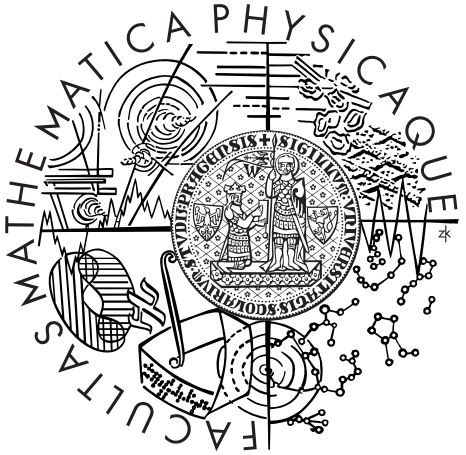
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Martin Pecka

Origami diagram creator

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Martin Petříček

Study programme: Informatika

Specialization: Obecná informatika

Prague 2011

I would like to thank my supervisor, Mgr. Martin Petříček, for valuable and helpful advices, and for smooth supervision.

Také bych chtěl poděkovat celé své rodině za podporu a příjemné prostředí pro tvorbu.

And I feel the obligation to thank all MFF teachers who taught me very much and broadened my mind in a great way.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date .....

Název práce: Origami diagram creator

Autor: Martin Pecka

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: Mgr. Martin Petříček, Kabinet software a výuky informatiky

Abstrakt: Program Origamist si klade za cíl pomoci s tvorbou návodů na skládání origami modelů. V současné době jsou nejběžnějšími metodami pro tvorbu těchto návodů buď ruční kreslení všech kroků v obrázkovém editoru, nebo nafocení jednotlivých kroků a manuální sestavení návodu z nich (opět v obrázkovém editoru). Origamist na toto pole přináší novou alternativu. Autor tak dostává možnost přenést posloupnost ohybů papíru, z nichž se návod skládá, do programu Origamist, jenž z nich dokáže vygenerovat několik druhů výstupu — v origamistických kruzích nejrozšířenější PDF návod, ale i návod jako obrázek (PNG, SVG), nebo dokonce jako animaci procesu skládání. Přidanou hodnotou pak je snadná možnost přeložit popisky kroků do více jazyků.

Klíčová slova: Origami, Java3D, skládání papíru, návod

Title: Origami diagram creator

Author: Martin Pecka

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Martin Petříček, Department of Software and Computer Science Education

Abstract: The main target of the Origamist application is to aid with creating origami diagram manuals. Recently, the most common methods for creating those manuals are drawing the steps in an image editor, or taking photographs of the folded steps and composing them together (again in an image editor). What Origamist brings is a new alternative to these methods. The manual's author gets the possibility to transfer the sequence of paper folds the manual consists of to the Origamist application, which is able to export several types of output - the most favourite (among origami folders) PDF manuals, but also image manuals (as PNG or SVG), and even as an animation of the folding process. There is also the added value to translate the descriptions of the steps to several languages.

Keywords: Origami, Java3D, paper folding, diagram

Contents

Introduction

Everyone who has ever tried to fold an origami model knows how important it is to have a *good manual* describing the steps of the folding process. A good manual consists of lots of images (generally one image per step) and their descriptions.

Why are the manuals needed? It is too hard (if not impossible) for most people to guess the folding process by only seeing the result shape. And this is even harder if you only see the result as a 2-dimensional image on a screen or paper. Thus, several methods to aid people with the folding process were invented.

The most practical help method is learning from someone who knows how to fold the desired model. This method has two disadvantages - a man can forget what he has learnt, and, in most cases, there is simply nobody who knows how to fold the desired model (except in origami communities). Therefore *paper manuals* were invented. They are (conceptually) eternal and are relatively easy to obtain (in books or on the Web). They have another disadvantage - if some steps are unclear in the manual (which is not a rare case), there is no other help. The last (relatively new) means to learn the folding process are *video tutorials*.

In the paper manuals (or bitmap image manuals, we will call both of them 'paper' manuals) there are some established graphical marks that indicate the operations to be done with the paper in the step. We will discuss these marks in higher detail further in the text. The marks cover the most of operations one would like to do with the paper, but their meanings aren't fixed and unambiguous, which can lead to lack of clarity of the manual.

'Paper' and video manuals share one more disadvantage, too. They aren't *simply editable* by other people. The 'paper' manuals are either printed or distributed as PDF or image files, and these aren't simply editable (PDF editors exist, but aren't widely used; editing an image manual involves some non-basic knowledge of computer usage). Video editing is even more difficult. So, ways to edit these manuals exists, but none of them is straightforward.

Origamist brings a new alternative to those types of manuals. It presents the concept of 'live' manual. Each folding step is represented as a 3-dimensional model, which the user can view from different viewing angles and zoom levels. Furthermore, everyone can simply edit the model in the Origamist Editor. It doesn't matter if the user just wants to add a translation of the descriptions of the steps, edit existing step descriptions or if he wants to add some more steps, all of these activities can be done straightforward in the Editor.

Also, all of the previously mentioned types of origami manuals (except personal assistance) can be exported from the Origamist application. Only the exported animation has no sound track, which is an important part of video tutorials (but it is possible to add this functionality, also the data model can be simply modified to store this type of descriptions).

Chapter 1 will guide the reader through the history and basics of origami. Chapter 2 presents some important mathematical and computational results in the field of origami folding. Chapter 3 takes a look at Origamist's technical background. Chapter 4 discusses the algorithms and data structures used in Origamist. Chapter 5 contains a user's manual for both Origamist Viewer and

Origamist Editor. And the last chapter serves as a programmer's guide through interesting points of Origamist.

1. What is origami?

Origami is the name of an ancient Asian art of folding various figures and shapes from a sheet of paper.

"Whether it is called 'zhe zhi,' as it is by Mandarin-speaking Chinese, or 'chip chee,' as Chinese who use the Cantonese dialect call it, or by the Japanese name 'origami,' it is generally agreed that the art of paperfolding originated in China perhaps before the 6th century." [1, p. 123]

The traditional Japan origami focuses mainly on animal figures and flowers. You could know the most famous origami animal - the crane. It is considered as a sample of the traditional technique.

In recent times, hundreds of other origami models can be found, including boxes, decorations and ornaments, envelopes, abstract things and much more. [2]

1.1 The rules of traditional origami

There aren't much constraints in the traditional origami. Everyone can fold whatever his fantasy invents, but to call the model a traditional origami, there are these rules: [3]

- To begin with a single square sheet of paper.
- Not to tear or cut the sheet of paper.
- Not to use glue.

And that's it. No more constraints on what can be done. This is also the set of rules Origamist tries to hold (in fact, it allows non-square papers).

1.2 Other kinds of origami

Besides the traditional origami, there are several other types, differing in what is allowed or disallowed. Here is a short list of some other techniques:

- *Modular origami*, which allows to use and combine multiple sheets of paper. [4]
- *Action origami*, which creates figures whose parts are moveable, so they can be used as toys. [5]
- *Pureland origami*, which only allows one fold to be done at once, and thus disallows folds like reverse folds or rabbit folds. This type of origami is suitable for beginners, children and disabled people. [6]
- *Kirigami*, which allows cutting the paper and using glue. Kirigami is mostly used to make decorations. [7].
- *Technical origami*, which develops the models based on computer-generated crease patterns (more on crease patterns can be found in 1.5). [8].

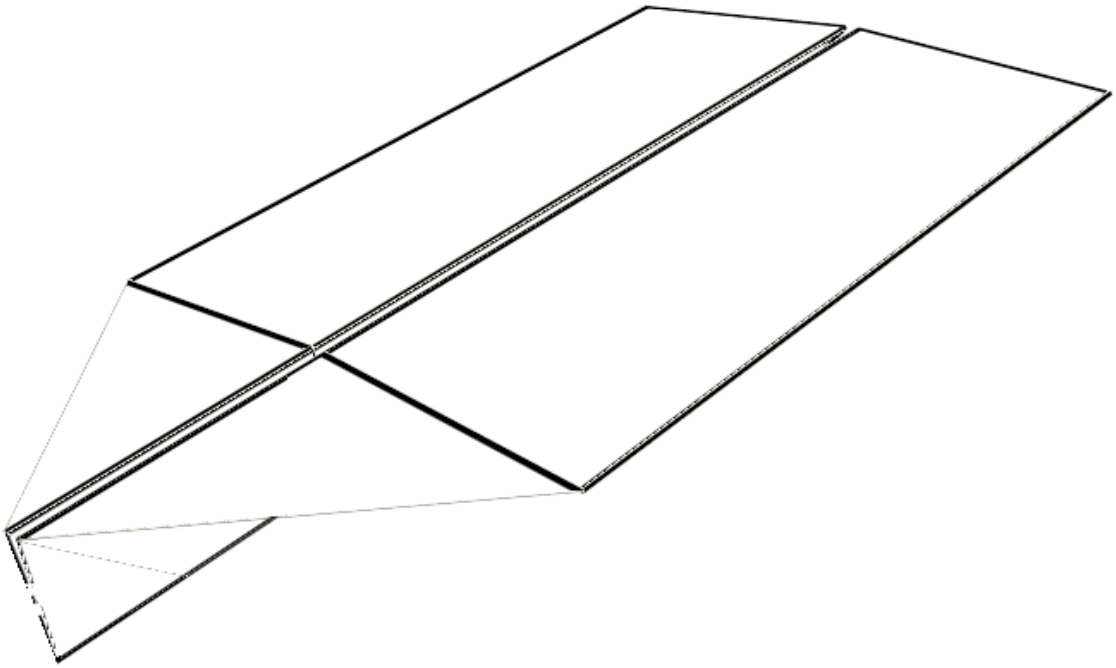
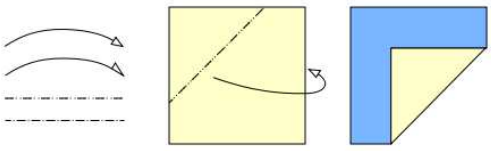


Figure 1.1: Origami model of a paper plane

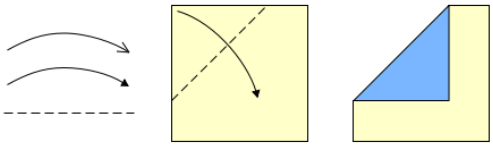
- *Mathematical origami*, which is just abstract and is used for some mathematical and algorithmic proofs.
- *Rigid origami*, which is a subset of mathematical origami and tries to find answers to the question "If we replaced paper with sheet metal and had hinges in place of the crease lines, could we still fold the model?" [8].

1.3 Basic folds

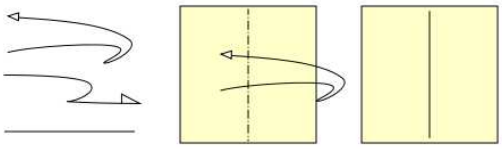
In this section, some basic fold types will be described, as will be the common marks for them. Sample images are taken from [9], the descriptions of the steps are inspired by [2].



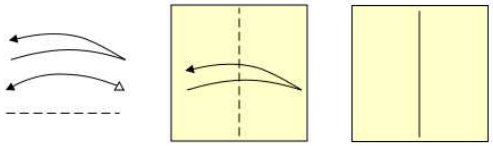
Mountain fold. This is a simple fold which bends the paper in the direction 'from the viewpoint'. An arbitrary angle can be specified for this type of fold.



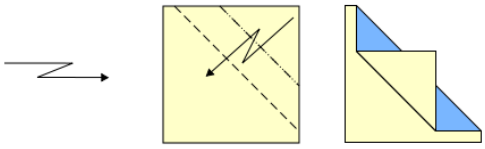
Valley fold. This is a simple fold which bends the paper in the direction 'towards the viewpoint'. An arbitrary angle can be specified for this type of fold.



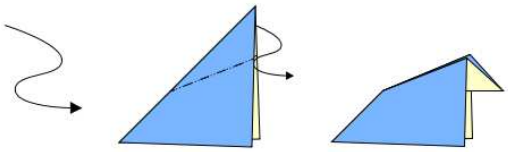
Mountain fold and unfold. This fold just makes a mountain crease on the paper. After doing it, the paper has the same geometry as before, but the crease has required moving the paper. This is mainly used for creating reference creases.



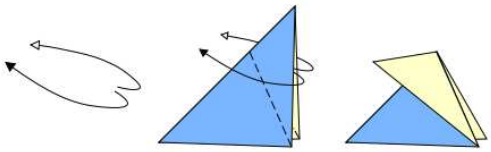
Valley fold and unfold. This fold just makes a valley crease on the paper. After doing it, the paper has the same geometry as before, but the crease has required moving the paper. This is mainly used for creating reference creases.



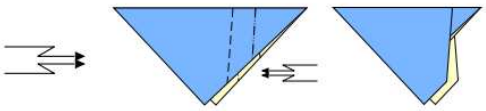
Thunderbolt fold. A double fold, one of the folds is mountain, the other is valley. An arbitrary angle can be specified for both folds (a different one for each of them).



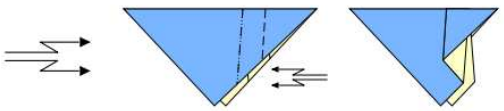
Inside reverse fold. Tuck a tip of a flap[1] inside. Uses two mountain folds for side creases, and a valley fold for the inner centre line. For every paper configuration, there exists only one angle for the created folds which guarantees to preserve the paper properties.



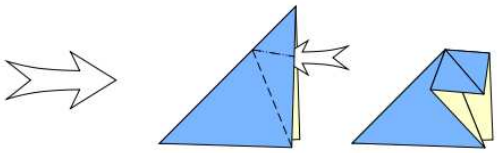
Outside reverse fold. Wrap a tip of a flap[1] around its outer side. Uses two valley folds for side creases, and a mountain fold for the outer centre line. For every paper configuration, there exists only one angle for the created folds which guarantees to preserve the paper properties.



Inside crimp fold. This is a double fold where both folds are inside reverse folds.

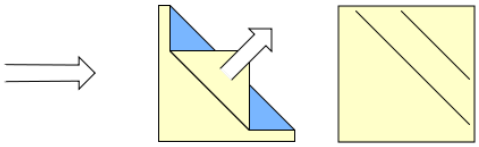


Outside crimp fold. This is a double fold where both folds are outside reverse folds.

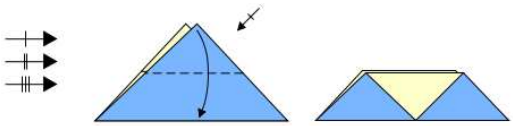


Open fold. Open or squash a flap[1] of paper. Origamist doesn't support this operation.

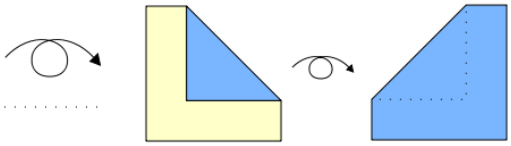
[1] A triangle standing out of the paper, which consists of at least two paper layers.



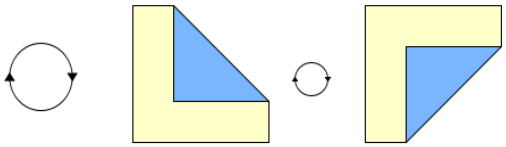
Pull fold. Unfold some previously created folds. Origamist has only partial support for this operation. Only one fold can be unfolded at a time, so eg. atomically unfolding a thunderbolt fold isn't possible.



Repeat. Repeat some previously done operations. This operation hides (for clearness) some steps and substitutes them with the repetition mark.



Turn over. Turn the paper to the other side. Origamist only supports turning around the horizontal axis of the current view, and a single turning angle - 180 degrees.



Rotate. Rotate the model of an arbitrary angle around the current view's direction axis (normal of the screen).

By using these operations, it is possible to create most of the basic and moderate origami models. Advanced models, however, often require the use of more advanced techniques. Some of them are described in [2].

