax for multiple folds has not been incorporated into the program; only simple folds have been implemented.

3.4. Labelling Vertices

There are two approaches to naming vertices created by a fold or a crease: they may be named by the user, or they may be given names automatically generated by the computer (which may be changed by the user). The convention for automatically generated names must be understandable by the user so the user knowns how to refer to any given vertex. A possible convention is to include in the name the two vertices at the endpoints of the edge the vertex was created on; this would create some very long names. Whatever way is chosen, it would be awkward for the user to work out the name for each vertex.

In order for the user to be able to name a vertex, the name must be given when the vertex is created; there is no other time at which the vertex can be unambiguously specified in a way that is clear to the user. To specify the vertex only the edge which it was created on needs to be given. Because the edge will not exist after the fold is done (since the fold split it into two parts), naming of vertices must be part of the fold command. It is true that the vertex may be specified unambiguously by giving the edge which it split some time later, but this would be utterly confusing to the user. The best place for the labelling is at the end of the statement. The syntax for simple folds becomes :

SimpleFold ::= [*valley* | *mountain*] *fold FoldType* [*and return*] [*Labelling*]

Labelling ::= *creating Identifier on Edge* { , *Identifier on Edge* }

Since unfolding is really a kind of simple fold, the syntax for the unfold statement also requires a labelling statement :

UnfoldStatement ::= *unfold Edge moving Vertex* [*Labelling*]

Multiple folds can also cause new vertices to be created, even though all of the vertices required to specify the multiple fold must already exist when the fold is done. More than one vertex could be created on the same edge in a multiple fold, so the syntax should provide a means of specifying which created vertex on a given edge is being referred to. If all of the created vertices were given names, an obvious syntax would be : multifold ... creating x,y,z on a-b. However, since only some of them might be given names, the syntax should allow the user to specify the ordinal value of the vertex. The user should still be allowed to use the normal naming syntax if only a single vertex was created on an edge. The following is imperfect but is offered as a potential syntax for labelling multiple folds :

MultipleFold ::= *multifold* [*EdgeList*] { , (*valley* | *mountain* | *unfold* | *reverse*)

(*EdgeList* | *between Edge and Edge*) } *holding Face* [*MultifoldLabelling*]

[*tucking Face under Face* [, { *Face under Face* }]]

MultifoldLabelling ::= *creating Ident* [: *Number*] *on Edge* { , *Identifier* [: *Number*] *on Edge* }

The "tucking" part is required to come after the labelling part because the vertices referred to in the "tucking" part may be created during the fold, and so will not be able to be referred to until after the labelling is done.

The fact that vertices must be named as soon as they are created means that if the user enters the instructions for the origami model as text, without the aid of the program, either every newly created vertex must be named (which is wasteful and inconvenient), or the user must know in advance which vertices will need to have names. To make matters worse, in order to be able to name a vertex the endpoints of the edge the vertex is on must both have names, and so the user will also have to work out in advance which vertices will have to have names because they will be used for naming other vertices. Even if all vertices are named as they are created, the user faces the problem of working out where the newly created vertices appear, which is difficult if several layers of paper are involved.

The solution is to use the program to enter the text. The computer should keep track of when each vertex was created and allow the user to name the vertex at any time. The program must "go back" and insert a labelling statement at the appropriate part of the text when this is done. This still leaves the problem of needing names for the endpoints of the edge on which a vertex was created. Either the user could be asked to name all such vertices (recursively) as soon as an unnamed vertex is named, or the vertices could be given names generated by the program. The latter seems to be a better solution, since the user may give meaningful names to those vertices later if desired.

3.5. Other Origami Operations

Some other common operations in origami are as follows : tucking in, turning the model over, unfolding the model completely, repeating a fold, curling the paper and inflation. The last two are beyond the scope of this work, since the origami language is restricted to flat origami models.

The various ways of tucking faces under other faces may be reduced to two possibilities, with which any kind of tucking may be done : tucking one face under another and untucking a face completely. There is a simple syntax for these :

Tuck ::= *tuck Face under Face*

Tuck ::= *untuck Face*

The standard way of indicating that the model is to be turned over is with the words "turn over ". Obviously two tokens ("turn" and "over") are not necessary, but they make the meaning clearer :

ViewStatement ::= *turn over*

Note that it is not necessary to have a statement to alter the viewpoint in any other way (such as turning the model clockwise 90 degrees), since all of the commands in the origami language are independent of the viewpoint (other than needing to know which side of the paper is "up").

The command to unfold the model completely may be combined with the existing command to unfold a folded edge. The "up" direction of the resulting model must also be specified; as with the initial piece of paper, the plain side will be up by default. The unfolding command becomes :

UnfoldStatement ::= *unfold Edge moving Vertex* [*Labelling*]

UnfoldStatement ::= *unfold all* [*Viewside*]

Repeating the previous fold involves generalizing the fold and applying it in a different situation. Unless the model was completely symmetrical before the previous fold this operation could be very complex to do and has not been included in the language.

3.6. Types And Expressions

The types required by the syntax so far are : vertex, point, edge, line, face, real number and angle. Absolute and relative distances have also been mentioned. There is also a string type, but a string can only occur in the header or an include statement. A formal syntax for expressions is as follows :

Expression ::= *Term* { (+ | -) *Term* } [*degrees* | *along Expression(=line/edge)*]

The terms may be of any type (vertex, point, edge, line, face, real, angle or distance). Only a real, an angle or a distance may be added to or subtracted from another term, which must be of the same type. If the keyword "degrees" follows a term or terms which are of type real, the expression type is angle. If the keyword "along" follows, then the return type will be point : if the rest of the expression is of type real, then the point is at a relative distance along the edge or line; otherwise, if the expression type is distance, the point is at an absolute distance along the edge or line. For example, "2/3 along a-b" is a relative distance, while "distance(c,d) along a-b" is an absolute distance.

Term ::= *Factor* { (* | /) *Factor* }

The factors may be of any type. One operand of the "*" operator must be of type real and the other must be a real, an angle or a distance. The result type is the type of the non-real operand, or real if both operands are real. A real, an angle or a distance may be divided by the same type to give a real. An angle or a distance may be divided by a real to give the same type.

Factor ::= *Number*

Factor ::= *Identifier*

Factor ::= *Identifier(vertex) - Identifier(vertex)*

Factor ::= - *Factor*

Factor ::= *distance* "(" *Expression , Expression* ")"

Factor ::= *angle* "(" *Expression , Expression , Expression* ")"

Factor ::= *sqrt* "(" *Expression* ")"

Factor ::= *centre* "(" *Expression* ")"

Factor ::= *intersection* "(" *Expression , Expression* ")"

Factor ::= *line* "(" *Expression , Expression* ")"

Identifiers consist of an alphabetic character followed by zero or more alphanumeric characters or underscores. An identifier may be of type vertex, real, angle or distance. An identifier cannot be of type point, because a point is not attached to the paper, and so when a layer of the paper at the place where the point is moves, it is unclear whether the point is on the moved part of the paper or on the unmoved part. Since the endpoints of lines are points, an identifier cannot be a line either. An identifier cannot be of type edge since edges are split during folding; similarly, it cannot be of type face. Therefore, vertices, reals, angles and distances are the only types which can have values with names. There is no need for an additional type which is a point associated with a face (similar to a vertex), since all locations on the paper may be specified in terms of vertices.

Edges are specified as a pair of vertices with a "-" between them.

Unary negation applies to factors of type real, angle and distance. The type of the factor is the same type.

The arguments to "distance" may each be of type point or vertex. The type of the factor is distance.

The arguments to "angle" are points or vertices. The value of angle(p1,p2,p3) is the smallest angle which the line(p1,p2) makes with the line(p2,p3).

The argument to the square root function is of type real. This function is useful if precise calculations are needed, such as in giving the dimensions of the starting piece of paper.

The argument to "centre" is a single face. The return value is the point at the centroid of the face (the average of the corners).

The arguments to "intersection" are lines or edges. The return type is point.

The arguments to "line" are points or vertices. The return type is line.

Faces are handled separately. A face may be specified using any number of vertices, as long as there is one and only one face which the given vertices have in common. Note that a "face" from the user's point of view is bounded by folded edges; there may be creases on the face. The syntax is :

Face ::= *face* "(" *Identifier(=vertex)* { , *Identifier(=vertex)* } ")"

There is one more kind of statement to be introduced : the assignment statement. In order to prevent confusion, assignments may only be made once to each identifier; identifiers represent constants rather than variables.

AssignStatement ::= *Identifier* = *Expression*

The expression may be of type real, angle or distance, for reasons which were discussed previously. There is no reason for the expression to be of type vertex since this would simply create an alias for an existing name.

3.7. Examples

3.7.1. An Example Using Simple Folds

Below are given the folding commands for a model which uses only mountain and valley folds.

```
origami swan
uses 1:1 paper

# This is a swan using only valley and mountain folds,
# taken from "Complete Origami" pp. 147-148 [12] (slightly modified)

# Pre crease the paper with a diagonal fold

fold b to d and return

step 1
fold c-b to c-a creating e on b-a
fold c-d to c-a creating f on d-a


step 2
fold c-e to c-a creating g on e-a
fold c-f to c-a creating h on f-a


step 3
```

fold c to 1.2 along c-a creating i on c-g,j on c-h,k on c-a

step 4
fold c-k to j through i creating l on c-j,m on c-k

step 5
valley fold at 90 degrees to c-m through l moving c creating n on c-i,o on c-m

step 6
fold c-l to c-n

step 7
mountain fold along k-a moving g

step 8
unfold o-c moving l

step 9
valley fold at 45 degrees to n-c through (distance(h,j) along n-c) moving c
creating p on c-n,q on o-c,r on c-l

step 10
fold at 40 degrees to r-c through 1/2 along c-p moving c
creating s on c-p,t on c-r,v on c-q

step 11
fold along s-p moving c creating u on c-t,w on c-v

step 12
valley fold at 70 degrees to c-s through u moving c
creating x on c-w,y on c-s

step 13
fold along c-x moving y
fold along o-q moving l

3.7.2. An Example Using Multiple Folds

Below are given the folding commands for a simple model which uses a multiple fold.

origami pajarita
uses 1:1 paper
plain side up

step 1
fold a to c

step 2
fold b to d creating e on b-d

step 3
fold e to 1/2 along b-c
creating f on a-e, g on b-e, h on c-e, i on d-e
unfold all
plain side up

step 4
mountain fold a to e creating i on a-b, j on a-d
and return

step 5
Collapse the entire model all at once
multifold a-f, i-f, j-f, g-f, h-i, d-i, h-c, b-g,
valley g-h, i-f, mountain h-f,
mountain between b-g, g-h,
mountain between d-i, i-h
holding face(f,h,i)

3.7.3. Standard Folds

Below are some examples of standard folds. The first three are used as a starting point for the others. Completed models corresponding to examples 2 to 8 may be found in the introduction.