1.4 More on operations' marks

As can be seen from the above images, some folds are drawn as solid lines, other are drawn as dashed lines and so on. Here are the rules for deciding how to visualise a fold line. The edges of paper are always drawn solid, a little thicker than other lines. Visible creases made in the previous steps (except the current

one) are drawn by solid lines (or they can be completely omitted). Invisible[2] folds are drawn by a dotted line (.....). And finally, the creases made in the current step of the manual are drawn as dashed lines (- - - - -) for valley folds and dash-dot lines for mountain folds (-..-..-). Note that the mountain/valley assignment is relative to the side of the paper we look at — one fold always creates both valley and mountain folds on the opposite sides of paper. The fold with non-convex angle around it is mountain, and the fold with angle less than 180° is valley.

1.5 Crease patterns

Crease patterns, although they are a very old concept, experience a boom in the latest years as an alternative to classical origami manuals. What is a crease pattern? It is an unfolded origami model, which retains the creases created during the folding process. So, basically, a crease pattern is a set of lines on a straight sheet of paper. Depending on the purpose, the directions of the folds (whether they are mountain or valley) may or may not be specified.

Why are they so popular? This may be due to two wholly different reasons. The first one is that folding a model (with known target shape) from a crease pattern is a challenge, and has no direct instructions. The second reason is that crease patterns are very important in mathematics and computational origami, which is shown in the next chapter.

[2]Folds hidden under other parts of paper. The visualisation using dotted lines is sometimes called X-Ray folds.

2. Origami, computers and mathematics

This chapter presents some recent origami-related results in the field of mathematics or computer science, and also presents some existing origami software. Although origami is mainly art, most of these results are very interesting, and some of them have even practical use.

2.1 Computational origami

"Most results in computational origami fit into at least one of three categories: universality results, efficient decision algorithms, and computational intractability results.

A *universality result* shows that, subject to a certain model of folding, everything is possible. For example, any tree-shaped origami base, any polygonal silhouette, and any polyhedral surface can be folded out of a large-enough piece of paper. Universality results often come with efficient algorithms for finding the foldings; pure existence results are rare.

When universality results are impossible (some objects cannot be folded), the next-best result is an *efficient decision algorithm* to determine whether a given object is foldable. Here 'efficient' normally means 'polynomial time'. For example, there is a polynomial-time algorithm to decide whether a 'map' (grid of creases marked mountain and valley) can be folded by a sequence of simple folds.

Not all paper-folding problems have efficient algorithms, and this can be proved by a *computational intractability result*. For example, it is NP-hard to tell whether a given crease pattern folds into any flat origami, even when folds are restricted to simple folds. These results mean that there are no polynomial-time algorithms for these problems, unless some of the hardest computational problems can also be solved in polynomial time, which is generally deemed unlikely." [10]

2.1.1 Other computational tasks

As [10] says, there is another division of computational problems. They are *origami design* and *origami foldability* problems. *Origami design* examines the possibilities (finding the needed creases) of folding arbitrary shapes with specified properties. On the other side, *origami foldability* tries to answer if the given crease pattern is foldable into any shape.

One complete straight cut problem solves the task whose input is a polygon drawn on paper, and whose output is a crease pattern which can be folded in the way that all the polygon's edges overlap on a single line. Why one straight cut? If you cut the paper along the overlap line, you will get the polygon cut out of the paper. [10] presents results of two different approaches, both of them proving that such crease pattern can be found for any polygon.

Silhouettes and polyhedra problem takes a 2- or 3-dimensional silhouette of the desired model as its input and returns a folding of the paper resulting in the

input shape. Again, according to [10], such result always exists.

Flat foldability tries to find out if a given crease pattern (without fold directions specified) is foldable to a flat model using a consistent fold direction assignment. [11] shows a linear-time algorithm that finds such direction assignment, or tells that the given pattern is not flat foldable.

2.1.2 Existing origami software

There is a number of computer applications related to origami. Here is a list of some of them:

- *TreeMaker*[1] by Robert Lang is a tool for designing crease patterns for given sketched shapes. The shape is basically sketched as a tree (in the algorithmic meaning of the word), thus TreeMaker. The program uses the disc packing method, described in [12].
- *Origami simulation*[2] by Robert Lang is a simple GUI tool for designing pureland origami. Since 1992 the development has been stopped, and it only works on PowerPC Macs.
- *ORIGAMI Playing Simulator in the Virtual Space* by Shin-ya Miyazaki, Takami Yasuda, Shigeki Yokoi and Jun-ichiro Toriwaki. Another interactive GUI tool for origami model folding, was presented in [13]. Although this seems to be the most capable GUI tool for origami, I couldn't test it, because the given paper doesn't provide a link to the application.
- *Doodle*[3] by Jérôme Gout, Xavier Fouchet, Vincent Osele, and volunteers. Doodle is a tool for generating origami diagrams from the given ASCII text instructions. This program produces nice diagrams, but needs a lot of additional information in the input files (eg. it is often needed to rotate some paper parts by an explicit command, because the fold commands just create the associated fold lines).
- *ORIPA*[4] by Jun Mitani. ORIPA is interactive GUI tool for designing crease patterns. It also provides an estimated preview of the folded model.
- *Computational Origami System Eos* by Tetsuo Ida, Hidekazu Takahashi, Mircea Marin, Asem Kasem and Fadoua Ghourabi. This software introduced in [14] is a non-interactive tool for folding origami models and automatic proving of some conclusions about the constructed models. EOS is written in Mathematica.

2.2 Origami in other sciences

Mathematics and computer science aren't the only disciplines origami brings benefits to. Here is a list of other uses of origami theory:

[1] `http://www.langorigami.com/science/treemaker/treemaker5.php4`

[2] `http://www.langorigami.com/science/origamisim/origamisim.php4`

[3] `http://doodle.sourceforge.net/about.html`

[4] `http://mitani.cs.tsukuba.ac.jp/pukiwiki-oripa/index.php?ORIPA%3B%20Origami%20Pattern%20Editor`

- *Stents* in medicine. Stents are small tubes used to reinforce or broaden clogged veins and arteries. Since they are transported to their destination through veins and arteries, they need to be in a folded state, then they can be transferred to the right place, and unfolded. A waterbomb base[5] is mainly used for stents. [15]

- *Eyeglass space telescope* needed the help of origami theory master Robert Lang[6] to invent a method of folding a large telescope into a space satellite. [16]

- *Space project SFU* of Japanese space organization JAXA used Robert Lang's origami experiences to efficiently fold and unfold a solar panel array in the open space. The fold they used is called Miura map fold[7].[17]

- Origami design techniques were used by a German company to simulate effective *packing of airbags* into car steering wheels. [18]

- Origami techniques helped with the development of *3D solar panels*, which increase the productivity of solar panels and have the advantage of not having any movable parts that can get broken. [19].

[5] One of the standard origami bases, which are just a series of steps shared by many models in the beginning.

[6] `http://www.langorigami.com/science/eyeglass/eyeglass.php4`

[7] `http://en.wikipedia.org/wiki/Miura_map_fold`

3. Origamist basic facts

Here I will present the basic structure of Origamist and the technologies used.

3.1 Basic structure of Origamist

3.1.1 Two parts of the application

Origamist isn't just a single application. It consists of two standalone parts.

Origamist Editor provides the tools needed to create new origami diagrams and manuals. So various folds can be added to existing models, or a brand new model can be created. Also the metadata of the model can be edited.

Origamist Viewer is a viewer application that can only display formerly created diagrams. Its main target is to be used as a webpage applet, so that the visitors of an origami site can comfortably browse the models the site offers. Although the viewer isn't primarily meant to create the manuals, it has the same export possibilities as the editor, so every user of any part of Origamist is able to create the manuals in various export formats.

3.1.2 Programming language and deployment technologies

Origamist has been written mainly in Java[1]. There were several reasons for this decision. Firstly, Java is a multiplatform language ([20]), which is a basic requirement due to the intended viewer usage. Secondly, Java applications can be integrated directly into a webpage (using the Java Web Start or Java Plugin), which rapidly simplifies access to the application. The third most important reason was that no platform-specific code has to be written (at least not in Origamist[2]), which greatly simplifies the programming work and allows to concentrate on the core business.

Java provides multiple solutions to application deployment, and Origamist uses these three (or four) of them:

- The first one is a standard Java application packed into a single JAR archive file[3]. So, anyone can run the application by moving to the folder containing the JAR and typing:

  ```
  java -jar OrigamiEditor.jar
  ```

- Another possibility is to include the application into a webpage as Java applet using the standard applet code. See listing 6.1 for a sample webpage.

- A similar one is to include the application into a webpage as Java applet using the *Next-Generation Java Plugin Technology* introduced into Java in version 6u10 ([21]). See listing 6.2 for a sample webpage.

[1] Several other languages have been used, such as XSLT and XML, but these are only scripting and tagging languages not providing the core functionality.

[2] Otherwise, Java allows to use platform-specific code, but doing so isn't a common practice in Java.

[3] Application libraries are standalone files, too.

- The last possibility is to launch and install the application through the *JNLP protocol*[4]. This protocol allows Java applications to be downloaded and installed on the target machine as standalone applications, so the end user can eg. make shortcuts to these programs and utilise them offline. JNLP also provides a way of automatic updates of the downloaded software. See listing 6.3 for a sample JNLP file that launches Origamist Viewer.

3.1.3 Data files

Origamist saves all its data in XML text files (except user preferences which are stored in the system registry). There are two types of files Origamist recognises.

Diagram files

Diagrams are XML files with the *.xml* file extension having their root element in a model-specific namespace. For technical details on this, consult the programmer's manual, 6.8.

The XML file contains all information needed to render the model and export the manuals, and also some metadata, like the model's name, author's name, description of the model, paper formats, licensing information and so on.

These files follow the convention of not storing any 3-dimensional data[5], so all folds are defined only by their position on the crease pattern.

Listing files

Origamist recognises one special type of XML files - those having their name *listing.xml* and having their root element in the listing-specific namespace. For technical details on this, consult the programmer's manual, 6.8.

Listing files are used by the viewer to define and organise whole sets of models. The listing files have a tree structure, where every model can be attached to a named category. Some excerpts of the models' metadata are also saved in the listing files, so that it isn't required to download the whole model to see its name or thumbnail.

Origamist provides no complete support for creating these files. Here are two main reasons. These files are supposed to be automatically generated on a server providing the models (this is the expected use case). Moreover, a partial support for manual creation of these files exists. If the user loads a whole directory structure into the viewer, it automatically creates categories for matching subdirectories using their names. The listing can then be saved from Origamist Viewer, however it doesn't support extended features like fully localised category names(these can be added manually to the exported listing file).

3.2 Interesting technologies used in the project

The most interesting technologies Origamist uses are listed here, for a full list of technologies and libraries, refer to the programmer's manual, 6.3.

[4] `http://www.oracle.com/technetwork/java/javase/index-142562.html`

[5] More on this topic in 4.8

3.2.1 Git

Git is a distributed versioning system which I have used for recording the work progress. It is easy to use and does its work well. Sometimes, the ability to amend last commit[6] is useful and helps keeping the repository clean. Even though nobody else used the repository and I have been the only commiter, the repository gave me bigger freedom in what I do with the code. The whole Git repository is on the attached CD.

3.2.2 JAXB

Java And XML Bindings[7] is a library for mapping XML files to Java objects and vice versa. It not only provides the mapping, but, since the XML files' schemata and the Java classes carry redundant information, offers to generate either the schemata or the Java classes.

I have chosen to write schemata for the data files (they are in XSD[8] format), and JAXB (using its tool XJC) generates the Java classes automatically. The generated classes are plain Java objects annotated by some special annotations, and they follow the Bean pattern[9]. Advantages of this approach are significant if there is the need to add some fields to the data files[10], just edit the XML schemata and the Java classes will be updated to be consistent with the schemata without any additional effort.

Among other advantages I would like to stress the simplicity of the Java $\leftrightarrow$ XML conversion (JAXB has the terms 'marshalling' for Java $\rightarrow$ XML, and 'unmarshalling' for XML $\rightarrow$ Java). It basically consists of just a few lines of code[11], but allows a great deal of customisation if it is needed.

Nevertheless, JAXB has one disadvantage. Since the classes are auto-generated, there is no chance of adding further functionality directly to those files, so they always remain plain 'data envelopes'. Although this may seem to be a big disadvantage, there is a near elegant solution to this problem. See 6.10.2 for detailed information.

3.2.3 Java3D

Java3D[12] is a powerful Java library for displaying and creating 3D graphics. It provides two basic modes of work — a 'direct' mode (called 'immediate mode' in Java3D) which makes working with Java3D much like working with OpenGL; the other mode is more object-oriented (it is called 'retained mode' in Java3D) and provides an easy-to-use interface for composing the 3D scene, which means that most of the computations are hidden from the programmer. Java3D integrates

[6] Additionally change the files or comment of the last commit.

[7] `http://www.oracle.com/technetwork/articles/javase/index-140168.html`

[8] XML Schema Definition, which is a promising successor of the old DTDs

[9] `http://download.oracle.com/javase/tutorial/javabeans/whatis/index.html`

[10] Which is often during the initial development, but not so frequent in further 'life' of the application

[11] Refer to ∼.services.JAXBListingHandler and ∼.services.JAXBOrigamiHandler

[12] `http://java3d.java.net/`

nicely with the Swing[13] GUI [14] library.

Since 3D graphics is basically a very low-level work, Java3D needs some native libraries to run. This effectively narrows the list of platforms Origamist will run on to the list of Java3D supported platforms. Fortunately, a Java3D implementation is available for most of the Java supported platforms [22]. As native libraries cannot be distributed in the standard way as JAR libraries are, they require special handling. Refer to the programmer's manual, 6.10.1, for more on this topic.

3.2.4 Forms

JGoodies Forms[15] is a Swing layout manager. Swing provides some layout managers in the standard JRE[16] distribution, but they either aren't much flexible, or are overly complex. JGoodies Forms allows the programmer to define a grid[17] on the current window or component, and put other components into this grid.

3.2.5 Batik

Batik[18] provides a simple way of generating SVG (Scalable Vector Graphics) and PDF files in Java. It has other SVG-related capabilities, too, but they are unimportant for this project.

3.2.6 Ant

Apache Ant[19] is a Java build tool. Its design is very robust, it allows to trigger innumerable different tasks during the build, so that everything needed to build and deploy a complex Java application is to run command

```
ant
```

in the command line from the application's base directory. Ant even supports plugins (and this project uses some of them).

[13] `http://download.oracle.com/javase/tutorial/uiswing/start/about.html`

[14] Graphical User Interface

[15] http://www.jgoodies.com/freeware/forms/

[16] Java Runtime Environment, the minimal installation of Java needed to run most Java applications.

[17] The use of grids is very common in UI design, but JRE doesn't provide any layout manager capable of working with more complex grids.

[18] `http://xmlgraphics.apache.org/batik/`

[19] http://ant.apache.org/

4. Origamist's algorithms & data structures

This chapter covers the most important computational algorithms, data structures and programming approaches the program uses.

4.1 Representation of the origami model

4.1.1 Triangles and layers

The model is processed simultaneously as a 3D model and 2D crease pattern. Both these models are represented by a set of triangles. Every 2D triangle corresponds to exactly one 3D triangle, and thus we will call the 2D triangle as the 'origin' or 'original' of the 3D triangle (because in the very first step, the corresponding 2D and 3D triangles are the same). The triangles are grouped to so called *layers* of paper.

Definition. *A* layer of paper *is a nonempty set of triangles meeting the following conditions:*

- *All triangles have their normals pointing in the same direction (this trivially holds for the original triangles).*
- *All original triangles from the set form a single, nondegenerated and connected polygon (possibly non-convex) on the crease pattern.*
- *All the 3D triangles from the set form a single, nondegenerated, connected and planar polygon.*
- *Every triangle belongs to exactly one layer.*

As a consequence of this definition, we see that the whole paper model can be parcelled into layers. A layer of paper can be imagined as the largest straight part of the paper bounded by creases or edges of the paper[1].