1. Book fold

```
origami book_fold
uses 3:2 paper

fold a-b to c-d creating e on a-d, f on b-c
```

2. Preliminary base

```
origami preliminary_base
uses 1:1 paper
plain side up

fold a-d to b-c creating e on a-d, f on b-c and return
fold a-b to c-d creating g on e-f, h on a-b, i on c-d and return
mountain fold a to c and return
mountain fold b to d and return
multifold g-b, g-d, g-e, g-f, g-h, g-i holding face(g,i,c,f)
```

3. Equilateral Triangle

```
origami equilateral_triangle
uses 1:1 paper

valley fold at 60 degrees to a-d creating f on d-c
valley fold at 60 degrees to b-c creating e on a-f
restart with face(a,b,e)
```

4. Inside reverse fold

```
origami reverse_fold
uses "bookfold"

fold along line(1/3 along e-f,2/3 along d-c) moving f
creating g on e-f, h on a-b, i on c-d
and return
multifold e-g, g-i, mountain g-h, reverse g-f holding face(a)
```

5. Squash fold

```
origami squash_fold
uses "prelim-base"

fold e-g to g-a creating j on a-e, k on e-d and return
multifold g-a, g-k, mountain g-j, unfold g-e holding face(c)
```

6. Petal fold

```
origami petal_fold
uses "prelim-base"

fold a-e to a-g creating j on e-g and return
fold a-h to a-g creating k on h-g and return
fold along line(j,k) and return
multifold j-k, j-d, b-k, mountain a-j, a-k, unfold e-j, h-k
holding face(c)
```

7. Sink fold

```
origami sink_fold
uses "prelim-base"

fold g to centre(g,h,a,e) creating j on g-e, k on g-h,
l on g-i, m on g-f, n on d-g, o on b-g
and return
multifold m-l, j-n, o-k, mountain l-n, o-m, j-k,
reverse j-g, k-g, l-g, m-g
holding face (c)
```

8. Rabbit ear fold

```
origami rabbit_ear
uses "eq-triangle"

fold a-e to a-b creating f on b-e and return
fold a-e to e-b creating g on a-f and return
fold a-b to e-b and return
multifold a-g, b-g, e-g, mountain between b-g, e-g
holding face (a,b,g)
```

3.8. Defining Folds

The origami language could be improved greatly by allowing the user to define folds instead of having to explicitly give pre-creasing instructions and all of the edges involved in a multiple fold. Two difficulties which arise are the naming of vertices created during the fold and determining the names of vertices required to do the pre-creasing and any multiple folds. The syntax given in this section has not been incorporated into the program since predefined folds which do not use multiple folds are not very useful, and multiple folds have not been implemented.

A possible syntax for defining a new fold is as follows :

FoldDefinition ::= *DefinitionHeader StatementList end*

DefinitionHeader ::= *define Identifier fold* "(" *Identifier* , { , *Identifier* } ")"

Vertex names mentioned within a fold definition must be local; it does not make sense to use global vertex names. Vertices may be created within the definition which will be referred to later on in the definition. The names given to these created vertices must be local; otherwise different vertices would be assigned the same name. The global names of the created vertices must be assigned when the fold is invoked. Note that a vertex may already exist at the location a vertex is "created". In this situation the name assigned to the vertex is a local alias for the existing vertex, just as vertices given as arguments are aliases.

The arguments to the fold definition need not be restricted to vertices; it may be useful to pass points, reals, angles or distances as arguments. There is no need to pass lines, edges or faces, since these may be constructed from vertices and points. The type of each argument can be determined from its usage within the fold definition.

Consider the following example of a definition :

```
define squash fold(v1,v2,v3,v4)
# v1-v2 is a folded edge, v2-v3 and v2-v4 are raw edges
# v3 is at the same location as v4 and is above v4
fold v1-v2 to line(v1,v3) creating v5 on v2-v3, v6 on v2-v4 and return
multifold v1-v3, v1-v6, mountain v1-v5, unfold v1-v2 holding face(v1,v4,v6)
end
```

For such a predefined fold to be useful, it must be usable on any suitable model no matter where creases already exist on the model. Therefore, the edges referred to in the definition might not be true edges (that is, there may be other vertices between the two endpoints of the edge) but are treated in the definition as though they are.

When the fold defined is used, the vertices created will have to be given global names. For example :

```
origami example_fold
uses preliminary-base

squash fold(g,e,a,d) creating j on a-e, k on d-e
```

In this example the vertex locally named v5 is given the global name j and v6 is given the name k.

Since a fold definition may involve multiple folds, the labelling statement for vertices created during a defined fold is the same as for multiple folds.

If the fold being defined creates a vertex which is not on an edge which exists before the fold is done then it is impossible to name the vertex when the defined fold is invoked, because an edge must be specified in order to name any vertex. As an example, consider a fold which makes two diagonal creases in a square :

```
define cross fold(a,b,c,d)
# a, b, c and d are the corners of a square
# such that a is opposite to c and b is opposite to d
fold a to c and return
fold b to d and return   # This will create a vertex on b-d
```

The vertex created by the second fold in the example cannot be named :

```
cross fold(a,b,c,d) creating e on ... ?
```

The solution is either to make all fold definitions in which creases intersect other creases illegal, which is too limiting, or to give the names of all such created vertices as arguments to the fold. The above example would then become :

```
define cross fold(a,b,c,d,NAME)
# a, b, c and d are the corners of a square
# NAME is the name of the vertex which will be created
fold a to c and return
fold b to d and return creating NAME on b-d
```

An example of the usage of the above fold definition is :

```
cross fold(a,b,c,d,e)
```

Another problem arises if this method of naming these vertices is used. If such a vertex is not named during the definition (that is, its name is not passed as an argument), that vertex can never be named. There are two ways to avoid this problem : either force all such vertices to be named, or allow the definition of the fold to be dynamically changed while the user is entering the instructions for the origami model, in the same way that unnamed vertices may be given names by the user after the vertices are created (and the program inserts the labelling statement for the vertex at the appropriate place in the text).

However, if the definition is changed, all instances of the use of the fold must also be changed (since an extra parameter will be added to the fold definition), and previously unnamed vertices will have to be named. As with the similar situation involving previously unnamed vertices in normal folds, such vertices could either be forced to be named by the user or could be assigned names generated by the program. Another implication of the definition being able to be changed is that the instructions given by the user must use a local version of the definitions used, and cannot "include" a copy of them, since they may be modified.

There is one other problem : as was mentioned previously, a vertex which appears to be "created" within a fold definition may already exist; if the vertex is already named, it will be assigned a second name. In order to prevent this, if the vertex already exists and is named then its existing name should be passed as the argument.

An alternative to the above method of naming vertices created on creases formed during a predefined fold is to make all fold definitions have optional arguments, one for each vertex which might need to be named. This would require some means of making clear which created vertex an argument corresponds to. An example use could be : "complex fold(a,b,c,d,label3=e,label6=f,label8=g)", meaning that the local names label3, label6 and label8 would correspond to global names e, f and g, respectively (note that the local labels would have to emerge from their domain to be able to do this). This appears to be a better method than the previous one. When the fold is defined, a semicolon could separate the optional arguments from the normal arguments. The syntax for fold definitions becomes :

FoldDefinition ::= *DefinitionHeader StatementList end*

DefinitionHeader ::= *define Identifier fold* "(" *Ident* { , *Ident* } [; *Ident* { , *Ident* }] ")"

It is also possible that with certain arguments, different vertices will be created or no vertices will be created at all. For example,

cross fold(a,b,d,c,e)

is illegal if (a,b,c,d) are the corners of a square in clockwise order, because the vertex to be named "e" will not be created during the fold. This is not a real problem, because the arguments to the fold must be assumed to be correct.

There is a problem related to specifying the vertices required to do the folds in a fold definition. For example, here is the definition of an inside reverse fold :

```
define inside_reverse fold(a,b,c,d,e,f,p1,p2)
# p1 is a point along the folded edge a-b
# p2 is a point along edge e-f
# The reverse fold will occur at p1 and will form an edge
# from p1 to p2 and an overlapping edge from p1 to edge c-d
# and will reverse the edge from p1 to b.
# face(a,b,f,e) is assumed to be above face(a,b,d,c)
fold along line(p1,p2) and return
creating p1vertex on a-b, p2vertex on e-f, p3vertex on c-d
multifold p1vertex-p3vertex, mountain p1vertex-p2vertex, reverse p1vertex-b
holding face(a,p1vertex,p2vertex)
end
```

The problem is that a reverse fold should intuitively be able to be specified using only two points and an edge (p1, p2 and a-b), but eight arguments are needed to specify the crease and the multifold. A solution would be to add operations for actions such as traversal of edges and faces. For the reverse fold, useful operations would be finding the adjacent faces to an edge, traversing the edges of a face and determining whether a line intersects an edge; such a language would include iteration, variables and conditional statements. The language would also be unwieldy and difficult for most people to follow or to write in, which does not correspond with the aims of the project. However, it would also overcome another problem : it would enable fold definitions to be created which are very general; for example, sink folds which would work on any interior vertex, not just vertices which have a fixed number of coincident edges.

Fold definitions should be able to be "included" in the instructions for an origami model rather than needing to be rewritten for each model. A simple syntax would be to allow the following statement anywhere a normal statement is possible :

IncludeStatement ::= *include* "*String*"

The included file is expected to have only fold definitions and/or "include" statements in it.

The scope of included fold definitions would be from the include statement to the end of the file. Fold definitions may be nested, with local scope (that is, up to the end of the enclosing definition). If a fold definition is multiply defined then the latest definition is used; this allows different files of definitions to be used which have some folds in common.

Chapter 4. A Comparison Of Folding Instructions

In this chapter four different methods of giving instructions for the traditional Japanese "Flapping Bird" are given : the standard method using text and diagrams, using the origami language, a natural language description, and as a crease pattern. The natural language description is given to show how difficult it is to specify folds without using labels for parts of the paper, and the method does not correspond exactly with the others. The steps using the traditional method and using the origami language correspond exactly. There is a discussion following the four methods.

4.1. Flapping Bird : Standard Method

4.2. Flapping Bird : Using The Origami Language

In the instructions given below, all of the instructions are expanded in full. It would be possible to eliminate steps 1-5 by starting with the statement : uses "preliminary-base", and to simplify steps 6-8 and 9-10 by making use of a predefined petal fold and reverse fold, respectively. This would reduce the instructions to six statements : two petal folds, three reverse folds and a "turn over" statement.

The step numbers correspond to the step numbers of the instructions given using the standard method.

```
origami bird
uses 1:1 paper
plain side up

step 1
fold d to b and return

step 2
fold c to a creating e on a-c and return

step 3
fold c-d to a-b creating f on b-c, g on a-d and return

step 4
fold b-c to a-d creating h on a-b, i on c-d and return

step 5
# (Collapse to form preliminary base)
multifold e-a, e-c, e-f, e-g, e-h, e-i holding face(e,i,d)

# Fold left and right flaps to the centre line
fold b-f to b-e creating j on e-f
fold b-h to b-e creating k on e-h

turn over

# Repeat on other side
fold d-g to d-e creating l on e-g
fold d-i to d-e creating m on e-i
```

step 6
fold along line(l,m) creating x on e-c, y on e-a and return

step 7
Unfold the left and right flaps
unfold l-d moving g
unfold m-d moving i
(Petal fold)
multifold l-m, a-l, c-m, mountain l-d, m-d, unfold l-g, m-i holding face(l,e,m)

turn over

Repeat on other side
unfold j-b moving f
unfold k-b moving h
multifold j-c, a-k, valley j-k, mountain j-b, k-b, unfold j-f, k-h
holding face(j,e,k)

step 8
fold along line(h,4/5 along a-k) creating n on a-k, o on a-l and return
Inside reverse fold
multifold o-y, mountain h-n, reverse y-a holding face(h,b,k)
Repeat on other half
fold along line(f,4/5 along c-j) creating p on c-j, q on c-m
multifold q-x, mountain f-p, reverse x-c holding face(j,b,f)

step 9
fold along line(2/5 along a-y,1/3 along a-n) moving a
creating r on a-y, s on a-n, t on o-a and return
Inside reverse fold to form the head
multifold r-t, mountain r-s, reverse r-a holding face(k,b,h)

4.3. Flapping Bird : Natural Language Description

Below is a copy of Lillian Oppenheimer's instructions for making a traditional Japanese flapping bird, from "The Art of Origami" [19]. It is meant to be understandable without visual aids. The step numbers do not correspond with the step numbers of the other methods given.

1. Hold your square of paper before you like a diamond. Bring the lower point to the upper point, making a diagonal crease. Open the paper and rotate it clockwise to bring the crease straight up and down.

2. Again lift the lower point to the upper point, making a second diagonal crease. You now have two valley folds crossing each other in the middle. Turn the paper over so that the other side faces you.

3. Fold the paper in half like a book, bringing the lower edge to the upper edge. Make a sharp crease. Open the paper and rotate it clockwise.

4. Repeat the book fold, bringing the lower edge to the upper edge again. Open the paper. Now check to be sure that you have made the fold correctly : do the creases alternate - valley, mountain, valley, mountain ?

5. Fold the top edge down to the bottom edge, so that the book fold runs along the top.

6. Grasp the extreme ends of the book fold. Push your hands together until the four corners meet at the bottom.

7. You should now have a shape that looks like a top or an umbrella. Take two of the four double corners in each hand, two and two. Hold the paper with its solid point away from you, and flatten it on the table, creasing it well. Take the double corner nearest you on your left and lay the open edges on the centre line. Do the same thing on your right to make a shape that resembles an ice cream cone: a cone below and a triangle of ice cream at the top. Turn the paper over. Make sure the solid point is still away from you. Repeat the ice cream cone fold.

8. Grasp the ice cream (the triangle at the top) and fold it toward you; then lift it back into place. Open the right and left sides of the cone slightly, like the doors of a cupboard.

9. Run your right index finger down between the doors to find the hidden point at the bottom. Lift up this point slightly.

10. You can now see ice cream cone creases in the next layer of paper. Place your left index finger at the bottom of these creases to hold them together, and with your right hand continue to lift the point. Lift it all the way up, so that the two raw edges come together in the centre, forming a wing. Repeat the same thing on the other side to make a second wing.

11. Hold the paper so that the two wings are up and the divided part downward. There are two blunt points on your left and two on the right. Take the one nearest you on your left and fold it over to the right like the page of a book. Repeat this on the far side of the figure.

12. Notice that the top is now divided and the bottom smooth. The wings are down below. Lift them upward, one in front and one in back.

13. You now have four points at the top. Grasp the inner point on the left side and pull it downward and to the left to make the neck. Set the angle permanently by creasing it well underneath. Do the same thing on the right to make the tail.

14. Hold the base of the neck firmly between your left thumb and your index finger, and pull the tail with your right hand. The wings should move.

15. The wings sometimes catch at the centre of the body, and to free them you may have to work the tail section straight up and down. Fold the head downward, making a preparatory crease, and unfold it again.

16. Place your left thumb in the groove of the neck, and with your index finger pull the head down between the sides of the neck as far as the preparatory crease. Curl the wings slightly to complete the bird.

17. Pull the tail and the bird will flap its wings.

4.4. Flapping Bird : Crease Pattern

Over the next two pages is a copy of a file giving the crease pattern for a flapping bird with its wings up. The file is the input to a program which folds a crease pattern, given the vertices, edges and faces of the model, which face is held still during the folding and zero or more relationships between faces to resolve any ambiguities in the layering of the faces after the fold.

The crease pattern represented is as follows :

Vertices And Folds *Faces*

The completed model is given below.

```
# Vertices

# Format:
# id  x,y EdgeType(r/v/m) AdjVert Face r/v/m AdjVert Face ..
# (Face information is redundant but useful for error checking)

0  0,1          r4 10 m9 7 m12 11 r7
1  1,1          r22 22 m20 20 v19 21 m21 23 r23
2  1,0          r6 9 m11 6 m10 8 r5
3  0,0          r7 15 m16 13 m14 12 m15 14 r6
4  .5,1         r23 19 m18 10 r0
5  1,.5         r2 8 m17 19 r22
6  .5,0         r3 14 m15 9 r2
7  0,.5         r0 11 m16 15 r3
8  .5,.5        m9 3 v13 2 m10 0 m11 4 v14 5 m12 1
9  .5,.7929     m18 3 m8 1 v12 7 m0 10
#      ^-(3-sqrt(2))/2
10 .7929,.5     m17 8 m2 6 v11 0 m8 2
#   ^-(3-sqrt(2))/2
11 .5,.2071     v10 6 m2 9 m15 4 m8 0
#      ^-(sqrt(2)-1)/2
12 .2071,.5     m16 11 m0 7 v9 1 m8 5
#   ^-(sqrt(2)-1)/2
13 .6464,.6464  m19 15 v17 2 v8 3 v18 17
#   ^-----^--(1-1/sqrt(8))
14 .3536,.3536  m3 13 v16 5 v8 4 v15 12
#   ^--^-(1/sqrt(8))
15 .4,.1657     v14 4 m11 9 m6 14 m3 12
#      ^-(sqrt(8)-2)/5
16 .1657,.4     v14 13 m3 15 m7 11 m12 5
#   ^-(sqrt(8)-2)/5
17 .8343,.6     m5 8 m10 2 v13 16 m20 18
#   ^-(7-sqrt(8))/5
18 .6,.8343     m4 19 m21 17 v13 3 m9 10
#      ^-(7-sqrt(8))/5
19 .8586,.8586  v1 20 m20 16 m13 17 m21 21
#   ^--^-(1-1/sqrt(50))
20 .9448,.8667  v22 18 m17 16 m19 20 m1 22
#   ^-13/15 ^-((sqrt(8)-2)/15)
21 .8667,.9448  v23 23 m1 21 m19 17 m18 19
#   ^--^-(same as above)
22 1,.8        r5 18   v20 22 r1
23 .8,1        r1 23   v21 19 r4
end
```

```
# Faces

# Format:
# id  Vertex Vertex Vertex.. (clockwise around the face)

0  8 10 11
1  9 8 12
2  13 17 10 8
3  9 18 13 8
4  8 11 15 14
5  12 8 14 16
6  10 2 11
7  0 9 12
8  17 5 2 10
9  11 2 6 15
10 0 4 18 9
11 0 12 16 7
12 14 15 3
13 16 14 3
14 15 6 3
15 7 16 3
16 17 13 19 20
17 18 21 19 13
18 22 5 17 20
19 21 18 4 23
20 20 19 1
21 19 21 1
22 22 20 1
23 21 23 1
end

# Which face is unmoving (held still while the other faces
move)

0

# Relationships between faces (to make the folds unambiguous)
# Format:
# A,B
# C,D ..
# (means that A is above B and C is above D)

14,15
```

4.5. Discussion Of The Instruction Methods

Out of the four methods of giving instructions, the crease pattern may generally be stored the most efficiently, but it is difficult to understand and gives no information about the folding sequence. However, if it is just the final model which is to be recorded and not a particular sequence of folds, then a program could be written which would determine a folding sequence for the model, and so storing a model's crease pattern could be the most efficient method of the four. Note that sometimes the large number of vertices, faces and edges of a model could make other methods much more concise; for example, folding a square vertically in half 10 times (assuming this to be possible) would create a model with 1024 faces, 2050 vertices and 3073 edges, which would take more room to store using the crease pattern than it would using any of the other methods, unless the symmetrical nature of the model was taken advantage of to store it more efficiently.