What is the term of layer good for? A layer is always the smallest unit of paper that can be moved, bent, or rotated. If a fold would go through the interior of a layer, a crease is created in the layer and it is subdivided into more smaller layers (convex layers always split to 2 parts, but non-convex layers can generate more sublayers).

4.1.2 Fold lines

Although it is not necessary to hold all the fold lines in memory[2], it shows that it is helpful to have quick access to them. So, every triangle remembers all the fold

[1] But it is allowed for a layer to have its boundary at a crease of angle 0°. This means it doesn't always have to be the largest straight part, but it is guaranteed that a layer doesn't have its boundary somewhere 'inside' (the boundary is always an edge of the paper or a crease).

[2] All fold lines could be just signalised by layers' boundaries.

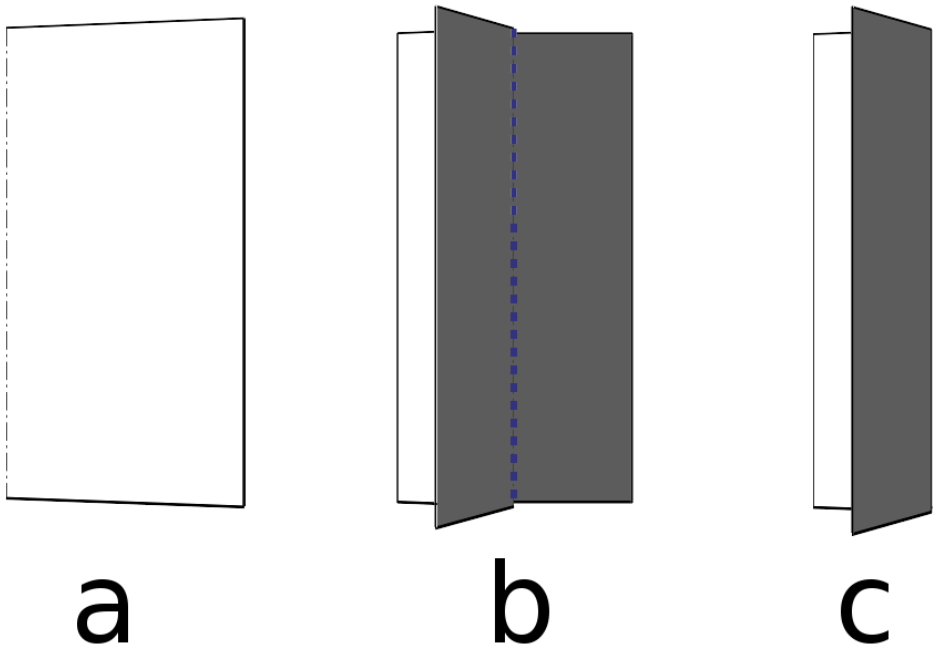


Figure 4.1: Different layers that can be bent

lines it lies along, and all fold lines have links to the triangles lying along them. Moreover, the folds remember the direction of the fold (mountain/valley), so it can be used later in generating crease patterns or for displaying the just done operations (as described in 1.4). They also remember their 'age' in the number of steps they were last 'touched' (used in a fold). The age can be used for blending old and probably unimportant creases.

4.1.3 That's all we need

The triangles, layers and fold lines are all we need to know to be able to correctly render and alter the model. More precisely, only 3D triangles and fold lines are needed for rendering, but layers and 2D triangles are needed for interaction with the model.

4.2 How to represent the operations?

Most operations consist of a set of lines, probably an angle or two, and... No, that isn't sufficient. Look at figure 4.1.

The last thing to be specified is which layers of paper should be bent. As you can see in the figure, no layers are bent (a), the first layer is bent (b), or both layers are bent (c).

So we have it - lines, angles and layers. Lines and angles are easy to represent and address. Layers are worse. There is no unambiguous designation of the

layers other than assigning them some artificial keys[3]. But what keys? Should we index the layers according to the order they have emerged? That sounds really unnatural and like a bad design idea — what if the bending algorithm got changed and the order of layers' emerging with it?

I have finally found a naturally-looking set of keys. The mapping is based on the active line[4] (the line the currently created fold will be folded along). The line always lies in a layer. So take this layer and make a stripe perpendicular to it and having its border lines at the endpoints of the active line (it's actually always a segment). Then record all intersections of this stripe with another layers not parallel with the stripe[5]. Sort the intersections by the distance from the viewer[6] (more precisely sort by the distance of their centres, because the intersections can have arbitrary angles towards the stripe and it wouldn't be clear what point to measure the distance from). Now, assign indices from 1 upwards to the sorted layers.

Although this key-mapping algorithm looks complex, it just does what people do naturally — how would I tell another person what layers to bend? I'd tell something like 'Bend the first four layers,' or 'Bend the topmost and bottommost layers.' Similar instructions can be found in many origami manuals. All these instructions refer to the same order of layers this algorithm defines.

So, in the data files, it is sufficient to first define the line to fold along, and then these keys can be used to describe the layers that will be bent. Figure 4.2 illustrates this.

4.3 How to bend the paper?

4.3.1 Which folds to implement?

We will show that the ability to perform valley and mountain folds (let's call them basic folds) is sufficient for making any operation of those listed in 1.3.

It is possible that during the decomposed operation the paper will get into inconsistent state (eg. the paper can intersect and tear), but after completing the whole operation, it will be either consistent, or the operation was invalidly specified.

Thunderbolt fold is just a composition of two basic folds. Do the first one, then the second one, and that's it.

Inside/outside reverse folds can also be substituted with two basic folds (here the paper will be torn in between of the operations). It's worth noting that reverse folds don't need to specify any angle of rotation, because there is only one nontrivial angle for which the operation will be valid (otherwise the paper

[3]We don't consider the option to label layers by themselves, because this is just useless - who would like to see the full list of triangles to be rotated in the model's file? There must be a better option.

[4]It is never needed to select layers without the need to select a line altogether, so this makes sense.

[5]Bending a layer parallel with the stripe would mean bending a layer perpendicular to another layer we will bend, which makes no sense. Although such folds are possible, it would be better to divide them into two separate folds.

[6]Since fold directions are dependent on the viewpoint, this brings no new dependencies to the algorithm.

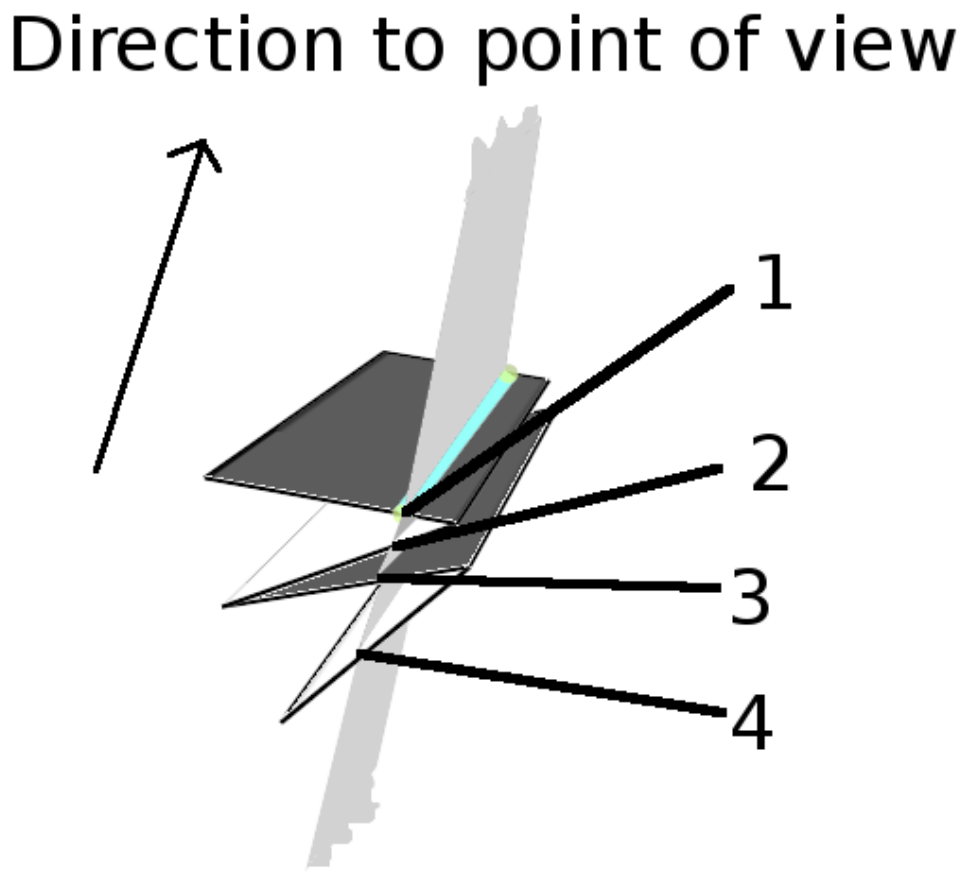


Figure 4.2: The indices mapped to layers according to the active line (azure on the image).

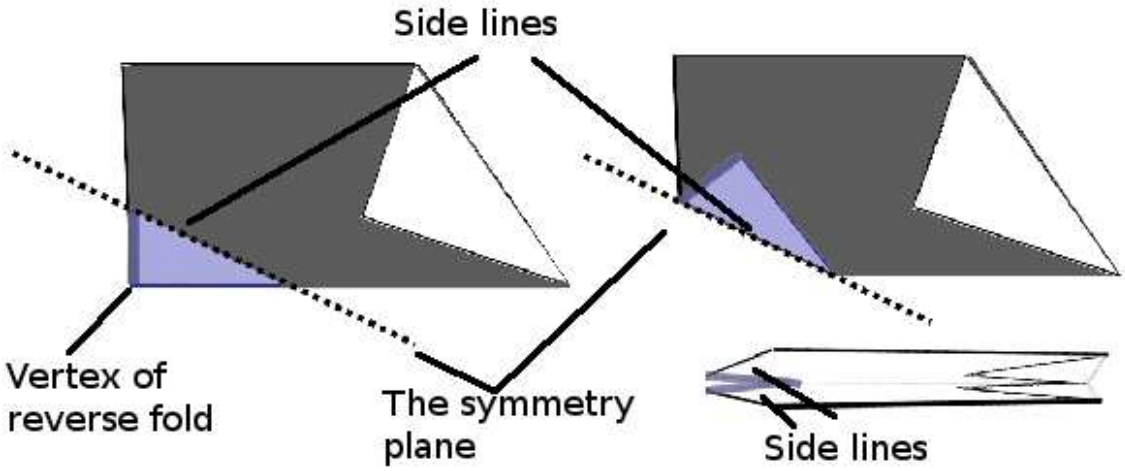


Figure 4.3: The symmetry of inside reverse fold.

would get curved). The angle is determined from the symmetry that reverse folds define. The symmetry is illustrated in figure 4.3

Inside/outside crimp folds are just double reverse folds and no additional paper tearing or intersecting can happen.

The pull operation also just rotates one part of the paper around the unfolded crease, so basic folds are sufficient.

The open operation is the most difficult, it could involve more layers to be rotated separately, but again, all of these steps will only be rotations of the paper around an axis (or multiple aces).

4.3.2 The basic fold operation

Let's get to the core of the folding. The operation is divided into two steps — triangle subdivision plus fold line creation (let's call it 'part 1'), and bending plus subdividing layers (calling it 'part 2').

Making creases

Firstly, the layers involved in the triangle subdivision are determined by the algorithm from 4.2. All triangles having a fold line going through their interior are subdivided into two or three smaller triangles. All triangles carry the list of their neighbors, so these lists are updated. These triangles copy the appropriate fold line data of the old triangle and add the new fold line to their fold line data. Internally, the direction of the fold is taken relatively to the triangle's normal[7]. So this has been the first part of the process.

Bending

Then the bending comes to the scene. We need to detect all layers that will be rotated (that are not only those found in part 1, but also all layers 'more distant' from the fold line but firmly connected to any already rotated layer).

[7]Here it is important to mention that the triangle subdivision process must retain the same normal direction, which is no problem, but it must be thought of.

Finding all layers to be rotated

I use an eager algorithm for finding these layers. I begin with the layers from part 1. I subdivide them into more smaller layers (along the fold line) and pick those that are to be rotated (will be discussed later). Then I go to all neighbouring layers and check if they can be added to the to-be-rotated set. What are the terms for adding a layer's neighbour? Firstly, a newly added fold line mustn't lie between these two layers. If it did, one layer would surely have been in the part we want to rotate, whilst the other one would have been in the part we want not to rotate. And secondly (which is nearly implicit from what has been said former), the layer must be a neighbour of an already rotated layer. These two conditions are sufficient to find the right set of triangles to rotate. And then comes the easiest part - taking all the layers and rotating them around a single axis.

4.3.3 Pitfalls

There are still some unanswered questions about the folding algorithm. Let's take a look on them.

What 'half' to rotate?

In the beginning of part 2, how do we choose the parts of the subdivided layers we want to rotate? Let's define a halfspace and take all layers the halfspace contains (the halfspace's border plane is the same as the plane of the stripe constructed for layer selection). What part of paper should 'stay' and what should rotate? Geometrically, the results are the same. But the real 3D coordinates of the points will differ. One possibility is to let the user to select a point in the 'half' of the model he wants to be rotated (these points are called 'refPoints' in the application). Otherwise, we have to guess. Here is a little heuristic. Firstly, we just select any of the 'halves' and find the layers to be rotated. But then the heuristic looks at the quotient of rotated and non-rotated triangles, and rotates the smaller of these two sets. This is based on the thought that the user mainly wants to rotate the smaller parts of paper, whilst the large remainder of the model should stay on place.

Layers in composite operations

The layer indexing and selecting algorithm fails while doing the second (third, fourth, ...) basic operation during a composite one (eg. a thunderbolt fold). We could define the layers using this algorithm after doing each substep, the disadvantage is that the paper doesn't have to be in consistent state between the substeps. So another method for layer selection is applied to composite folds. The first substep has its layers of operation defined normally by the indices. After doing the first substep, it passes the rotated layers to the input of the second substep, and so on. This practice is based on the fact that all the composite operations work ordinarily with the same number of rotated layers for all substeps[8], so it perfectly matches the real folding process.

[8] And from a human point of view, these are 'still the same layers'.

Angle of rotation for reverse folds

How to determine the angle of rotation for a reverse fold? We have already shown the reverse fold symmetry in figure 4.3. The symmetry is drawn as a line in the figure, but in fact it is a plane symmetry (the plane goes through the two side lines). So, to determine the angle, we just take the image of the reverse fold vertex (marked in the figure) and compute the angle between the image and the original.

Multiple layers in the same plane

Origamist could handle better the case when multiple layers of paper are folded so that they lie in the same plane. In this case, the ordering of layers for a fold operation isn't precisely defined. It doesn't do problems in the case of interactive model creation as long as the algorithm for layer indexing indexes the layers in the same plane equally. It will be needed to implement something like [13] has — the layers folded into a single plane remember their relative ordering, so it is possible to unambiguously identify them (independently on the indexing algorithm's subtleties). This would also have better effect on the paper intersection check.

4.4 Paper geometry checks

Origamist implements two checks of the paper geometry. The paper tearing check, and paper intersection check.

4.4.1 Paper tearing check

This check is made very easily. Since all triangles know both their 2D (original) position and their 3D position, all that is needed to do is to iterate over 2D neighbors and check if they also are neighbors in 3D (since the triangles carry the lists of their neighbors, this is really simple).