The natural language description is probably the clearest of the four, but at the expense of verbosity; nine separate sentences are used to describe a petal fold, while three simple folds and one multiple fold are enough to describe it using the origami language. All complex folds suffer from this problem. An advantage of using natural language is the ability to use the "repeat" instruction, which requires the ability to generalize the previous fold and apply it to a different situation.

The standard method using diagrams and text ought to be the clearest and simplest of all : standard terminology makes descriptions of complex folds short and simple, and the diagrams may be used to show locations on the paper which are difficult to describe. However, it is not very easy to describe non-standard folds; the diagrams cannot show fully what is happening, as Robert Lang mentions in the introduction to his book "The Complete Book Of Origami" [13], and so the text must be used to describe the fold in terms of parts of the paper, and the same difficulty arises as with the natural language description. Non-standard folds do not cause these problems using the origami language.

If the folder has access to only the written instructions, the natural language description and the standard method using diagrams would be simpler to understand than the origami language; however, the language is meant to be used in conjunction with a computer program. When this is done, the folder is free to move backwards and forwards through the folding sequence and is free to examine the model in various ways; the model is not static as it is in diagrams. This interactive method of giving origami instructions is much more understandable than any of the other methods. Understanding the origami language without the aid of a computer program is not difficult after the names of the vertices have been determined.

For purposes of recording the folding sequence in the first place, without the aid of a computer the origami language would be more difficult to use than any method except for finding the crease pattern, which often involves complex calculations. However, when used in conjunction with the program, descriptions are simpler to make than using natural language. If only standard folds are used then using origami diagrams might be simpler (apart from the effort of drawing the diagrams), but the origami language is definitely superior if non-standard folds are done.

Chapter 5. Effects Of Folding And Tucking

5.1. Effects Of Simple Folds

5.1.1. Moving And Folded Faces In Simple Folds

When a simple fold is made, part of the paper moves and the rest of the paper is held still. The creases form on the faces which have part of the face moving and the other part held still. Moving some faces will force other faces to move as well. Below an algorithm is given to find the faces which move, given part of the paper which moves, the fold line along which creases appear and the type of fold (mountain or valley).

The algorithm assumes that the fold is being done "on the table" rather than "in the air"; the latter case is more complicated and will sometimes be ambiguous.

Definition

Face A is attached to face B if moving face B makes face A (or a part of face A) move too.

Face A is attached to face B if either of the following is true :

1. A and B have a common edge which is partially or completely on the same side of the fold line as the moving part of the paper.
2. A and B overlap partially or completely on the same side of the fold line as the moving part of the paper, and either A is above B and the fold is a valley fold, or A is below B and the fold is a mountain fold.

The input to the algorithm is the current state of the origami model, the type of fold (valley or mountain), a moving part of the paper and the fold line. The moving part of the paper will be a vertex. If an edge is specified as the moving part of the paper, one of the vertices at the endpoint of the edge is used. If the fold being done is of the form "fold A-B to C through D" then the vertex used is whichever of A and B is on the opposite side of the fold line to D. The output is a set of faces which will move and a set of faces which will have folds in them or which will have one of their edges folded.

Let *F* be the set of faces which will have folds in them or have an edge on the fold line and let *M* be the set of faces which will move. An algorithm for finding *F* and *M* is as follows :

1. For each face which is adjacent to the moving vertex, if the fold line intersects the face, or the fold line is on a non-raw edge of the face, add it to *F,* otherwise add it to *M.*

2. Choose any face which is attached to a face in either *F* or *M* and which is not in either *F* or *M.* If the fold line intersects this face, or the fold line is on a non-raw edge of this face, add it to *F,* otherwise add it to *M.* Repeat this until there are no more faces which are attached to a face in *F* or *M* and which are not in *F* or *M.*

This will result in *M* containing all of the faces which will move and *F* containing all of the faces which will have folds in them, including faces with a non-raw edge on the fold line (and so are moved but are not split into two faces).

The reason why this algorithm works is as follows: all of the faces which are adjacent to the moving vertex must move when it moves. No part of the paper on the opposite side of the fold line can move. The paper becomes folded where one part of a face is found to be moving and another part is found to be unmoving; on such faces, the fold will appear along the fold line. If a face or part of a face is found to be moving then any face which is adjacent to the moving part of that face will also move. If a face is above a moving part of the paper in a valley fold or below in a mountain fold then it will be forced to move when the moving part of the paper moves. The algorithm determines which faces move or partially move, and then finds which other faces must also move or partially move. This is repeated until all such faces have been determined.

Some examples are illustrated below :

Moving faces

Folded faces

Moving part of the paper

5.1.2. Doing Simple Folds

During a simple fold, vertices are moved, vertices are created, faces reverse direction, faces and edges are split in two, edges change their fold type and there are changes to the layering relationships between faces.

The vertices which move are the vertices of the moving faces; this will include some of the vertices of the split faces (the faces which have a fold in them). The new location of a vertex which moves is the reflection of its old location about the fold line.

Faces which are split in two will have an edge between two points on the face, each of which will either be an existing vertex or a point on an edge of the face. In that case, the edge will be split and will be replaced with two new edges.

Every face which moves will reverse its direction (that is, which side of the paper is the plain side). The moving part of split faces will face in the opposite direction to the original face.

If there is an edge which is on the fold line and is between a moving and an unmoving face, the fold type of the edge may be changed. If the edge was folded, it will become unfolded but will retain the same fold type. Otherwise, the edge is a crease and it will be folded with the appropriate fold type.

The layering relationships between all moving faces are reversed. The relationships between moving faces and unmoving faces are changed completely, since some faces will no longer overlap other faces and will overlap new faces. Moving faces become above the new faces they overlap in a valley fold, or below in a mountain fold.

Faces which are split may be handled in two steps : first create the two new faces of every face which is split, then do the same thing as for other moving faces. The layering of the new faces can be found as follows :

The relationship (above, below or "unrelated") between each half of the split face and the faces the original face was above or below is changed to "unrelated" if that half of the face does not overlap them. The relationships between the halves of different split faces will be the same as for the original pair of faces before they were split, except that the moving halves will become unrelated to the unmoving halves (since they are on opposite sides of the line).

5.2. What Happens When A Crease Pattern Is Folded

This section applies to multiple folds where the paper is not initially folded, but is creased in preparation for a multiple fold; in other words, the entire model is being folded all at once. The general case will be the subject of the next section.

The problem may be restated as follows : given a crease pattern with the location and kind of each fold (mountain or valley), a single fixed face (ie. a face which is held still) and some layering relationships between faces (to resolve ambiguities), determine the new location of each vertex, the direction each face is facing and the layering relationship between the faces after all the folds are made. It is assumed that every edge is folded.

Origami has the surprising property that for every possible assignment of mountain and valley folds in a given crease pattern, the final location of each vertex and the direction each face is facing will be the same; the mountain and valley folds only determine the layering of faces.

If two faces are joined by a folded edge, the faces will face in the opposite direction, whether the fold is a valley fold or a mountain fold. Since all of the edges are folded, this means that the faces of a flat origami model are 2-colourable, and all faces of the same colour face in the same direction. This can also be shown using the fact that the degree of every interior vertex is even, by theorem 3 (see [7]).

Consider two faces A and B, their common edge E and a vertex V which is on face A. Relative to face B, the location of V after E is folded is its reflection about the line which E lies on, whether the fold is a valley fold or a mountain fold. Therefore, the following algorithm will give the final location of any vertex after all of the folds are made, relative to the fixed face : let V be the vertex and F be the fixed face. Select any face which V is on. Choose an adjacent face. Reflect V about the line on which their common edge lies. Repeat this until the face F is reached. V will now be at the location it would be at after the fold was made.

The directions of all of the faces and the final location of every vertex may be efficiently calculated as follows : create a breadth-first search tree of the faces, starting from the fixed face. Faces at an odd distance from the root will be facing in the opposite direction to the fixed face (since in a 2-colouring of the faces as described above they will be given a colour opposite to the fixed face). Maintain a set of vertices which have been visited When a new face is added to the tree, add any unvisited vertices of the face to the set and follow the links in the tree back to the root, reflecting the vertices about the edges between adjacent faces as the tree is traversed. This will give the final location of each vertex.

An example is given below. The first diagram shows the crease pattern of the "pajarita" (little bird). The second diagram shows a breadth first search tree with the central white triangle as the fixed face. Faces at odd distance from the root are shaded; these will all be facing away from the viewer after the fold is made. The third diagram shows the result of the folds and the final locations of the vertices.

To determine if a crease pattern is potentially foldable, the results of theorems 2 and 4 are required : for an origami model to fold flat, the difference between the number of mountain folds and the number of valley folds at every interior vertex must be 2, and the sum of alternate angles about every interior vertex must be 180°. If these two conditions are met, the model may still be unfoldable : it is also necessary for there to be no "tearing" of the paper.

To determine of the model is foldable, unfoldable or ambiguous, inferences must be made about the layering of the faces. The mountain and valley folds will determine the layering relationships between all adjacent faces as follows : Let A and B be adjacent faces. If A is facing in the same direction as the fixed face then B will be above A if the fold is a valley fold, and A will be above B is the fold is a mountain fold. If A is facing in the opposite direction to the fixed face then the opposite will be true.

The layering relationships between all non-adjacent overlapping faces must be found to determine the final state of the model. If there are some relationships which cannot be found, the model is ambiguous; if there is a contradiction in which A is found to be above B and B is found to be above A, the model is unfoldable.

There are three inferences which may be made about the layering of the faces :

1. If (face A is above face B) and (face B is above face C) and (faces A, B and C overlap) then (face A is above face C). The reason for the third part of the condition is that it is possible for three faces to cyclicly overlap each other but for there not to be an area in common with all three, as is shown in the diagram below :

2. Let faces A and B be adjacent edges and E be their common edge. Then if E overlaps some face C, (C is above A and B) or (C is below A and B). This situation is illustrated in the diagram below :

3. Let faces A and B be adjacent faces with common edge E, and C and D be adjacent faces with common edge E2. Then if edges E and E2 overlap and (A is above B) and (C is above D) then (A and B are both above C and D) or (A and B are both below C and D) or (C is above A which is above B which is above D) or (A is above C which is above D which is above B). This situation is as illustrated in the diagram below :

To determine the layering, these three inference rules may be applied repeatedly until there is a contradiction or until no more inferences can be made. If every pair of overlapping faces is assigned a relationship by the rules then the model is unambiguous.

In order to determine if three faces all overlap in a common region, it is useful to be able to determine if one face is completely within another face. If it is known that three faces pairwise overlap each other and one face is completely within either of the other two, then all three must overlap, and it is not necessary to do any clipping to determine if they do. Otherwise one face must be clipped against another and a test must be done to determine whether the remaining face overlaps the region of intersection.

Care must be taken when determining whether one face overlaps or is inside another face, since the highly symmetrical nature of many origami models makes borderline cases common. For example, edges will overlap other edges and different faces may have several points in common. Some of these special cases are diagramed below :

Note that the "unfold all" statement may be implemented in a similar but simpler way, since unfolding the model is really a kind of multiple fold in which every edge is unfolded. Unfolding every face is simpler because layering relationships will not be required. The fixed face may be any face which is facing the direction which the paper is meant to be facing after the fold, or alternatively, any face may be the fixed face, and if it is the wrong way up then a "turn over" may be done afterwards.

5.3. What Happens During Multiple Folds

The problem of determining the effects of multiple folds on paper which is already folded is more complicated, since it must be determined which parts of the paper are forced to move when the fold is done.

The first part of the procedure for determining the effects of multiple folds is as follows : check if the multiple fold being attempted is potentially foldable using theorems 2 and 4 just as was done in the description of how to fold a crease pattern, and at the same time determine the locations of any edges which were given as being "between" two other edges in the multiple fold command.

If a vertex is mentioned only once in the list of edges which are folded during the multiple fold (including any vertices which will be on an edge between two given edges) then a simple fold will pass through that vertex when the fold is done.

If a vertex is mentioned more than once, then it will not be a simple fold that is done at that vertex when the fold is done, and if the vertex is an interior vertex, then theorems 2 and 4 may be used to check if the conditions for interior vertices to fold flat are satisfied, as follows :

To determine the difference between the number of mountain folds and the number of valley folds which will be about the vertex after the fold is done, add up the number of existing mountain folds and subtract the number of existing valley folds which are currently folded and which will not become unfolded when the multiple fold is done, then add to this the number of mountain folds minus the number of valley folds which will be done, then add twice the number of valley folds which are reversed minus the number of mountain folds which are reversed. For the model to fold flat, the result must be plus or minus two.

To determine whether alternate angles about the vertex add up to 180°, as theorem 4 requires, include only the angles between edges which are either currently folded and do not become unfolded during the fold and edges which become folded during the fold. If an edge will appear between two given edges during the fold, the location of the edge may be determined at this stage using the other angles about the vertex.

Once the multiple fold has been determined to be potentially foldable and the locations of all of the edges involved in the fold have been found, the moving and folded faces may be determined. A complete algorithm for determining the moving and folded faces in a multiple fold has not been discovered. The algorithm would have to incorporate parts of the algorithm for doing simple folds and the algorithm for folding a complete crease pattern. It is not straightforward to determine which parts of the paper move in a multiple fold if more than one layer of paper is involved, and there are many cases in which the layering after the fold is done will be ambiguous. There are also some cases which cannot be described by the existing syntax; for example, folding a 2:1 rectangle vertically in half then creating a waterbomb base on the top face only leaving the bottom face unfolded (if the existing syntax would do this, then there is no way to describe folding the waterbomb base in both layers in a single fold).

5.4. What Happens During Tucks

A tucking command, which will be one of *tuck Face under Face* or *untuck Face,* never creates new faces or edges, but only affects the layering of the faces.

After the command "tuck A under B", the layering will change so that A is under B but is as close as possible to B. The command "untuck A" will change the layering so that A has the minimum possible number of faces above it without the paper tearing. In both cases the layering of every pair of adjacent faces due to mountain and valley folds will be unchanged. Faces A and B may be any pair of overlapping faces. The "tuck" command is illegal if A cannot be tucked under B; the "untuck" command is illegal if there are no changes to the layering after it is done.

A complete algorithm for handling tucks has not yet been discovered. The algorithm could not simply change the layering so that A was under B and then determine which faces must have been also forced to be under B and which faces attached to B must be above A, since sometimes the tuck would still be physically impossible without making a face pass through another face.

Chapter 6. Data Structures

The entire data structure consists of the current state of the model, a list of differences between steps, the text of the origami instructions, a list of fold definitions and the initial state of the model. The initial state is required so that the folding sequence may be replayed.

6.1. Representation Of The Origami Model

The data structure must be able to represent the locations of the vertices, faces and edges of the model, adjacencies, which faces are above or below which other faces, kinds of folds in the edges (or whether the edges is a raw edge) and which direction each face is facing.

The state of a model is represented by a vertex table, an edge table, a face table, the layering relationships, a symbol table and the current viewpoint.

Vertices

Associated with each vertex is its id number (the reason for this will be given later), its location (coordinates), a list of adjacent vertices and the symbol table entry of the vertex (if there is one). The edges coincident with a vertex may be determined using the adjacent vertices, and these edges may in turn be used to determine the faces adjacent to a vertex. The vertices are stored in a vertex table which is accessed through a vertex id.

Because vertices may be named by the user at any time and the program must be able to "go back" and insert the labelling command, also associated with each vertex are some other values : the edge which the vertex was created on (and an ordinal value if it was a multiple fold which created the vertex and there was more than one vertex created on that edge by the fold), a pointer to the line of text containing the folding command which created the vertex, a pointer into the difference list to the name assigned to the vertex and a list of pointers to lines of text in which the label appears. This last list is used if the vertex is renamed.

The vertex table has a "next available id number" which will be one more than the highest id number assigned so far to a vertex.

The vertex table is implemented as a hash table, hashed on id. Since ids are assigned in order with no deletion, the hash table is optimally time efficient if the hash function is the id modulus the size of the hash table.

Edges

Associated with each edge are the two vertices at its endpoints, the type of fold in the edge (mountain or valley with respect to the plain side of the paper, or the fact that the edge is a raw edge), whether the edge is folded or is just a crease, and the two faces adjacent to the edge (one if the edge is a raw edge). Edges are stored in an edge table which is accessed using a pair of adjacent vertices. The edge table is implemented as a hash table, with a function of the pointer values of the two end vertices as the hash function.

Faces

Associated with each face is its id number, a list of the vertices of the face in clockwise order with respect to the plain side of the face and the direction the face is facing (whether or not the plain side of the paper is facing "up"). Faces are stored in a face table which is accessed through a face id. The face table has a "next available id number" which will be one more than the highest id number assigned so far to a face. The face table is implemented as a hash table, hashed on face id. This is optimally time efficient for the same reason as the vertex table.

Layering

A separate structure is needed to represent layering. If the type of fold between each pair of adjacent faces (mountain or valley) is known, only a few explicit relationships between faces are needed to make the layering unambiguous, but storing only these would mean that the layering would need to be recalculated at each step; a more efficient way of representing layering is to store the relationship between every pair of faces : above, below or unrelated (if there is no overlap between the faces). Note that cyclic layering relationships can occur in origami (for example, if A is above B which is above C which is above A), and so it would not be possible to simply give each face some kind of "layer number" to indicate which faces it is above and below. The layering is implemented as a two dimensional boolean array, where an entry of 1 indicates that the face with the first index as its id is above the face with the second index as its id.

Symbol Table

The symbol table stores the names and values of vertices and constants (reals, angles and distances). The symbol table is implemented as a hash table, hashed on the symbol name.

Viewpoint

The viewpoint consists of a three coordinates. The x and y coordinates represent the point at the centre of the display, and the z coordinate gives the scale with which the model is drawn The z coordinate gives the "up" direction of the paper : a positive z-value indicates that the paper is facing upwards. Only orthogonal or close to orthogonal views are permitted, since the model will be drawn with a false perspective rather than using actual three dimensional coordinates for points. This is because in some models it is impossible to assign three dimensional coordinates to the vertices without causing faces to interpenetrate, as is discussed in chapter 7.

6.2. Representation Of The Folding Sequence

A simple but inefficient way to store the folding sequence would be to store the complete model after each new fold. Since the model before the fold has a lot in common with the model after the fold, a better method is to store only the differences. For each kind of statement in the origami language there is a corresponding difference : turn over, restart with a face, assignment, unfold all, simple fold (including unfold and crease), multiple fold and tuck. A difference list consists of a list of individual differences for each step, along with a pointer to the corresponding command in the text. The list of differences is actually a list of sublists of differences, one sublist for each step, but it is effectively a single list. Associated with each sublist is a step number.

The reason why identification numbers are needed for faces and vertices is that references must be stored in the difference list to faces and vertices which may not exist in the current state of the model, such as when the folding sequence is being replayed from the beginning. Edges do not need identification numbers because they can be stored as pairs of vertices.

The information stored in the various kinds of difference are as follows :

No additional information is stored for "turn over".

For "restart" statements, the ids of all of the vertices of the face restarted with are stored, along with the direction the new face is facing (plain side up or down).

For assignment statements, the identifier name and the type and value it will be assigned are stored.

For "unfold all", the new positions of all vertices, the direction the faces will all be facing and the new viewpoint are stored.

There is a lot of information associated with simple fold differences : whether it is a fold or a crease, the number of faces which will be split in two (so that the size of the new layering may be determined), a pointer to the line in the text which the fold corresponds to, a list of vertices (stored as ids) and the locations they move to, a list of vertices which will be created (given as name and location), a list of faces which move during the fold but on which a fold does not appear (along with lists of the unmoving faces which it will become above or below), a list of faces which do not move and a list of faces on which a fold appears (including faces which move during the fold and have an edge on the fold line, and so are not split in two).