4.4.2 Paper intersection test

This test is rather more demanding. To check if the paper doesn't intersect, we need to find all mutual intersections of layers and check if they are at most 'touching'. Layers lying in the same plane are considered touching, layers with distinct parallel planes cannot intersect. The last possibility is that the layers' planes have a single common line. If the line isn't contained in at least one of the layers, the layers don't intersect. Finally, in the remaining case, we must check the touching. This is effectively done by constructing a halfspace with its border plane in one layer's plane, and checking if all of the other layer's points not lying in this border plane lie only on a single side of the defined halfspace (either all lie in the halfspace or none of them lies in it). Then the same is done vice versa.

This check runs asymptotically in $O(n^2)$, but practically it just consists of a small number of simple computations, and the count of layers isn't a very large number in basic and moderate origamis.

4.4.3 When to run these checks?

These checks cannot be run directly after a basic fold operation is done, because some composite operations need to get the paper into an invalid state during their work. If we could do the check after the basic folds, it would simplify the check, because it would be sufficient to take only the rotated layers into account. But since multiple basic folds can be done before the check, I rather run the check for the whole model[9].

4.5 Delayed operations

In Origamist Editor, all operations are completely done as soon as they are defined. However, this is not suitable for the viewer and for manual generation. The classical origami manuals divide most operations into two steps. In the first one, only the creases the operation creates are marked. All the bending is done in the second step. This is for better clarity. It is simpler to show a fold's direction when it lies in a straight plane, than if it goes along an already bent crease.

The first idea I had was just not to perform the bending at all in the first step. But that would work only if just one simple operation is permitted per step (pureland origami). If multiple basic operations were done in a step and the bending weren't performed, the layer indexing algorithm could fail for the second and subsequent folds. Also, eg. the thunderbolt fold uses the list of layers rotated in the first substep to define the list of layers to be rotated (and also intersected) in the second substep.

So another approach is needed. Before the very first operation is done in a step, all triangles are instructed to save their current 3D position (every triangle has a special data field for this purpose). Then all bending operations are done normally, and after the last operation has been performed, triangles just restore their original positions — but the created creases remain, which is exactly the desired result.

4.6 Struggling with floating-point arithmetic

The whole Origamist application is based on computing with floating-point numbers. Because high precision computations are needed in the application, the Java type *Double* is used. But even this isn't sufficient for the operations to be precise. I try to have all the model's vertices' coordinates between -1 and 1, because Double has the highest density in this interval, so it is supposed to have the least rounding errors.

All comparisons are done by checking if the absolute value of the difference of the compared numbers is less than a specified ϵ. What ϵ to take? I use 10^{-6} as it has a good mapping to the reality. As the longest side of the paper has the length of 1 unit (usually about 20 cm in physical world), the epsilon corresponds to $2 * 10^{-7}$ meters. Due to the thickness of a paper sheet (being something like 10^{-4} meters) this ϵ value seems to be appropriate.

[9] Hopefully it could be sufficient to accumulate the rotated layers for all the substeps and do the checks for them.

This ϵ is thus good for operations on the paper's vertices. But what about comparing angles or other paper-unrelated numbers? Well, for simplicity, Origamist uses the same ϵ everywhere, but the precision of computations would be better if the ϵ were scaled according to the unit it is used for. [23] provides some hints on how could this be done better.

4.6.1 ϵ-comparing maps?

I had the idea to develop a map (a tree) that could store 3D points or lines and search among them to return those that are ϵ-equal to a given search key[10]. Classical approaches aren't successful. Hashmaps don't work because ϵ-equal items don't have to have equal hash code, and there is no suitable mapping that could assign the hashcodes reasonably[11]. Classical binary search trees also fail quickly with floating-point numbers. Only plain sequential lists would work, but that is a slow solution.

I had the idea to use multidimensional interval trees, which could work for this purpose. Or R-trees could work, too. But instead of implementing these complex data structures (Java doesn't provide any default implementation), I have found out that they aren't needed at all. All the information that seemed to need these maps could be stored somehow else. Eg. the list of triangles' neighbors is stored inside the triangles and so on. This finally seems to be the most effective approach.

4.6.2 What about rational numbers?

The use of rational numbers for all computations would greatly increase the precision of the computations. But there are some facts that disallow this. The main problem are angles[12], which generally have irrational sines and cosines (and those are used in computing the rotations). So, if we restricted to angles with rational sines and cosines, it maybe would be possible to switch to rational numbers. But then we couldn't represent the often used angles like 45° (which has both sine and cosine equal to $\sqrt{2}/2$). So this question remains open.

4.7 Operation marks

How to place the marks for performed operations onto the model (in Viewer and manuals)? There are no precise rules on how to do this, so I have chosen a heuristic which looks to produce good results.

For every fold operation the furthest rotated point can be determined (furthest from the rotation axis). Then this point's position determines one endpoint of the mark, and the nearest point on the rotation axis to this furthest rotated point

[10] In order to store the neigbors lists and such things outside the triangles.

[11] Partitioning the space to ϵ-equal blocks doesn't work, because if you put a number from the lower bound of one block into the map and then search for a number even smaller, the search will fail even if the distance of these items is less than ϵ.

[12] Because Origamist doesn't need any other 'irrationalizing' operations like log or square roots.

determines the centre of the mark. So we have two points and can properly rotate and scale the mark.

For non-fold operations (like rotation), the marks are placed around the model's bounds.

4.8 Storage of the model files

In the stored models, I prefer not to save any 3-dimensional coordinates — due to the simple reason — all the model's 3D data are redundant, since they can be computed from the 2D positions. Maybe there will be the need to store non-model-related 3D data in the future, and then it will possibly have no other solution than to store the 3D coordinates (eg. if I wanted to implement custom operation mark positioning). But I prefer to store only the 2D coordinates.

5. User's manual

This chapter describes how to use Origamist Viewer and Origamist Editor.

Origamist applications are designed to help with creating and distribution of origami model manuals. You can find a lot of such manuals on the web or in books, but none of them is 3D or live! Origamist provides a full 3D preview of every step in the manual, and has excellent export options. So, from the 3D manual, you can export the standard manual formats, such as PDF manuals or images, but you can also generate some fancy formats (SVG - it is vector graphics format), or you can even export the whole folding process as an animation! Cool! Origami Editor provides support for creation of the 3D manual, so you can play with paper on your screen as if you held it in your hands!

Also, everyone can assemble his own sets of origami models (called listings here), sorted into categories, and distribute them wherever he wants.

Installation & supported platforms

How to install Origamist applications? Well, no installation is needed! Everything you need is a web browser with Java installed (ensure you have at least Java 6 or 1.6). Origamist Viewer and Editor are designed as web applets, so they can be loaded directly into a webpage. As Origamist is Java application, it is multiplatform - runs on Windows, Mac, Linux[1], Solaris and maybe further platforms.

The only thing you have to do, that isn't automatised, is pressing Yes or Run in a security dialog which appears shortly after the applet starts loading.

There are other ways of launching Origamist. It is also available as a JNLP application, which means it can be installed on your computer and then run as a standalone application without the need to launch web browser. If you want to use it this way, just click a link to the Origamist JNLP file you can find on the web, and the installation will start.

As the last option you can download Origamist and all of its libraries, and run it as a standard Java application (it is packed in a JAR file, so it should be sufficient to type the 'java -jar OrigamistViewer.jar' command and that should be everything).

5.1 Viewer

It will be best to describe the viewer's controls on screenshots, because there is no complex functionality. So, look at figures 5.1 to 5.10.

5.2 Editor

Origamist Editor is for creating new and editing existing origami models. It also has the same export options as viewer (except listing export, since the Editor

[1] Linux users, please use the Oracle JRE version of Java, other versions, such as Icedtea Java, have problems with Origamist

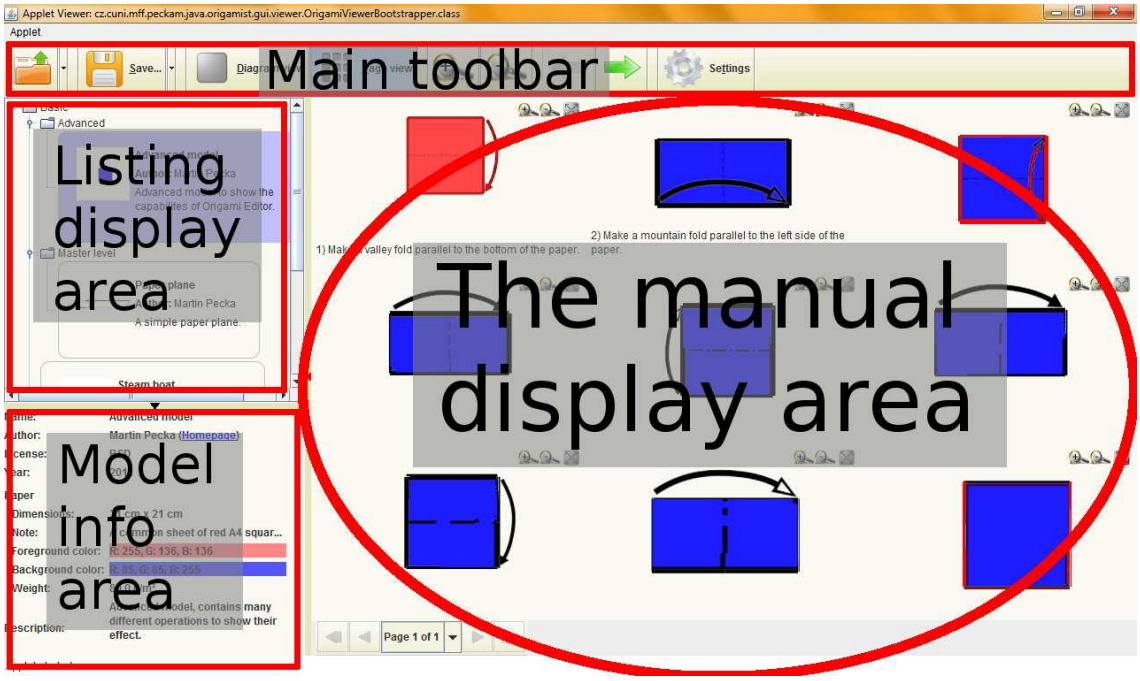


Figure 5.1: The viewer's main window. You see the main areas of the application.

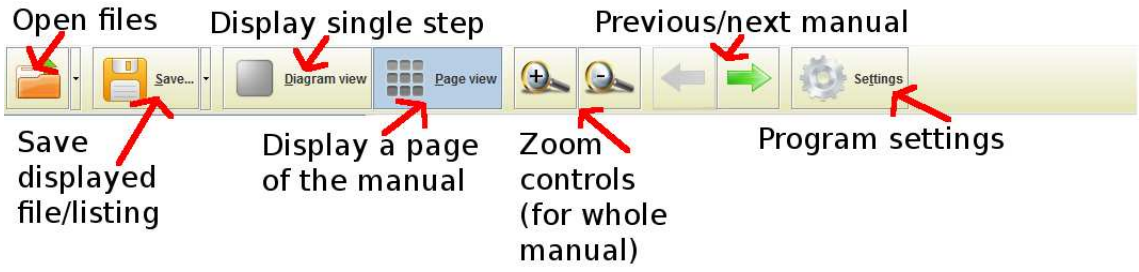


Figure 5.2: Viewer's toolbar. Most buttons are either intuitive or described later. Diagram View and Page View buttons switch the view modes - diagram mode displays only one step at a time in the Manual display area, whereas page mode displays a whole page of the manual in that area. The zoom buttons in this toolbar set zoom for all steps from the current manual, so they are an easy way how to zoom all steps at once.

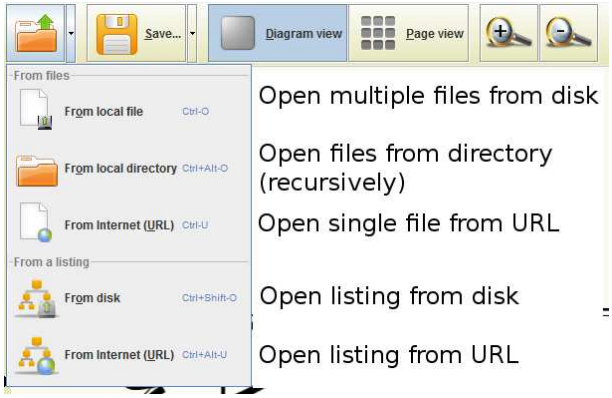


Figure 5.3: Open file dialog. First three buttons are for opening models (multiple local models by the first option, a whole directory of models by the second one, and a single model from Internet by the third one). Last two buttons are intended to handle loading of whole listings. So you can either load a listing from the local computer, or from Internet.

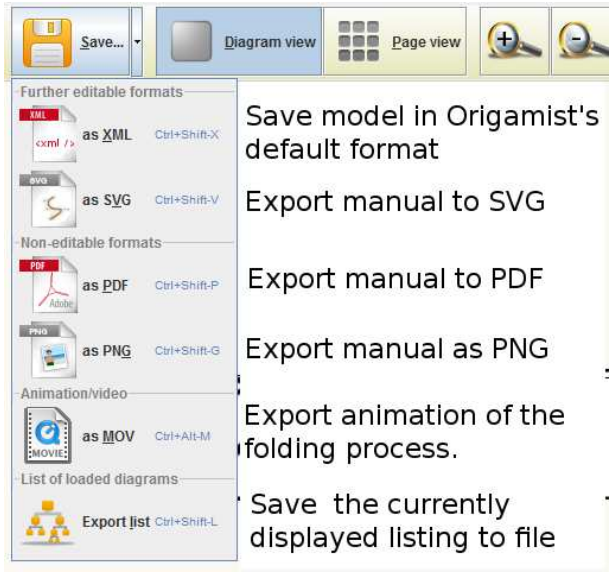


Figure 5.4: The save options of the Viewer. In the top-bottom order, they are: Save as XML, which is Origamist's default format and allows the model to be again loaded in Origamist applications; Save as SVG, which saves the manual as several SVG files (one per manual page; the filename entered to the save dialog will be used as a pattern for filenames of the generated files); Save as PDF, which saves the manual to a single PDF file; Save as PNG, which saves the manual to a series of PNG images (one per manual page; again, the entered filename serves as pattern); Save as MOV, which generates an animation of the folding process into a Quicktime MOV video file; Export the currently loaded listing (can be used to either save a remotely loaded listing, or to generate a listing from the loaded directory structure). After you select to export the model to a format, export options dialog will appear allowing you to specify the details of the exported manual (such as the used steps' descriptions language).

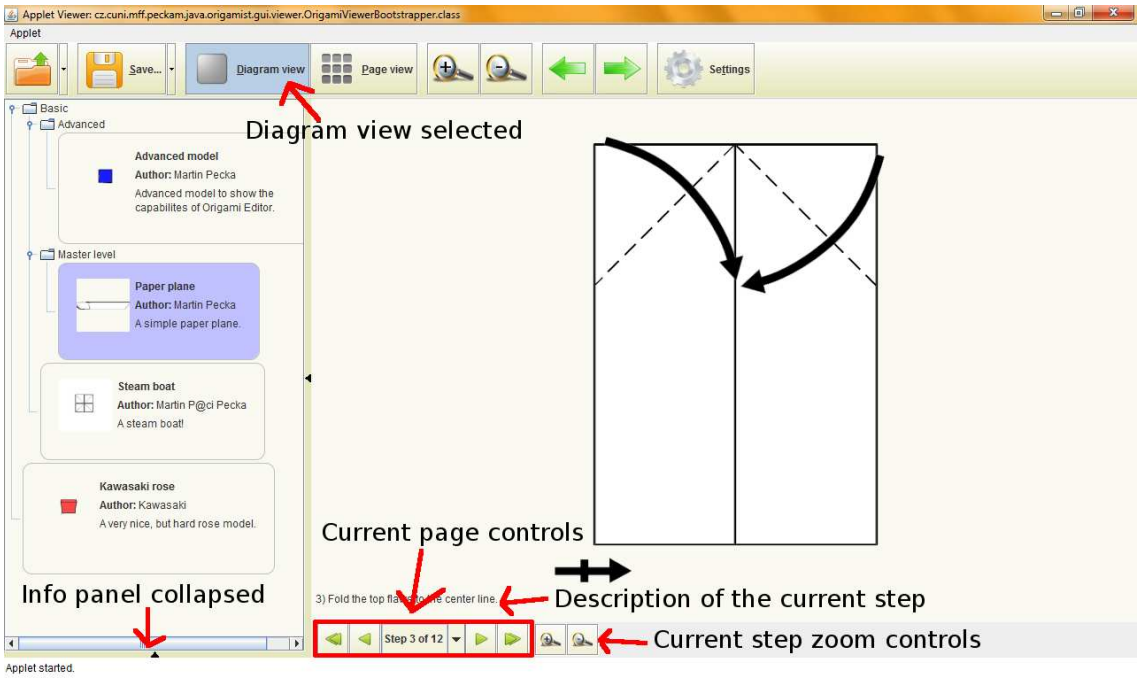


Figure 5.5: The viewer in diagram mode, where only one step is displayed at once. You may notice the info panel is collapsed by clicking the small arrow on the top of it. You also see the step controls which allow to browse the manual, description of the currently displayed step, and the zoom controls for just this one step (the zoom isn't applied globally).