For each face on which a fold appears, the following information is stored : the id of the face, the type of fold which is made, whether the fold is made along an existing edge of the face, the faces the unmoving part of the face becomes unrelated to (ie. faces which the original face overlaps but which the new face does not overlap), the faces the moving part of the face becomes unrelated to, unmoving faces which the moving part of the face becomes above or below (depending on the type of fold), and the locations of the endpoints of the edge which is created on the face. Each endpoint is given as the id of the vertex, the ids of the endpoints of the edge which is split by the fold (if the vertex does not already exist), and the ids of the faces which will be adjacent to the two new edges which will replace the existing edge (if the vertex does not already exist). When the fold is done, the moving half of the face will get the next available face id and the unmoving half will get the same id as the original face.

Multiple folds have not been implemented. The kind of information stored in a multiple fold difference would be similar to the information stored for a simple fold.

"Tuck" and "untuck" statements are not implemented. The difference for both kinds would consist of the changes to the layering after the tuck or untuck statement.

Fold definitions are treated as macros with values being substituted for parameters when used, so they are stored as text. The parameter list is stored two lists of arguments, the second list containing the optional ones. A fold definition difference consists of a pointer to the definition.

6.3. Effects Of Commands On The Data Structures

This section traces the effects of parsing and executing the commands in the origami language on the data structures. A complete syntax for the language may be found in Appendix B. The effects of multiple folds and tucking statements are not included, as algorithms for these kinds of operations have not been discovered.

The input to the parser consists of a stream of characters, a flag indicating whether a difference list is to be created and the maximum step number to parse up to. These are required because the the parser may be called recursively on another file with the "uses" command, in which case a difference list will not be created and there may be a maximum step number.

The output is the initial state of the model, the final state of the model after the last step and a list of differences.

The following sections give details of the effects of the various parts of the syntax on the list of differences and the initial and final states of the model.

Effects Of The Model Header

Parsing and executing the header (including the "uses" part) gives the initial state of the model. The initial model is then copied to the final model, and the step list is parsed and executed, creating the list of differences and modifying the final model to its state after the final step.

Effects Of The "Uses" Part

A "uses" statement will either give the width and height of the initial piece of paper to use or the name of a file containing another model to use as a starting point. The effect of parsing and executing the "uses" part is that a new model is created.

If the dimensions of the paper are given then the following occurs. If a Viewside (such as "plain side up") follows then it is parsed and the initial direction of the paper is determined, otherwise the initial direction is assumed to be "plain side up". The width and height are normalized so that the maximum value is 1.00. The model is given the following values :

Vertex table: 4 vertices are created with id numbers 0 to 3, clockwise with respect to the plain side of the paper, with x-coordinates equal to *width* or *-width* and y-coordinates equal to *height* or *-height* such that vertex 0 is at (*-width*,*height*) if the plain side is up or (*width*,*height*) if the plain side is down. Vertex 0 is made adjacent to 1 and 3, 1 to 0 and 2, 2 to 1 and 3 and 3 to 2 and 0. All pointer values associated with each vertex are given the value NULL except for the symbol table pointers, which point to the appropriate entries. The next available vertex id number is given the value 4.

Symbol table: symbols ’a’, ’b’, ’c’ and ’d’ are created with values vertex 0, vertex 1, vertex 2 and vertex 3, respectively.

Edge table: raw edges (0,1), (1,2), (2,3) and (0,3) are created, each adjacent to a single face with id 0.

Face table: a single face with id 0 is created with vertices 0, 1, 2 and 3 and the appropriate direction (plain side up or down). The next available face id is given the value 1.

Layering: no relationships between faces.

Viewpoint: equals (0.0,0.0,1.0). The x and y coordinates represent the point at the centre of the screen, and the z coordinate represents the scale to use when the model is drawn (not the actual location of the eye).

If the "uses" statement gives the name of a file containing another model then the following occurs. The character input function is changed to a function to return the next character in the file and the parser is called recursively with differences not being saved and the maximum step number set to the one given in the "uses" statement, if present (a step number of 0 will make it parse up to the last statement before the first step). Vertices which are created during the parsing will have the pointers to the line they were created on set to NULL and cannot be renamed. The character input function is then restored to its previous value and the model is returned.

Effects Of Step Lists

Parsing and executing a step list results in a difference list being created, which is made up of the results of parsing and executing statement lists. Step numbers are inserted in the difference list at the start of each step; if there is a statement list before the first step then it is assigned a step number of 0. The difference list is not created if the differences are not being saved. Parsing terminates after the step with the maximum step number, if there is a maximum step number.

Effects Of Statement Lists

Parsing and executing a statement list results in a list of differences being created unless differences are not being saved. Each difference is the result of parsing and executing a statement.

Effects Of Statements

Parsing and executing a statement results in a difference being created. The difference is then applied to the final model. If the differences are not being saved then nothing is returned, otherwise the difference is returned.

Effects Of Viewpoint Change Statements

Parsing and executing a viewpoint change statement, that is, "turn over", creates a "viewpoint" difference (in which no values are stored).

Effects Of Restart Statements

Parsing and executing a restart statement creates a "restart" difference, consisting of the vertices of the given face and the direction the face will be facing (if the viewside is not given, the default value is "plain side up").

Effects Of Assignment Statements

Parsing and executing an assignment statement creates an "assignment" difference, consisting of a symbol name, a type and a value.

Effects Of Simple Fold Statements

Parsing and executing a simple fold statement creates a "fold" difference. This is done by determining the fold line and the moving and folded faces, then finding the new locations of the vertices and faces and determining the new edges, faces and vertices as described in section 5.1. The labelling will be parsed before the "fold" difference is worked out. This will create a list of pairs of identifiers and edges which is passed to the part of the program which works out the "fold" difference.

Effects Of Unfold Statements

Unfolding a folded edge may be considered as a simple fold.

The effect of the "unfold all" statement is to create an "unfold all" difference. This is done by picking a face to be the fixed face (the one with id 0) and performing the algorithm described at the end of section 5.2. The values stored are new locations of all vertices, the direction the fixed face is facing and whether the direction the fixed face is facing is the desired "up" direction (given in the statement, "plain side up" by default).

Effects Of Fold Definitions

A fold definition statement creates a fold definition difference consisting of the definition of the fold. The definition consists of two lists of parameters, the second one being the optional parameters, and the unmodified text of the definition.

Effects Of Predefined Fold Statements

A predefined fold statement causes the arguments to be substituted for the parameters of the fold definition and the fold definition to be expanded. The expanded text is treated as a statement list and parsed, and so any locally defined fold definitions will be expanded recursively. These local fold definitions are removed from the list of fold definitions after the parsing has been done so that they have local scope.

Effects Of Include Statements

An include statement causes the stated file to be parsed with the syntax :

{ *FoldDefinition* | *IncludeStatement* }

which creates a list of differences.

Effects Of Expressions And Faces

The effects of expressions does not need any explanation in most cases. The result of parsing and executing an expression is a type and a value.

Whenever an identifier of type vertex is used in an expression, a pointer to the current text line is added to the reference list for that vertex.

Parsing a face gives a face id. The face id is the id of any of the faces which are within the conceptual face the user specified, that is, the region bounded by folded or raw edges which would be a single face if there were no creases crossing it.

6.4. Effects Of Applying Differences On The Data Structures

When a folding sequence being replayed is advanced one step or one statement, the differences are applied to the current state of the model. Differences are also applied after the user enters a new step. The effects of applying the various kinds of difference are given below.

Effects Of Applying Viewpoint Differences

A viewpoint difference corresponds to the "turn over" statement. The effect is to negate the z-coordinate of the current viewpoint.

Effects Of Applying Restart Differences

Application of a "restart" difference will have the following effects on a model :

Vertex table: all vertices except for the corners of the face are deleted. The ids of the vertices are re-assigned values from 0 to N-1 where N is the number of vertices remaining. The next available vertex id is changed to N.

Symbol table: all symbols of vertices other than those remaining in the vertex table are deleted.

Edge table: all edges other than those on the given face are deleted.

Face table: only the given face remains and its id is changed to 0. The next available face id is changed to 1.

Layering: there are no relationships between faces.

Viewpoint: if a different "side up" was given to the direction the face is already facing then the z-coordinate of the viewpoint is negated.

Effects Of Applying Assignment Differences

Application of an "assignment difference" causes the given symbol to be inserted into the symbol table with the given type and value.

Effects Of Applying Unfold All Differences

Application of an "unfold all" difference will have the following effects on a model :

Vertex table: all vertices are moved to their new location.

Symbol table: unchanged.

Edge table: unchanged.

Face table: the direction of all faces is made the "side up" given (which will be the "side up" which the fixed face was before the fold was made).

Layering: there are no relationships between faces.

Viewpoint: if the "side up" given was different to the "side up" desired then the z-coordinate of the viewpoint is negated.

Effects Of Applying Simple Fold Differences

Application of a simple fold difference is done in several stages :

1. All new vertices are created, each with the next available id (which is incremented each time). Locations of vertices and the pointers to the text line and place in the difference list where the vertex was created are as given. The name and id of the vertex is put into the symbol table and the pointer to the symbol table entry in the vertex is assigned. The vertex is not initially adjacent to any other vertices.

2. All moving vertices are moved to the given locations.

3. A copy of the old layering is created which will replace the old layering and which has room for the layering relationships of all of the faces which will be created.

4. All of the faces which have an edge on the fold line and move are folded. If the edge is currently unfolded it becomes folded with the given fold type, otherwise it becomes unfolded and retains its old fold type.

5. All of the faces which will get folds in them are split in two. The unmoving part of the face will retain the old face id and the moving part will get the next available face id (which is incremented). A new edge is created along the fold with the given fold type. The direction of the moving part of the face is opposite to the direction of the old face. For each endpoint of the new edge, if it is an old vertex then the other vertex is added to the list of adjacent vertices for that vertex, otherwise the edge is divided into two parts and the adjacent vertices of the vertex are the original endpoints of the edge and the other vertex. The original edge is stored in the vertex as the edge which the vertex was created on. The layering relationships of the moving part of the face are copied from the relationships of the original face.

6. The layering is completed as follows. Relationships between moving parts of split faces and moving parts of other split faces are copied from the original pair of faces. Relationships between faces which move and have an edge on the fold line and moving parts of split faces are copied from the relationship between the original unsplit face and the other face. All relationships between moving and unmoving faces (including split faces) are made equal to "unrelated". All split parts of faces found to be unrelated to other faces after the split are made "unrelated". Relationships between moving and unmoving faces which are known to be "above" or "below" are given the relationships. All layering relationships between moving faces are reversed.

Effects Of Applying Fold Definition Differences

A fold definition difference makes a fold definition come into scope, so the effect of a fold definition difference is to add the fold definition to the list of fold definitions.

Chapter 7. Implementation

7.1. Issues Involved In Implementation

One problem which must be resolved in the implementation of the program is how to display the flat model so that the vertices which could be involved in a fold are visible. The user must be able to see vertices in order to be able to select them with the mouse. If the model was displayed using the actual positions of all of the vertices then the layering of the faces would be invisible and it would be impossible to see vertices or faces which were between layers.

All of the vertices could not be made visible at the same time. It would be possible to give the model a false perspective so that it may be viewed from various viewpoints close to perpendicular to the paper by giving each vertex a parallel offset. In this way all of the vertices would be able to be seen. However, preventing apparent interpenetration of faces would be difficult. Giving vertices three dimensional coordinates is impossible in the general case, since some models have a layering such that there is no assignment of coordinates to the vertices which keeps all faces planar and has no interpenetration of any faces; an example is given below.

In origami books, vertices and faces at the same location are drawn in such a way that they can be distinguished. However, changing the locations of all vertices so that they are slightly apart from each other must be done in such a way that no faces interpenetrate. This is difficult to do at every location of the paper simultaneously. An alternative display method is to use the actual coordinates and not show the location of every vertex and face but just display the top faces of the model, perhaps drawing two lines instead of one to represent multiple edges. When a part of the paper is selected, it is "magnified" in a separate window, showing all actual vertices and faces at that location slightly separated from each other. The actual vertex may then be selected. This is simpler to do at a single point that at all points on the model simultaneously. A problem with this method is that when the folds are replayed it will not be clear which vertices are involved in the fold if there are several at the same location.

The user should also be able to change the x, y and z-coordinates of the current viewpoint. This is done most naturally using a mouse. The sign of the z-coordinate may not be changed, since this would change the meaning of the words "mountain" and "valley". Perhaps the model should also be able to be rotated.

The displayed model should show where the folds in the next command will be done, using the standard symbols for mountain and valley folds. In multiple folds, folds of type "reverse" and "unfold" need not be shown, as in traditional origami diagraming. Existing crease lines should be shown as well.

Another issue in the implementation is input. The input is primarily text, and so an editor is needed. At any time during the input of text the user may wish to select a point with the mouse and insert its name into the text, so the editor must be a part of the program. The user may wish to refer to a point which is not a vertex. This could either be done by making the user specify such points in terms of vertices, or the user could be allowed to select points which are not vertices, which would mean that the program must be able to create the appropriate expression to be inserted into the text. Creating expressions for points which are not on the paper may cause difficulties (the simplest general method is to specify the point as a relative distance along a line through a vertex and a point on an edge). Editing previous instructions should be illegal, because doing so may cause contradictions in the following instructions.

Representation of coordinates is also an issue. Using floating point values, locations of vertices cannot be represented exactly. Lines which come near vertices are considered to pass through them. Alternate angles in multiple folds which add up to an amount close to 180° are considered to be equal to 180°.

7.2. Implementation Achieved

A program to handle simple folds has been created, with the facility to view the model from different viewpoints using false perspective as described above and to perform every kind of simple fold. The parts of the syntax not implemented are multiple folds, tucks, fold definitions and restart statements, as well as the "centre" expression (the reason the last two were not implemented was of lack of time and not the difficulty of the problem).

The program allows a mixture of text and mouse input. Folding and other commands are entered in a text window and the model is displayed in another window. If a vertex is selected with the mouse, its name is inserted into the text at the current cursor position; if the vertex is unnamed then the user is prompted for a name, and the program recursively inserts the labelling commands for that vertex and any other vertices which must be named to name the vertex into the appropriate locations in the text, using automatically generated names if they are needed. There is a facility to find the labels of any or all of the vertices displayed on the screen.

The folding sequence may be replayed from the beginning one statement at a time; at any time the user may return to the start or skip to the end of the folding sequence. The crease lines of the folds in the next statement are displayed. There is a command to allow the types of the creases in the model to be displayed (with crease lines being displayed as mountain or valley fold lines rather than unbroken lines). Previous commands cannot be re-edited.

File functions "New", "Load", "Save" and "Save As" have been fully implemented. "New" creates a default header for the origami model including the command "uses 1:1 paper", which may be altered to any other kind of header.

The program is implemented in the C language (10343 lines) on a Sun sparc station using SunOS under XWindows. The complete folding sequence for a paper aeroplane is given as an example on the next three pages.

A separate program which folds crease patterns has also been created. Input consists of the faces, vertices and edges on a flat, unfolded piece of paper; output consists of the new locations of all vertices and the layering and direction of all faces after the edges are folded. Ambiguities in the layering are reported. Relationships between specific pairs of faces may be given to disambiguate the layering. If the crease pattern is impossible then the reason why is reported. An example of the input to the program may be found in section 4.4. The output of the program using this input as well as the "Pajarita" from section 5.2 is given below.

Chapter 8. Conclusion

A language for giving folding sequences for flat origami models has been designed which enables most flat origami models to be described. The language has been shown to be a superior method of recording general folding sequences compared to other existing methods in terms of clarity and ease of recording, as long as the instructions are entered and executed using the program. Issues involved in the creation of an origami language have been addressed. A subset of the language has been implemented.

Algorithms for simple folds and folding a model given its crease pattern have been discovered and implemented. Progress has been made towards algorithms for multiple folds and tucking.

Future Research

There are many unexplored areas relating to computers and origami. With regards to a language for describing origami, there is the potential for an origami description language in which variables, iteration and conditional statements are allowed, as described in section 3.8. The origami language described in this thesis could also be extended to three dimensions, or folding " in the air" could be permitted. There are possible variations in the language, such as some more natural means of referring to vertices (for example, in terms of directions and other parts of the model). Another obvious extension to the language is to allow cutting or multiple sheets of paper (kirigami and modular folding respectively).

There is still a lot of scope for research within the area of animation of origami involving complex folds. Multiple folds more complex than a reverse fold often have many possible ways of being animated, and it is difficult to determine which way will best reflect the behavior of physical paper.

Physical properties of paper have not been modeled extensively; properties such as paper thickness, how well the paper holds a crease, elasticity and others could be modeled. Curves in the paper also present a complex problem yet to be solved.

There are some applications of Artificial Intelligence in the area of computer origami. If a "repeat" statement was introduced to the language, common in traditional origami instructions, then the computer would have to generalise the previous operation and apply it in another context. Using similar methods new folds could be defined by example, rather than by giving explicit commands. Another application of AI is in determining folding sequences for complete origami models, perhaps improving traditional methods of folding.

There are many possible variations in the data structures used for representing origami. Use of infinite precision values to store angles and coordinates of points in order to prevent the "drift" effects of floating point numbers is one possibility.

Appendix A. Theorems

Preliminary Definitions

The following definitions are used in the theorems in this appendix :

A *raw edge* is an edge which is adjacent to only one face - that is, it is on the "outside".

A *face* is a region bounded by coincident raw or folded edges in which there are no other edges. There may be creases on a face.

An *interior vertex* is a vertex with no coincident raw edges.

Theorem 1

The line formed by a mountain or valley fold is the perpendicular bisector of any two points which are brought together in the fold.

The following proof appears in [8] and [5].

Proof

Let P and Q be two different points which are brought together in a mountain or valley fold, and let X be a point on the resulting fold line, as in the diagram below.

$\therefore$ PX = QX (since P and Q are at the same point)

$\therefore$ every point on the fold line is equidistant from P and Q, which is the locus of the perpendicular bisector of the line PQ.

□

Theorem 2

Let M be the number of mountain folds at an interior vertex and V be the number of valley folds. Then M-V = ±2.

This theorem was originally formulated by Jun Maekawa. The proof is taken from [7], and was originally conceived by Jan Siwanowicz.

Proof

Consider a model with an interior vertex. If the model is cut near the interior vertex after it is folded flat, a polygon will be formed with each angle equal to 0° or 360°, as in the diagram below.

The sum of the interior angles of any polygon equals 180(n - 2) where n is the number of interior angles. In this case, n = M + V, so we have

$\sum$ (all interior angles) = 180(M + V - 2)

If the interior vertex points "up" then each valley fold corresponds to an angle of 360° in the polygon and each mountain corresponds to an angle of 0°, so

$\sum$ (all interior angles) = 360 V + 0 M

$\therefore$ 180(M + V - 2) = 360 V + 0

$\therefore$ M + V - 2 = 2 V

$\therefore$ M - V = 2

If the interior vertex pointed "down" instead of "up" (which would be the case if V > M) then each valley fold would have corresponded to an angle of 0° and each mountain fold to an angle of 360°, and so M - V would equal -2.

$\therefore$ M - V = ±2

□

Theorem 3

The number of folded edges coincident with an interior vertex is even.

Proof

Let M be the number of mountain folds at the interior vertex and V be the number of valley folds. We know M - V = 2 or M - V = -2 (from theorem 2). If M - V = 2 then the total number of folds M + V = 2 V + 2, which is even. Similarly, if M - V = -2 then M + V = 2 V - 2, which is even.

□

Theorem 4

The sum of alternate angles about an interior vertex is 180°.

This theorem is also known as Kawasaki's Theorem, and appears in [7] and [10].

Proof

Consider an interior vertex with angles $a_1, a_2, a_3 \ldots a_n$, as in the diagram below. Let $f_1, f_2, f_3 \ldots f_n$ be the edges coincident with the vertex such that f_1 and f_2 are the edges which together form the angle a_1, f_2 and f_3 form the angle a_2, and so on. The number n must be even, since the number of edges is even (from theorem 3).