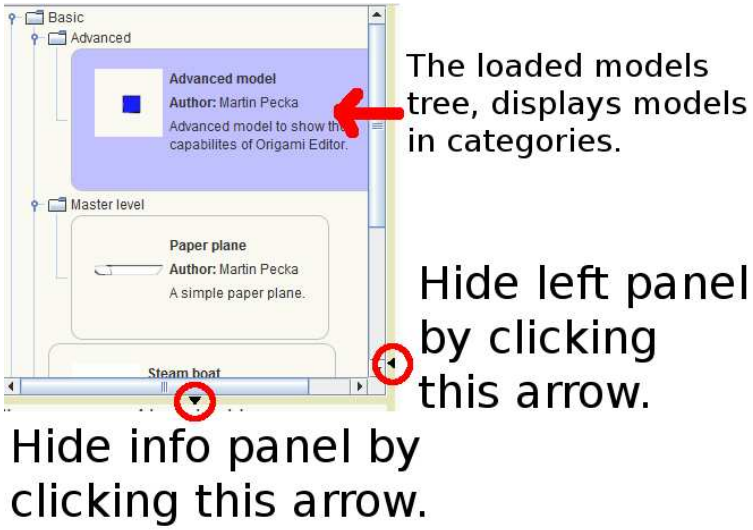


Figure 5.6: The loaded models tree displays all loaded models (either from a listing or from directories). If the models are loaded from directories, then the directory names will be displayed as category names, otherwise the category names will be loaded from the listing. You can hide the loaded models tree (along with model info panel) by clicking the small arrow on its right side.

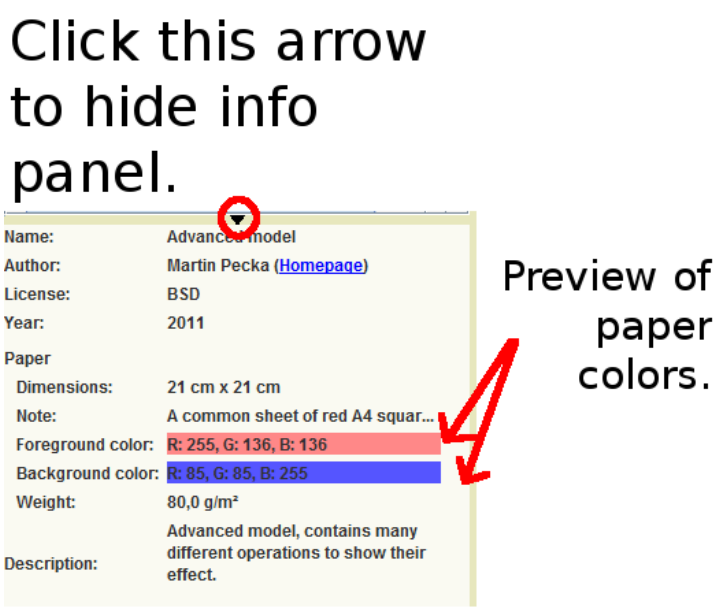


Figure 5.7: The model info panel. Displays some basic information about the currently displayed model. Notice the preview of paper foreground/background colors, by which the author says how the used paper should be colored. You may show/hide this panel by clicking the small arrow on its top.

doesn't handle listings).

To create a new model, click the New button (the leftmost in toolbar), and a dialog asking for inputting several information will open. Detailed description of this dialog is included in the PDF with Origamist manual (you should be able to get from the same source you have got Origamist from); although the dialog contains many fields and may look complex, it just asks the basic information - your name, name of the model, some description of the model, type of the paper the model is created from and so on. Most of the fields are optional, so it is fine to leave them blank. Please, pay attention to the License fields. Origamist wants that every model has clear rules of its usage. You can choose among several licenses. If you have invented the model, you can even issue it under the Public domain, which means that no restrictions apply to the model. If you just crate a model you have learnt from someone or from another manual, you should consider asking the original author if you can distribute the manual, and under which terms (if you just create the model for your own fun, you don't have to bother selecting a license, but then you cannot distribute it).

Now you can take a look at figures 5.11 to 5.14 to get used with the editor environment.

5.2.1 Creating the model

After you have created a new or opened an existing model, you can edit it. More precisely - you can edit only its last step, so if you have loaded an existing model, all the operation buttons will be disabled until you select the last step (however, there are so-called non-deforming operations, which are allowed to be added, changed or removed independently on whether they are or are not in the last

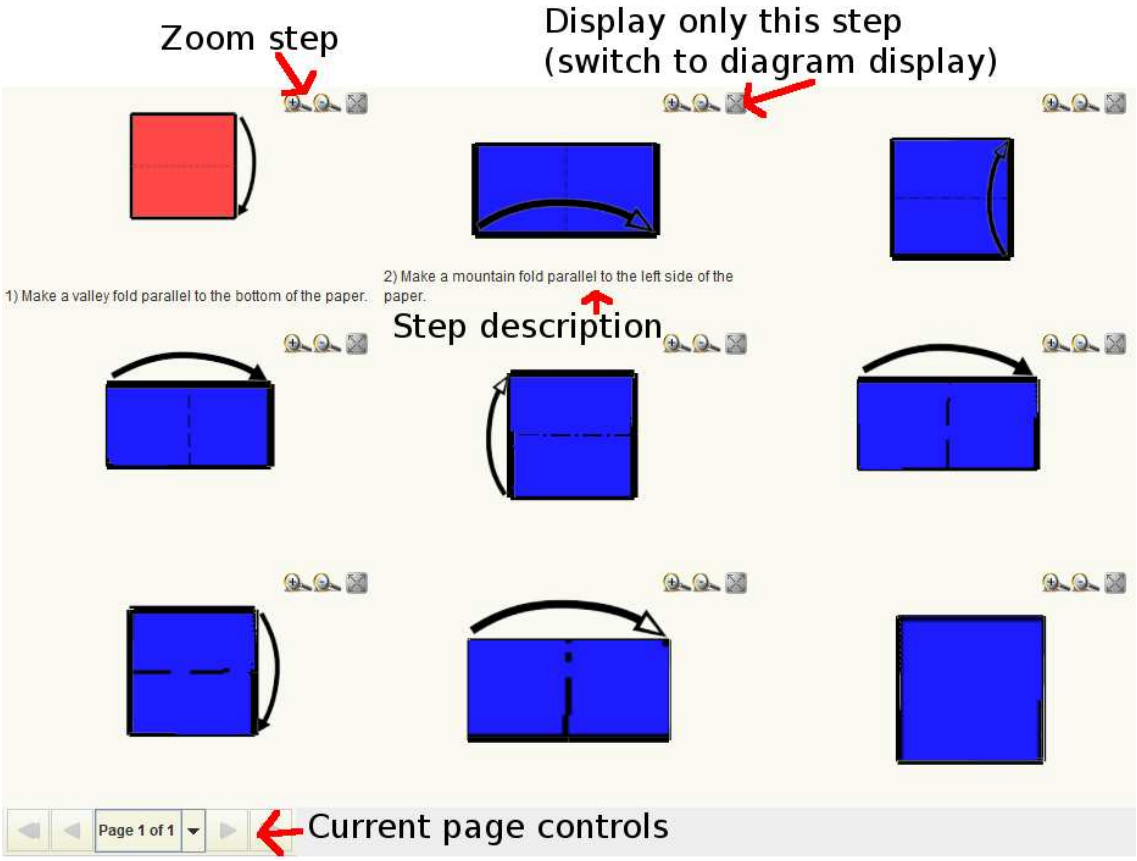


Figure 5.8: Manual display area in page mode, which shows a single page from the manual (the definition of the page layout, number of steps per page and so on is saved in the loaded model). Every step has its small toolbar with zoom buttons (these buttons zoom only that one particular step), and the fullscreen button, which displays the particular step in diagram mode.

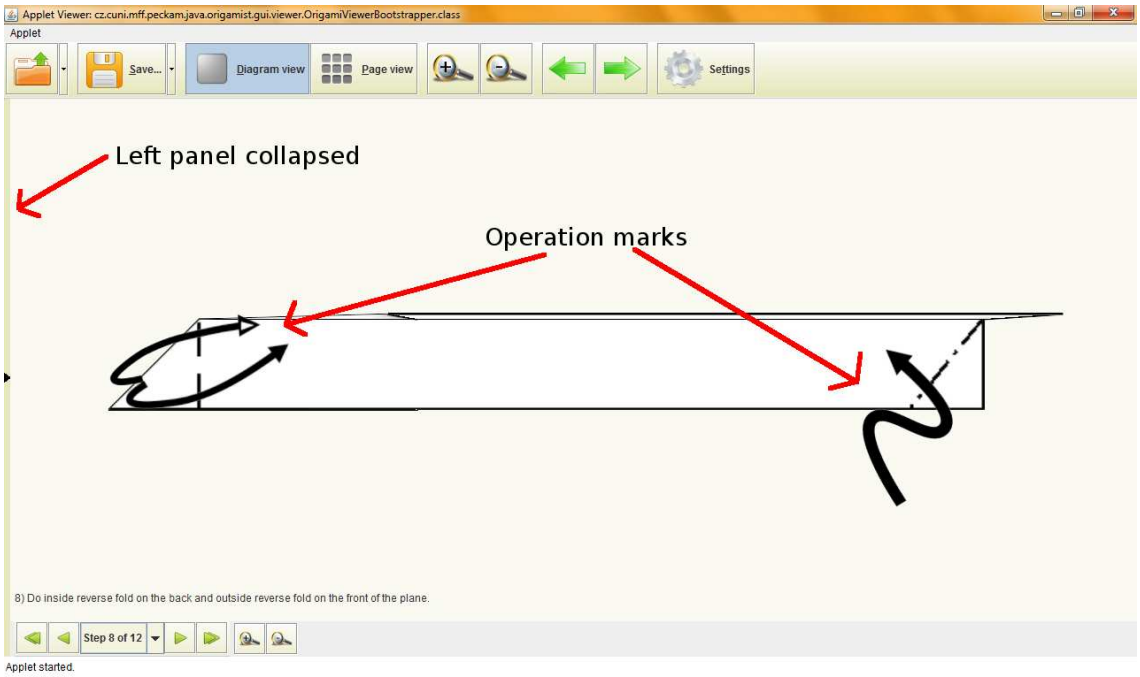


Figure 5.9: The viewer with the left panel completely collapsed (you can either do it manually, or it is done automatically if you load only a single model). Notice the operation marks designing what operations are done in this step.

Figure 5.10: Settings of the application. Program language is the language of program controls and dialogs (only several languages are available, and if the yours is not, English is selected), whereas language of diagrams is the preferred language of the steps' descriptions. This dialog is the same for both Viewer and Editor and the share these settings.

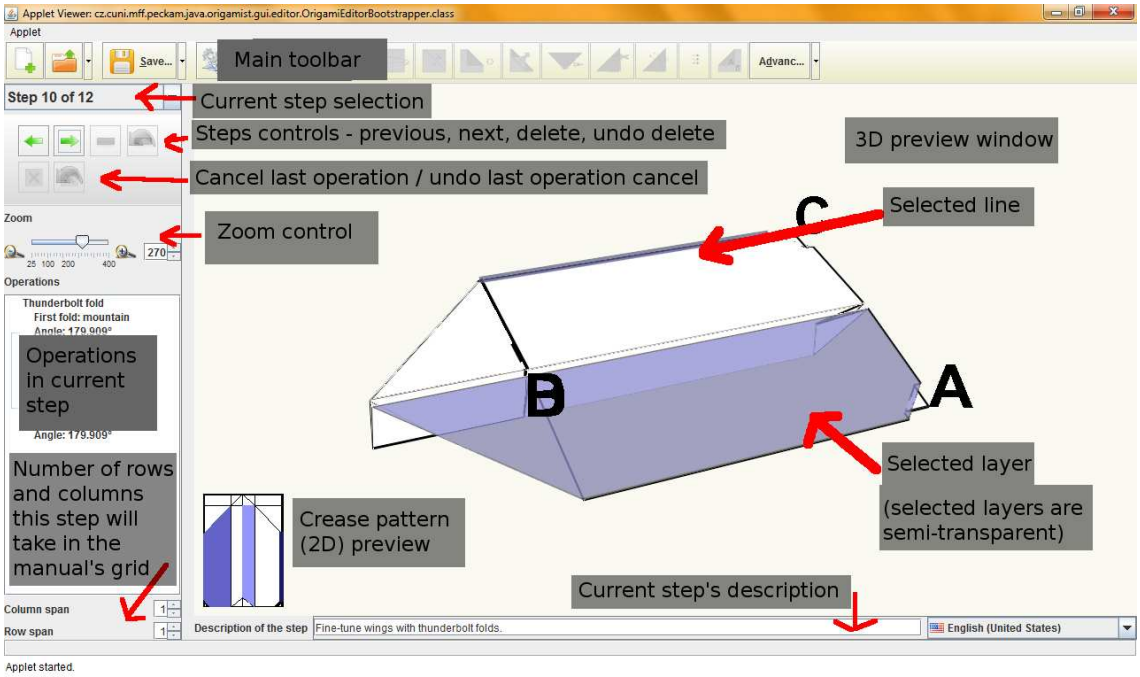


Figure 5.11: The main Editor window.

Figure 5.12: The main toolbar of Origamist Editor

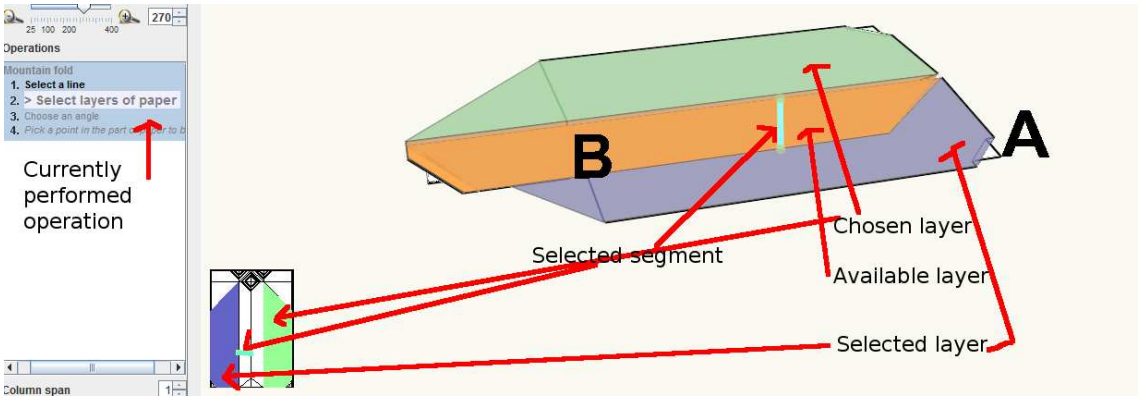


Figure 5.13: The meanings of layer colors.

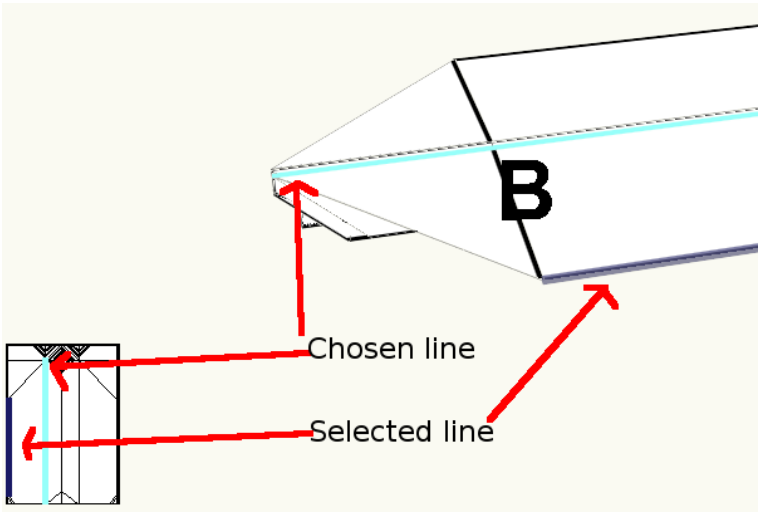


Figure 5.14: The meanings of line colors.

step. These are: rotate, turn over, mark).

You can select any operation among the operation buttons in the toolbar, and apply that operation to the model. If you don't know what that operation means, you can try reading the tooltip of the button (which shows when you hover it with mouse), and if even that doesn't help, then please refer to an origami book or a webpage dedicated to origami (Origamist uses a standard origami nomenclature, though, there isn't only one standard nomenclature in origami).

After selecting an operation, an item appears in the left operations list, that tells you what the editor needs to know to perform the operation. These are basically 1 to 5 steps where you select lines, points, paper layers, angles and so on.

Take a look in the upper right corner of editor, you can see a small help there. In this area, tips related to the currently performing operation appear. So, if you do eg. a mountain fold, the first tip in this area will tell you that it is needed to select a line, along which the fold will be bent; the second one will tell you to select the layers of paper the operation should affect, and so on.

Let's call these steps needed to be done to perform an operation as the 'operation arguments'. If you finish filling up one operation argument, you can move to the next argument by pressing Space or Enter. If you have just completed the last argument, the operation will be performed after pressing Space or Enter. If you have done a mistake, you can press Esc to return to the previous argument (if you are on the first argument, the whole operation will be cancelled).

5.2.2 How to select points, lines etc.?

So, if an operation argument requires to select some lines, points or layers of paper, what are you supposed to do?

First (without an operation selected) notice, that pressing the middle mouse button changes the mouse cursor. The standard cursor selects points, the next is the crosshair cursor, which selects lines, and the last one is the hand cursor, which selects layers of paper.

So, if you are asked to select something, you must switch to the right selection mode (however, most operation arguments will do the switch automatically).

Then you can begin selecting. You select/deselect by left-clicking. If there are multiple items under the cursor (eg. four layers of paper), use mouse wheel to choose among them (exactly one of the items will be highlighted, and that is the one that will be selected when you click). You can perform the selection both in the 3D preview or in the 2D preview (combining these two you should be able to select precisely those items you want to). Sorry, but there is no support for keyboard selection (but you can emulate it using keyboard-mouse emulation your operating system surely provides).

Now, some terminology is needed to be declared. Items can be either non-selected, selected, available or chosen (I know that there's only little difference between selected and chosen, but, please, treat it like terminology). Non-selected items have their default color and are opaque. Selected items are dark blue and semi-transparent (points are dark green). Only layers can be available, and available layers are orange and semi-transparent. Chosen items are green or azure, and chosen layers are semi-transparent. Every item can be highlighted (be the active one under the cursor), such items are opaque and are 'lifted up' to the foreground.

One more term needs to be defined - a layer of paper. You can imagine it as a straight (unfolded) part of paper. So, if you eg. fold the paper in half and again in half, you get 4 layers of paper.

What does this terminology mean? If you want to fill up an operation's argument, you need to choose the desired item. If no operation is being filled-up right now, you always only select the items. You also select items if you are in other selection mode than the current argument requires (eg. if you have to choose a point, and switch to line selection, you no longer choose - you select). Available layers will be defined later.

How to choose points

Points can only be chosen in point selection mode. However, both line and layer selection modes are also available. You can select lines or layers to constrain the places where a point can be selected. So, if it is hard to select the correct point you want, but it easy to select the layer it lies in, first select the layer and then only points lying in this layer can be chosen. You can do the same using lines.

Note that you can only select points on edges and creases, you cannot select points lying 'inside' a layer.

When you choose points, some of them are magnetic to help you. Magnetic points are squares (instead of circles). They snap to the start- and end-points of lines, and also to the exact halves of lines. Also, magnetic points will be shown to allow you select $90°/45°$ angles.

How to choose lines?

There are two types of lines — those existing, and those that don't exist. You can always choose a line in the line selection mode, and you can constrain it by selecting a layer (if a layer is selected, only lines going through this layer can be chosen).

There is one more option how to select a line (a non-existing one). You can switch to the point selection mode and select two border points of the line. Here available layers come to the scene. After you select the first point, some layers will be made available. Then only points lying in the available layers can be chosen as the opposite end of the line.

How to choose layers?

There's no magic behind layer selection. Just highlight the layer you want to choose, and click on it. Use mouse wheel to iterate over all layers under the mouse cursor.

What is needed to be known of layer selection, is what is it for. Well, imagine you have the paper twice folded in halves. It has four layers. Now you want to fold it again in half. You have three options — you can either fold the top two layers, the bottom two layers, or all four layers. That is what are layers for. If you select layers affected by an operation, you always select only those layers that are crossed by the fold line. The rest of affected layers (those that are firmly attached to the already affected layers) are detected automatically.

Be careful when thinking about what layers to select, because wrongly selected layers won't allow the operation to be completed.

5.3 How to compose it together?

Now you can fill up every argument an operation may need. So you are ready to create origami models! Every operation shows a description of the meaning of the current argument (in the top right corner), so you shouldn't have problems with understanding what is the current argument needed for.

5.4 A symmetry helper

Using the Symmetry button, you can manually define a symmetry on the model, which will then help you by offering repetition of operations that seem to be symmetric. Whenever the symmetry is broken, Origamist asks you to remove the symmetry, or you can choose to ignore it and to retain the symmetry for the next steps.

5.5 Use the New step button and fill up step descriptions

It is a good practice not to do many operations in one step. If they are repeating operations, you can hide them under the repeat operation. So, if you have done a small number of operations, fill in the step's textual description (in the bottom; you can type it in any number of languages you want) and click the Next step button (which changes to Add new step button in the last step). That's all. And — don't forget to save the final work!

5.6 A little tutorial

The following figures show a step-by-step tutorial on how to fold a frog base. Enjoy and good luck!

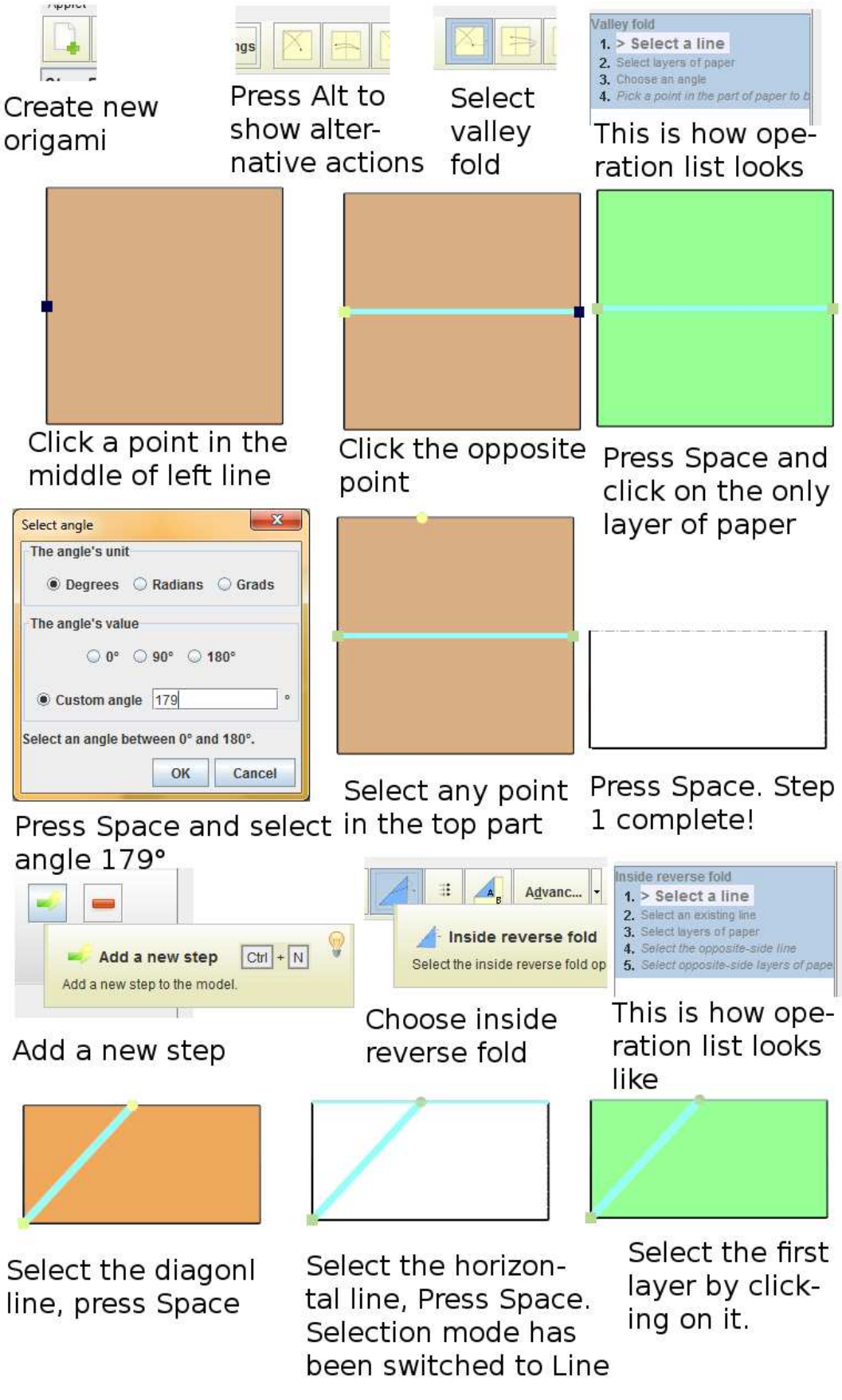


Figure 5.15: Frog base tutorial, part 1

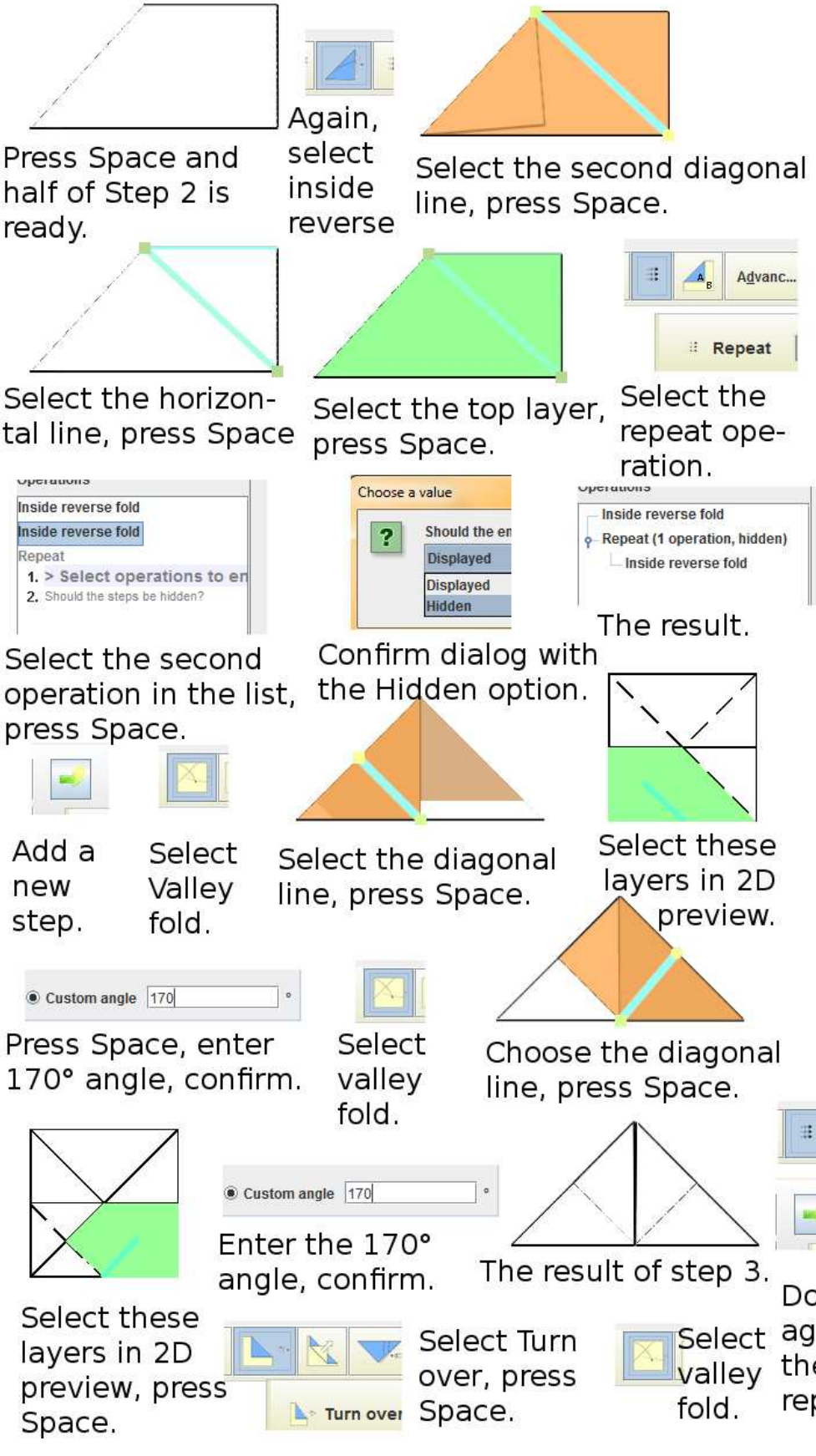
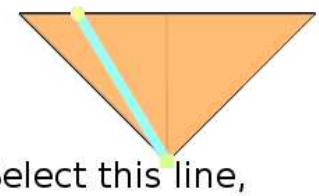
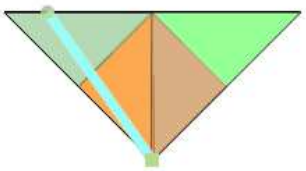
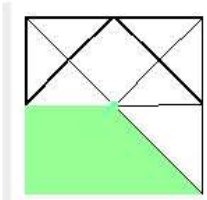


Figure 5.16: Frog base tutorial, part 2



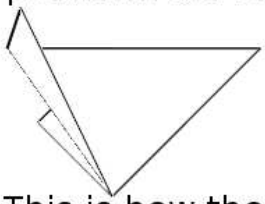
Select this line, press Space.





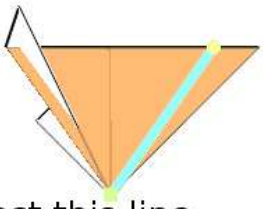
Select two layers (both 3D and 2D previews are shown).