After the folds are made,

the the angle which f_2 forms with f_1 is a_1

the angle which f_3 forms with f_1 is $a_1 - a_2$

the angle which f_4 forms with f_1 is $a_1 - a_2 + a_3$

...

the angle which f_n forms with f_1 is $a_1 - a_2 + \ldots + a_{n-1}$

But the angle which f_n forms with f_1 must be a_n

$\therefore\ a_1 - a_2 + a_3 - \ldots + a_{n-1} = a_n$

$\therefore\ a_1 - a_2 + a_3 - \ldots + a_{n-1} - a_n = 0$

We also know that $a_1 + a_2 + a_3 + \ldots + a_n = 360$

$\therefore\ 2a_1 + 2a_3 + 2a_5 + \ldots + 2a_{n-1} = 360$

$\therefore\ a_1 + a_3 + a_5 + \ldots + a_{n-1} = 180$

$\therefore\ a_2 + a_4 + a_6 + \ldots + a_n = 180$

□

Theorem 5

If the initial piece of paper is convex then every face in a flat origami model folded from that piece of paper will be convex.

Proof

Consider any interior vertex v. Since the sum of the alternate angles about v is 180° (from theorem 4), no single angle about v will be greater than 180°.

Consider any non-interior vertex v. If the initial piece of paper is convex then the initial (unfolded) angle at v cannot be greater than 180°. When a fold passes through v, the two angles formed will be less than 180°.

So the angles about every vertex are no greater than 180°, and so every face of the model is convex.

□

Appendix B. Origami Language Syntax

Syntax Conventions

In the syntax given below, the following conventions are used :

::= means "is defined as"

{ s1 s2 ... } indicates that "s1 s2 ..." occurs 0 or more times.

[s1 s2 ...] indicates that "s1 s2 ..." occurs 0 or 1 times.

| means "or"

(s1 s2 ...) groups "s1 s2 ..." together.

A quoted "(" or ")" is to be taken literally.

(=a/b/c...) indicates that the type of previous object must be one of (*a* or *b* or *c* or ...)

<type=...> indicates the type of the object defined.

Words beginning with a capital letter have a definition.

All other words and symbols are to be taken literally.

There may be zero or more white space characters between tokens.

Model ::= *Header StepList*

Header ::= *origami Ident UsesPart*

UsesPart ::= *uses* (*Expr* : *Expr paper* [*Viewside*]) | ([*step Number of*] "*String*")

StepList ::= [*StatementList*] *step Number StatementList*

StatementList ::= *Statement* { *Statement* }

Statement ::= *SimpleFold* | *MultipleFold* | *UnfoldStatement* | *ViewStatement* | *RestartStatement*
  | *AssignStatement* | *TuckStatement* | *IncludeStatement* | *FoldDefinition* | *PredefinedFold*

ViewStatement ::= *turn over*

Viewside ::= (*plain* | *coloured*) *side* (*up* | *down*)

RestartStatement ::= *restart with Face* [*Viewside*]

AssignStatement ::= *Ident* = *Expr(=real/angle/distance)*

TuckStatement ::= (*tuck Face under Face*) | (*untuck Face*)

Expr ::= *Term* { (+ | −) *Term(=real/angle/distance)* }
  [*degrees* | (*along Expr(=line/edge)*)]

Term ::= *Factor* { (* | /) *Factor(=real/angle/distance)* }

Factor ::= Number <type=real>
| Ident(=real/angle/distance/vertex) <type=same as Ident>
| Ident(=vertex) – Ident(=vertex) <type=edge>
| – Factor(=real/angle/distance) <type=same as Factor>
| distance "(" Expr(=point/vertex) , Expr(=point/vertex) ")" <type=distance>
| angle "(" Expr(=point/vertex) , Expr(=point/vertex) , Expr(=point/vertex) ")"
<type=angle>
| sqrt "(" Expr(=real) ")" <type=real>
| centre "(" Face ")" <type=point>
| intersection "(" Expr(=line/edge) , Expr(=line/edge) ")" <type=point>
| line "(" Expr(=point/vertex) , Expr(=point/vertex) ")" <type=line>
Face ::= face (Ident(=vertex) { , Ident(=vertex) })
SimpleFold ::= [valley | mountain] fold FoldType
FoldType ::= (Ident(=vertex) to Expr(=point/vertex))
| (Expr(=edge) to Expr(=line/edge))
| (Ident(=vertex) to Expr(=line/edge) through Expr(=point/vertex))
| (Expr(=edge) to Expr(=point/vertex) through Expr(=point/vertex))
| (along Expr(=line/edge) moving Ident(=vertex))
| (at Expr(=angle) to Expr(=line/edge) through Expr(=point) moving Ident(=vertex))
[and return] [Labelling]
UnfoldStatement ::= (unfold Expr(=edge) moving Ident(=vertex) [Labelling])
| (unfold all [Viewside])
MultipleFold ::= multifold [EdgeList] { , (valley | mountain | unfold | reverse)
(EdgeList | (between Expr(=edge) and Expr(=edge))) } holding Face
[MultifoldLabelling] [tucking Face under Face [{ , Face under Face }]]
EdgeList ::= Expr(=edge) { , Expr(=edge) }
Labelling ::= creating Ident on UnlabelledEdge { , Ident on UnlabelledEdge }
MultifoldLabelling ::= creating Ident [: Number] on UnlabelledEdge
{ , Ident [: Number] on UnlabelledEdge }
UnlabelledEdge ::= Ident – Ident
IncludeStatement ::= include "String"
FoldDefinition ::= DefinitionHeader StatementList end
DefinitionHeader ::= define Ident fold "(" Ident { , Ident } [; Ident { , Ident }] ")"
PredefinedFold ::= Ident fold "(" Expr(=vertex/real/angle/distance) ,
{ , Expr(=vertex/real/angle/distance) } { , Ident = Ident(=vertex) } ")"

Notes On The Syntax

Types are as follows : real, angle, distance, vertex, edge, point, line.

Identifiers consist of an alphabetic character followed by any number of alphanumeric characters or underscores. An identifier cannot be a keyword.

Strings may include any character except for newline. "*String*" represents a string enclosed in double quotes.

Numbers consist of 1 or more digits followed by an optional decimal point and 0 or more digits. If there are digits after the decimal point then it is legal to have no digits before the decimal point.

Comments start with "#" and continue until the end of the line.

Some semantic information has been left out in the above syntax :

AssignStatement: the identifier which is being assigned to must not already have a value.

Face: the vertices given must have exactly one adjoining face in common.

Expr: can only add Terms of the same type; return type is the same as the term type. The word "degrees" converts a real to an angle. The word "along" followed by a line or edge converts a real or distance to a point; if it is a real, it represents a distance relative to the length of the line or edge (for example, 1/2 along a-b gives the midpoint of a-b).

Term: the result types of multiply and divide operations are as follows :

Real * any type = same type; Any type * real = same type.

Any type / same type = real; Any type / real = same type.

The legal types for the operands of "*" and "/" are real, angle and distance.

Factor: the argument of the "sqrt" function cannot be negative.

FoldType: the keyword "moving" may be omitted along with the following vertex if a crease is being done (ie. if the words "and return" appear).

MultipleFold: at least one edge must be specified in the fold.

Labelling and Multifold Labelling: the Idents must not already have a value.

Include: the contents of an included file of fold definitions has the syntax :

{ *FoldDefinition* | *IncludeStatement* }

DefinitionHeader: the identifiers after the ";" represent optional arguments which are labels, as described in the section on defining new folds.

References

1. Agui, Takeda and Nakajima, Animating Planar Folds By Computer. *Computer Vision, Graphics and Image Processing* **24**, 244-254 (1983).

2. Agui, Takeda and Nakajima, *Conditional Key Frame Animation*. Tech. Rep. 1-2 (1981), IPSJ(Japan).

3. Engel, Peter, *Folding the Universe : Paper Folding From Angelfish to Zen*. Dover, ?.

4. Fushimi, K. and Fushimi, M., *Origami Geometry*. Nippon Hyoronsha Publishing Company, 1979.

5. Gardner, Martin, Mathematical Games. *Scientific American* **201**, 138-143 (1959).

6. Gray, Alice, O.I.L. : An Origami Instruction Language. *The Origamian* **13**, (1975). (Published by The British Origami Society)

7. Hull, Thomas, *On the Mathematics of Flat Origamis*. University of Rhode Island, 1994. Available via anon. ftp from rugcis.rug.nl in /origami/articles.

8. Johnson, Donovan A., *Paper Folding Geometry*. J. Weston Walch, 1976.

9. Kasahara, Kunihiko, *Origami omnibus : paper-folding for everybody*. Japan publications, Inc., 1988.

10. Kasahara, Kunihiko, *Origami for the Connoisseur*. Japan publications, Inc., 1992 (5th printing).

11. Kasahara, Kunihiko, On the Relation Between Mountain-creases and Valley-creases of a flat origami. *Origami Science and Technology*, 1989, 229-237.

12. Kenneway, Eric, *Complete Origami*. Fontana, 1987.

13. Lang, Robert J., *The Complete Book Of Origami*. Dover Publications, Inc., 1988.

14. Lang, Robert J., Albert Joins The Fold. *New Scientist* **23**, 38-47 (1989).

15. Mander, Dippel and Gossage, *The Great International Paper Aeroplane Book*. Simon & Shuster, Inc., 1967.

16. Miyazaki, Yasuda, Yokoi and Toriwaki, An Interactive Simulation System of ORIGAMI Based on Virtual Space Manipulation. *IEEE International Workshop on Robot and Human Communication*, 1992, 210-215.

17. Miyazaki, Yasuda, Yokoi and Toriwaki, Interactive Manipulation of ORIGAMI in 3D Virtual Space. *Transactions of The Information Processing Society of Japan* **34**, 1994-2001 (1993). (in Japanese)

18. Murakami, Yukio, Origami Geometry. *IEEE International Conference on Systems, Man and Cybernetics*, 1987, 1107-1109.

19. Randlett, Samuel, *The Art of Origami*. Faber and Faber, London, 1961.

20. Uchida and Itoh, Knowledge Representation of Origami and Its Implementation. *Transactions Of The Information Processing Society of Japan* **32**, 1566-1573 (1991). (in Japanese)