Press Space, enter the 170° degree, press Space.

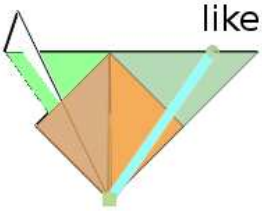


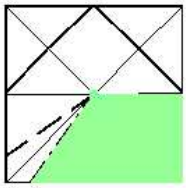
This is how the model should look like.

Select valley fold.

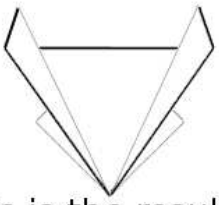


Select this line, press Space.

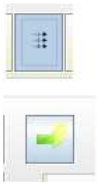




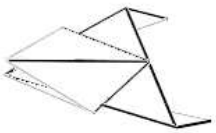
Select these layers (3D and 2D shown)



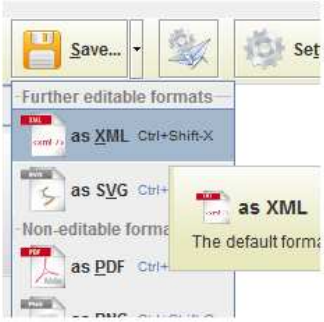
This is the result.



Hide the second fold under a repeat, create new step.



You can view the 3D result!



Save your work!

Figure 5.17: Frog base tutorial, part 3

6. A programmer's manual

The last chapter covers all information a developer may want to know about Origamist.

For information about the purpose and usage of Origamist applications, please refer to 5.

6.1 Source code versioning

The Git versioning system is used as the main code repository. The public online repository is hosted at `https://github.com/peci1/Origamist`. Everyone has read-only access to the repository, and if you wanted to contribute, contact me at peci1@seznam.cz and we'll discuss it.

6.2 Licensing

Both Origamist Viewer and Origamist Editor are issued under the GNU Affero GPLv3 license. Although I'd like to use a less restrictive license, as long as this application uses the iText library, this license is a need.

6.3 Used technologies

The project is written in Java 6. Compilation of the applications is done using Apache Ant. The application uses some third party extensions and libraries. I have divided them into two categories: compile-time and runtime.

6.3.1 Used runtime libraries

These libraries are located in the */lib* directory and are essential for running Origamist applications:

- *Java3D (j3dcore.jar, j3dutils.jar, gluegen-rt.jar and vecmath.jar)* is used as the 3D rendering engine.
- *JGoodies Forms (forms.jar)* is a great Swing layout manager.
- *JAXB2.0 runtime library (jaxb2-basics-runtime.jar)* is used for the Java $\leftrightarrow$ XML conversion.
- *Log4j (log4j-1.2.16.jar)* is used for logging.
- *Batik SVG Toolkit (files prefixed batik- and the following: pdf-transcoder.jar, xml-apis-ext.jar)* is used for SVG and PDF generation.
- *iText PDF library (itextpdf-5.1.0.jar)* is used for merging PDF files.
- *JPEG movie animation library (JPEGMovieAnimation.jar)* is used to generate Quicktime MOV animations.

- *JCalendar (jcalendar-1.3.3.jar, looks-2.0.1.jar)* is a Swing date picker component.

You can notice there is a bin/ subfolder of the /lib folder. More about this can be found in 6.10.1.

6.3.2 Compile-time libraries

The following libraries are located in the */lib-compile-time* folder and are essential only for building Origamist applications:

- *JAXB compile-time libraries (activation.jar, files prefixed jaxb-, jsr173_ api.jar)* are needed for the generation of Java classes out of XSD schemata.
- *JAXB2 Commons project libraries (files prefixed jaxb2- and commons-)* are some JAXB plugins.
- *Ant contrib package (ant-contrib-1.0b3.jar)* provides the <if> task for Ant.
- *Orangevolt Ant Tasks (orangevolt-ant-tasks-1.3.8.jar)* provide the <jnlp> task for Ant.
- *JUnit (junit.jar, org.hamcrest.core_ 1.1.0.v20090501071000.jar)* is used for unit testing.

6.4 Compilation

The whole compilation process is 'packaged' in the Ant's build.xml file in the root directory. The default target regenerates any JAXB-generated classes, compiles all sources, packs them into JAR archives, signs them and creates the appropriate JNLP launch files.

Origamist uses 3 JAR archives. *Origamist.jar* for the common codebase, and the *OrigamistViewer.jar* and *OrigamistEditor.jar* archives for the individual application parts.

Signing the archives is important - otherwise the application would launch neither as browser applet, nor as JNLP application. What is important - also all libraries the application uses have to be digitally signed! And if the libraries are signed with a different signature, Java displays a security warning, which is unnecessary, I mean. So, there is the *libs-sign* target which digitally signs all libraries in the /lib folder.

What digital signature is used for the signing process? There are the files *signapplet.bat.example* and *signapplet.sh.example* in the repository, which should be edited (and not included into the repository, since they need to contain the digital signature's password!) to one's needs and then their *.example* extension should be removed. Afterwards, the Ant target detects these files and uses them for signing (.bat files on Windows, .sh files on Linux). If these files are not present, Ant just writes a warning to the console and skips the signing process. The unsigned archives can be used for local testing if they are run by the commands *java -jar archive.jar* or *appletviewer preview.html*.

The default task requires user input for the JNLP task. The programmer should enter the absolute URL of the location where the JNLP files will be deployed. This sounds horribly, but it is a need, because the newest version of JNLP requires to have this absolute URL written in these files[1].

6.4.1 Other Ant targets

The build.xml file provides more targets the programmer can utilize:

- *jaxb* target regenerates Java classes from the XSD schemata.
- *run-** targets perform the default target and then open the editor or viewer as an applet in browser/applet in appletviewer/JNLP application.
- *just-run-** targets do the same as the run-* targets, but do not run the default target. So, if you have all sources compiled, you can use the just-run-* targets to launch the applications without recompiling them.
- *javadoc* target generates the JavaDoc documentation.

6.5 Deploying

6.5.1 Deploying as an applet

To deploy Origamist as an applet, upload /Origamist.jar, /OrigamistViewer.jar, /OrigamistEditor.jar, all /*.jnlp files and the whole /lib directory to your webserver. The HTML code needed to insert the applets into the page can be found in files /preview.html (for the viewer) and /preview-editor.html (for the editor).

6.5.2 Deploying as JNLP application

To deploy Origamist as a JNLP application, deploy it as an applet, but instead of using the HTML from /preview.html and /preview-editor.html use the direct link to the JNLP files. The user then just clicks the link on your webpage and the JNLP launch process has been started. Note that your server must send the JNLP file with the correct MIME type (application/x-java-jnlp-file). An example *.htaccess* file for Apache web server is provided in the root directory to handle this issue.

6.5.3 Deploying as standard Java application

If you want to deploy Origamist as a standard java application, just redistribute the /*.jar files and /lib directory. The OrigamistViewer.jar and OrigamistEditor.jar files are executable Java archives, so typing the command *java -jar OrigamistViewer.jar* is all the end user needs to run the application.

[1] It is weird, because some older versions of JNLP (6u20 and less, I think) didn't need an absolute path, a relative one was sufficient. But in a newer version the relative paths ceased working.

6.5.4 Parameters

Both viewer and editor applications accept some parameters (discussed later). To specify these parameters in applet, use the <param> tags inside the <applet> or <object> tag you are using to display the applet. If you use the applet code for New Generation Java Plugin, the JNLP files need to be edited, too. Here, put the <param> tags into the <application-desc> tag. If you want to specify parameters for Origamist deployed as JNLP application, just edit the JNLP files as has been written in this paragraph. Finally, to pass parameters to the standard application, just pass them as command-line arguments after the command to run the application.

6.6 Parameters of the applications

As has been said in the previous section, both applications accept some parameters. Here they are.

6.6.1 Viewer parameters

- *displayMode*: value *PAGE* means to switch the Viewer to page display mode after initialization, value *DIAGRAM* means to switch the Viewer to the diagram display mode (only one step is displayed).

- *modelDownloadMode*: value *ALL* means that all files referenced from the loaded listing (if any) will be downloaded after the listing is displayed. Value *HEADERS* means that only the metadata of all referenced models should be downloaded after initialization (to be able to display the models' names and thumbnails in the listing tree). Value *NONE* means that the model files will be downloaded after the user requests them (if he wants to display them). Finally, a non-negative integer value means that for the first n (where n is the given number) models, mode *ALL* will be used, and for the rest of them mode *NONE* will be used.

- *recursive*: value *RECURSIVE* means that any directories specified in the *files* parameter will be loaded recursively with their contents. A non-negative integer value specifies the number of recursion levels allowed for searching in subdirectories (specify 1 to only include files directly placed in the referenced directories). Note that the recursive search can only work on local machines, since remotely listing a HTTP directory isn't possible.

- *files*: contains a space-separated list of files to be loaded after initialization. Relative URLs are resolved with the applet's directory as the base URL. If the list contains a file named listing.xml, then all other items are ignored and the given listing is loaded. If an item references a local directory, then its contents and subdirectories may be also displayed, depending on the value of the *recursive* parameter.

6.6.2 Editor parameters

- *file*: this parameter can contain a URL of a single origami model's XML file that will be loaded after the Editor is initialized (see the Viewer's *files* parameter to see how the URL is resolved).

6.7 Localization

Both applications are ready-to-be-localised. The default version ships with English and Czech locales. The localization can be done the standard Java way. So, find all *.properties files in the JAR archives, create new files for your locale (take the base name of the file - all country and language codes stripped out, and add your country and language ending)[2]. The localisation strings are in the form 'key = value' and must follow the Java properties files conventions[3] — mainly — they have to be encoded as Latin1[4]. Localising plurals is simple - refer to the Java class MessageFormat[5] to see how to localise not only plurals, but also numbers, dates and so on.

You may notice that all localisation files contain the English files twice. It is needed to be able to force English locales on machines with non-English default locales. The filename with no additions is the basic fallback file for all unfound locales. If Java cannot find a non-fallback .properties file for the requested locale, it first tries to find a .properties file for the system locale. And that is the problem. If my system had had cs_CZ as its default locale, and I would have liked to set English in the application (and the _en.properties file wouldn't have existed), Java wouldn't have fallen back to the no additions file, but it would have found the file _cs.properties for my system locales, and thus Czech will remain to be set. So, English (or generally the default language's) locales need to be included twice.

6.8 Structure of the data files

Origamist distinguishes two types of files - model files, and listing files. Both of them are XML files, and thus can be distinguished by the namespace of the root element.

Model files use the namesapce
`http://www.mff.cuni.cz/~peckam/java/origamist/diagram/v2`, where v2 is the latest version of the model file schema (and thus the element's schema determines the version of the model file).

Listing files must be named exactly 'listing.xml' and must have their root element within the namespace
`http://www.mff.cuni.cz/~peckam/java/origamist/files/v1`.

The structure of these files is defined by these three schemata:
/resources/schemata/common_v1.xsd,

[2] Eg. if you are German and you would like to localise the file application.properties, create file application_de.properties

[3] `http://en.wikipedia.org/wiki/.properties`

[4] non-Latin1 characters must be entered as Java Unicode escapes, \uxxxx .

[5] `http://download.oracle.com/javase/1.4.2/docs/api/java/text/MessageFormat.html`

/resources/schemata/files_v1.xsd and
/resources/schemata/diagram_v2.xsd.

Names of the elements are designed to be self-explaining, but if you have doubts about the meaning, try to read JavaDoc for the associated Java class (in most cases this class is specified by an <implClass> tag inside the element's definition in XSD).

6.9 Model file versions

Origamist has built-in support for converting between older and newer versions of the model files. As the application will develop, the need to change the schemata is unavoidable. So Origamist keeps all the schemata and defines a mechanism to convert between different versions (only old to new direction is supported). The files can either be transformed by XSL transformations, or some Java code can be executed to convert between the versions. Look at ∼.gui.common.CommonGui#registerServices()[6] method, where the BindingsManager is constructed. This is the class (and the whole ∼.jaxb package) responsible for converting different file versions. An example of XSL transform usage is given here.

In result, the conversion should be fully transparent for the rest of the application.

6.10 Deeper in the code

6.10.1 How to load Java3D native files?

Java3D, as a 3D graphics library, needs to use some native libraries on different operating systems and platforms. The problem is that Java doesn't allow to load native libraries by default. They have to be preinstalled on the system, which was something I didn't want to accept. It would greatly decrease the user comfort.

Two ways exist how to solve this problem. The first one is JNLP's mechanism for loading libraries. JNLP documentation says that it should be sufficient to write a tag in the JNLP file and the library should have been automatically downloaded and installed. However, I have tried it several times, and it had never worked for me. Also, this solution would disable old applets.

The second solution is to use a custom classloader. Classloaders provide a mechanism to find not-yet-loaded class files, and — native libraries. So Origamist has all the native libraries in the /lib/bin folder to be able to use them. To use a custom classloader in applets, a bootstrapping technique is needed to be used. So, the bootstrapping applet starts with the default classloader, then creates an instance of the custom classloader, and loads the base applet using the custom classloader[7]. The bootstrapping applet must also delegate all of its other methods to the base applet. See ∼.gui.common.Java3DBootstrappingApplet.java for

[6] I will use the ∼ as a shorthand to the package prefix 'cz.cuni.mff.peckam.java.origamist'.

[7] It showed that all application-specific classes, libraries and so then need to be loaded by this same custom classloader, too.

implementation details.

6.10.2 JAXB and the generated classes

Files in packages ∼.common.jaxb, ∼.files.jaxb and ∼.model.jaxb are automatically generated from schemata /resources/schemata/common_versionNr.xsd, /resources/schemata/files_versionNr.xsd and /resources/schemata/diagram_versionNr.xsd, where versionNr is the newest version of the XSD. So, firstly, these files aren't intended to be manually edited. Files in packages without the .jaxb extension extend these generated files and they are intended to be the place for modifications. However, some code would be greatly duplicated if we hadn't touched the generated files. So, 5 JAXB (or, XJC) plugins were developed, which do the needed modifications to the generated files during the generation process. This way, for example, can all fields be turned into bound properties. The XJC plugins are in package com.sun.tools.xjc.addon .

In the previous section I have written that Origamist handles different versions of the model files. But how is it done, if the file version is included inside the XML we need to parse, when, to be able to parse XML, we need to know, against what schema to parse? This is done using a little trick. JAXB is firs commanded to parse the XML file without a schema, and a callback is called after JAXB reads the first line, which contains the file version. Then the correct schema can be chosen and JAXB can be instructed to use the proper schema.

The previous trick has one more advantage - it allows us to 'cleanly' implement the *HEADERS* option of *modelDownloadMode* applet parameter. We can wait until all metadata are read, and then we force the internet connection to be closed, and simulate the end tag for JAXB to be happy it encountered a valid document. This way, JAXB can convert even these incomplete files to Java objects and we can easily read them (the only disadvantage is that we must define that the content part of a model can be empty in the XSD).

6.10.3 ServiceLocator

As I have tried to minimise the count of static classes used, a ∼.services.ServiceLocator has been introduced. This class provides utilities that can be used in various parts of the application and have no direct connection to these parts.

The services are registered by the name of the interface they provide, and can be registered either as objects, or as callbacks that create the service at the time when it is first needed.

Currently, Origamist uses these services:

- *StepThumbnailGenerator* is a service that is able to generate thumbnails of the given origami and its step.
- *OrigamiHandler* is a service capable of serialising and deserialising XML models.
- *ListingHandler* is a service capable of serialising and deserialising XML listing files.

- *ConfigurationManager* provides access to the current application configuration (such as selected language).
- *TooltipFactory* is a GUI service that generates nice-looking tooltips used all over the application[8].
- *BindingsManager* is a service capable of converting between different versions of the XML files.
- *MessageBar* service is used only in Editor, and serves as a service that is able to display text messages to the user.

6.10.4 Receiving change events from the loaded model file

It is profitable to have the option of being notified of any changes made to the model's object form. But, since the model consists of many nested (not in the Java language meaning) classes, it would be very hard, or even impossible, to correctly install a listener to, say, a fourth-level class. You must have added listeners to all its containers that would have manage attaching and detaching of the listener if their containers change, and so on.

So, rather, a cumulative listener model is implemented. This means that every object notifies its container of changes in its properties, and in properties of its children. This way all events 'bubble up' to the parent container (the Origami class), so the programmer can just attach a listener to the origami object and listen to changes in eg. fourth-level properties. As some data fields can have equal names, while being in different classes, it is needed to distinct such names. This is done so that every object sending an event to its container prefixes the event with the name of the property it is attached to in the container. This way we get unique identifiers. It has also the advantage of the ability to attach prefixed listeners, which are listeners that listen to all changes in a property or its sub-properties.

It is also important to be notified of changes in list properties. Java has no support for such listeners, so I wrote a custom ∼.utils.ObservableList interface and its implementation, which is a list that notifies its listeners of addition, removal or changes of its items.

The observable lists are also chained in the bubbling event hierarchy, so it is even possible to listen to changes in properties of any item of the specified list.

As a result, we have a very complex and powerful system of property change listening and are able to detect every change in any arbitrarily complex hierarchy.

All of this functionality is hidden into the ∼.common.GeneratedClassBase class, which is the ancestor of all JAXB-generated classes, and into several JAXB plugins.

6.10.5 Package structure

Here comes the structure of the application's packages and their purpose:

- *com.sun.tools.xjc.addon* package contains JAXB plugins

[8]Swing only accepts strings as tooltips, not components, so this factory basically only wraps the tooltip text into HTML.

- *javax.swing.origamist* package contains some Swing components that could be used in other projects.
- *org*
  - *w3._2001.xmlschema* package contains some JAXB-generated adapters.
  - *w3c.tools.timers* package contains a time-based event queue.
- *resources*
  - *images* package contains all used images.
  - *schemata* package contains all used XSD and XSLT files.
- *cz.cuni.mff.peckam.java.origamist*
  - *common* package contains JAXB-generated classes (in the .jaxb subpackage) and extensions to them, that are common for both model and listing files.
  - *configuration* package contains a configuration manager and its helper classes
  - *exceptions* package contains custom exceptions
  - *files* package contains JAXB-generated classes (in the .jaxb subpackage) and extensions to them, that are used to handle listing.xml files.
  - *gui*
    - *common* package contains GUI classes common for both Viewer and Editor.
    - *editor* package contains Editor-specific GUI classes.
    - *viewer* package contains Viewer-specific GUI classes.
  - *jaxb* package contains classes utilised by the JAXB mechanism to convert different file versions.
  - *logging* package contains only one class, GUIAppender, which is a logger outputter that handles fatal errors specially - displays a warning message and then stops the applet.
  - *math* package contains all mathematical and geometry-oriented classes (some geometrical classes are defined in the Java3D vecmath library).
  - *model* package contains JAXB-generated classes (in the .jaxb subpackage) and extensions to them, that are used to handle model XML files.
  - *modelstate* package contains the most important classes - the classes the perform paper bending, and classes representing some paper attributes and properties.
  - *services* package contains most service classes.
  - *tests* package contains some JUnit tests (not much).
  - *unused* package contains classes developed for the needs of Origamist, but abandoned for some reason.
  - *utils* package contains the rest — mainly utility classes.

If you would like to know precisely, what is a class or function for, consult its JavaDoc. I have tried to make it as useful as it can be.

6.10.6 The paper bending core

If you are looking for the classes responsible for folding the paper, look into the ∼.modelstate package, especially the ModelState class. This is the heart of Origamist.

6.10.7 Miscellaneous Java-related programming problems

If you are interested in some weird, hacky, or just surprising solutions to things one would think should be easy in Java, see the /doc/odd_things_and_how_to_fix_them.txt file.

Conclusion

Writing Origamist has been a great experience, I have taught a lot of things not only about programming. A common sheet of paper can get very interesting in the while one begins to think about the mathematics and physics it is affected by. Some of these phenomenons are big algorithmic problems, whereas others have a surprisingly simple answer.

If I should look into the future of Origamist, it has a stable, extendible and promising application base, but the folding algorithms need to be developed to much higher detail. Folding advanced models is still nearly impossible in Origamist (due to the lack of the Open operation, which is broadly used), and the floating-point rounding errors also sometimes influence the model in a way that a in-reality-valid fold cannot be done. On the other side, for simple models and pureland origami, Origamist provides an exciting environment, and its export options are very promising (and exporting the origami as a 3D model wouldn't be complicated, which can bring a brand new set of use cases).

In the recent times, 3D technology and visualisation is spreading all over the world in a high speed, so I hope Origamist will be a part of this 3D wave and that it will help more people fold (at least virtually) nicer and more complicated origami models, because the art of origami is just amazing.

Bibliography

[1] TEMKO, Florence; JACKSON, Paul. *Paper pandas and jumping frogs.* China : China books & periodicals, 1986. 133 p. ISBN 0-8351-17701770-7.

[2] LANG, Robert J. *Origami design secrets : mathematical methods for an ancient art.* Massachusetts (USA) : A K Peters, Ltd., 2003. 585 p. ISBN 1-56881-194-2

[3] PEN, K. *Origami Paper Folding: Rules of the Game.* Associated content [online]. 2009 [cited 2011-05-25]. Available at WWW: `http://www.associatedcontent.com/article/1818997/origami_paper_folding_rules_of_the.html?cat=24`.

[4] FUSE, Tomoko. *Unit Origami: Multidimensional Transformations.* Japan : Japan Publications, 1990. ISBN 0870408526.

[5] LANG, Robert J. *Origami in Action: Paper Toys that Fly, Flap, Gobble and Inflate.* Massachusetts (USA) : St. Martin's Griifin, 1997. ISBN 0-312-15618-9.

[6] SMITH, John S. *Pureland Origami.* Great Britain: British Origami Society, 1980.

[7] TEMKO, Florence. *Kirigami Home Decorations.* China : Tuttle Publishing, 2006. ISBN 0-8048-3793-7.

[8] MOBILEREFERENCE. *Asian Art.* USA : SoundTells, 2003.

[9] WIKIPEDIA CONTRIBUTORS. *Origami techniques.* Wikipedia, The Free Encyclopedia [online]. 2011 [cited 2011-05-25]. Available at WWW: `http://en.wikipedia.org/w/index.php?title=Origami_techniques&oldid=420452758`

[10] DEMAINE, E. D.; DEMAINE, M. L. *Recent results in computational origami.* In Origami : Proceedings of the 3rd International Meeting of Origami Science, Math, and Education. California (USA) : Monterey, 2001. pp. 3–16. Available at WWW: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.8809&rep=rep1&type=pdf`

[11] BERN, Marshall; HAYES, Barry. *The complexity of flat origami.* In Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 175ďż˝183, Atlanta, January 1996. Reprinted in Proceedings of the 3rd International Meeting of Origami Science, Math, and Education, 2001

[12] LANG, Robert J. *TreeMaker 4.0: A Program for Origami Design.* [online] 1998 [cited 2011-05-25]. Available at WWW: `http://www.langorigami.com/science/treemaker/TreeMkr40.pdf`

[13] MIYAZAKI, Shin-ya; YASUDA, Takami; YOKOI, Shigeki; TORIWAKI, Junichiro. *An ORIGAMI Playing Simulator in the Virtual Space.* In The Journal of Visualization and Computer Animation, vol.7(1) Japan, 1996. pp.

25–42. Available at WWW: http://www.om.sist.chukyo-u.ac.jp/main/research/origami/journal/jvca.html

[14] Ida, Tetsuo; Takahashi, Hidekazu; Marin, Mircea; Kasem, Asem; Ghourabi, Fadoua. *Computational Origami System Eos* Japan : Department of Computer Science, University of Tsukuba. Available at WWW: http://www2.score.cs.tsukuba.ac.jp/publications/Eos.pdf/view

[15] You, Zhong; Kuribayashi, Kaori *A NOVEL ORIGAMI STENT* In 2003 Summer Bioengineering Conferencee, June 25–29. Florida (USA), 2003. pp. 257–258. Available at WWW: http://www.tulane.edu/~sbc2003/pdfdocs/0257.PDF

[16] Lawrence Livermore National Laboratory. *Eyeglass space telescope*. National Ignition Facility & Photon Science : Bringing Start power to Earth [online]. 2002 [cited 2011-05-25]. Available at WWW: https://lasers.llnl.gov/programs/psa/pdfs/technologies/eyeglass_space_telescope.pdf

[17] Shibayama, Y.; Arai, H.; Matsui, K.; Hama, K.; Ushirokawa, A.; Natori, M.; Takahashi, K.; Wakasugi, N.; Anzai, T. *SFU solar array.* European Space Power Conference. ESA Special Publication. 1989. , pp. 557-562. Available at WWW: http://adsabs.harvard.edu/abs/1989ESASP.294..557S

[18] Gďż˝rtner, Torsten; Eriksson, Magnus; Fďż˝ltstrďż˝m, Jonas. EASi GmbH, *Advanced Technologies for the Simulation of Folded Airbags*, 2nd European LS-DYNA Conference, Gothenburg, Sweden, June 1999.

[19] Choi, Charles Q. *Solar Panel Productivity Boosted by Origami.* LiveScience.com [online]. 2010 [cited 2011-05-25]. Available at WWW: http://www.livescience.com/10625-solar-panel-productivity-boosted-origami.html

[20] Oracle *Supported System Configurations for Java SE 6 and Java For Business 6* Oracle.com [online]. [cited 2011-05-25]. Available at WWW: http://www.oracle.com/technetwork/java/javase/system-configurations-135212.html

[21] Oracle *Next Generation in Applet Java Plug-in Technology* Oracle : Sun Developer Network [online]. [cited 2011-05-25]. Available at WWW: http://java.sun.com/developer/technicalArticles/javase/newapplets/

[22] *Java 3D TM Downloads: Release Builds* java.net : The Source for Java Technology Collaboration [online]. [cited 2011-05-25]. Available at WWW: http://java3d.java.net/binary-builds.html

[23] Dawson, Bruce. *Comparing floating point numbers* Cygnus-software.com [online]. [cited 2011-05-26]. Available at WWW: http://www.cygnus-software.com/papers/comparingfloats/Comparing%20floating%20point%20numbers.htm

Attachments

A sample webpage displaying Origamist as a standard Java applet using the old Java Plugin

Listing 6.1: Classic Java Plugin HTML page code

```
<html>
    <body>
        <applet code="cz.cuni.mff.peckam.java.origamist.gui.viewer.
           OrigamiViewerBootstrapper"
             width="800"
             height="600"
             archive="
               lib/log4j-1.2.16.jar,
               lib/j3dcore.jar,
               lib/j3dutils.jar,
               lib/vecmath.jar,
               lib/jaxb2-basics-runtime-0.6.0.jar,
               lib/forms.jar,
               lib/batik-anim.jar,
               lib/batik-awt-util.jar,
               lib/batik-bridge.jar,
               lib/batik-codec.jar,
               lib/batik-css.jar,
               lib/batik-dom.jar,
               lib/batik-ext.jar,
               lib/batik-gvt.jar,
               lib/batik-parser.jar,
               lib/batik-script.jar,
               lib/batik-svg-dom.jar,
               lib/batik-svggen.jar,
               lib/batik-transcoder.jar,
               lib/batik-util.jar,
               lib/batik-xml.jar,
               lib/itextpdf-5.1.0.jar,
               lib/JPEGMovieAnimation.jar,
               lib/pdf-transcoder.jar,
               lib/xml-apis-ext.jar,
               Origamist.jar,
               OrigamistViewer.jar
            ">

            <param name="files" value="diagrams/paper_plane.xml
               diagrams/advanced_diagram.xml diagrams/diagram.xml" /
               >
            <!-- These are the files to be displayed at startup. -->
```

```
39          <param name="startupMode" value="page" />
            <!-- Startup mode of the page, either "page" or
41          "diagram". -->

43          <param name="java_arguments" value="-Xmx1024m" />
            <!-- Require more memory than the default portion. -->
45      </applet>
        </body>
47 </html>
```

A sample webpage displaying Origamist as applet using new generation Java plugin, if the client has installed it

Listing 6.2: New Generation Plugin HTML page code

```
1 <html>
    <body>
3       <applet code="cz.cuni.mff.peckam.java.origamist.gui.viewer.
          OrigamiViewerBootstrapper"
            width="800"
5           height="600"
            archive="
7             lib/log4j-1.2.16.jar,
              lib/j3dcore.jar,
9             lib/j3dutils.jar,
              lib/vecmath.jar,
11            lib/jaxb2-basics-runtime-0.6.0.jar,
              lib/forms.jar,
13            lib/batik-anim.jar,
              lib/batik-awt-util.jar,
15            lib/batik-bridge.jar,
              lib/batik-codec.jar,
17            lib/batik-css.jar,
              lib/batik-dom.jar,
19            lib/batik-ext.jar,
              lib/batik-gvt.jar,
21            lib/batik-parser.jar,
              lib/batik-script.jar,
23            lib/batik-svg-dom.jar,
              lib/batik-svggen.jar,
25            lib/batik-transcoder.jar,
              lib/batik-util.jar,
27            lib/batik-xml.jar,
              lib/itextpdf-5.1.0.jar,
29            lib/JPEGMovieAnimation.jar,
              lib/pdf-transcoder.jar,
```

```
31              lib/xml-apis-ext.jar,
                Origamist.jar,
33              OrigamistViewer.jar
            ">
35
                <param name="jnlp_href" value="origami_viewer.
                    jnlp">
37              <!-- This is the only difference between the old
                    and new generation plugins. Listing of the
                    archive attribute and filling up params in
                    this page isn't used by the new generation
                    plugin, but it is there to work either on user
                     clients with old plugins. -->

39          <param name="files" value="diagrams/paper_plane.xml
                diagrams/advanced_diagram.xml diagrams/diagram.xml" /
                >
            <!-- These are the files to be displayed at startup. -->
41
            <param name="startupMode" value="page" />
43          <!-- Startup mode of the page, either "page" or
            "diagram". -->
45
            <param name="java_arguments" value="-Xmx1024m" />
47          <!-- Require more memory than the default portion. -->
        </applet>
49       </body>
</html>
```

A sample JNLP file to run Origamist viewer

Listing 6.3: JNLP file to launch Origamist viewer

```
<?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE jnlp PUBLIC "-//Sun Microsystems, Inc//DTD JNLP
    Discriptor 1.5//EN" "http://java.sun.com/dtd/JNLP-1.5.dtd">
  <jnlp spec="1.0+" codebase="http://www.ms.mff.cuni.cz/~peckam/
    origamist" href="origami_viewer.test.jnlp"><!-- codebase must
    point to the !absolute! location of the JNLP file, otherwise
    the newest Java versions won't run it! -->
4   <information>
      <title>Origamist viewer</title>
6     <vendor>Martin Pecka</vendor>
      <homepage href="http://github.com/peci1/Origamist"/>
8     <description>A viewer for digital origami models.</description>
      <description kind="short">An origami viewer application.</
        description>
10    <offline-allowed/>
    </information>
```

```
<security>
  <all-permissions/>
</security>
<resources os="Windows">
  <property name="sun.java2d.noddraw" value="true"/>
</resources>
<resources os="Mac OS X">
  <property name="j3d.rend" value="jogl"/>
</resources>
<resources>
  <j2se version="1.6+" href="http://java.sun.com/products/autodl/
     j2se" initial-heap-size="128m" max-heap-size="1024m"/>
  <jar href="OrigamistViewer.jar" main="true"/>
  <jar href="Origamist.jar" main="true"/>
  <jar href="lib/batik-anim.jar"/>
  <jar href="lib/batik-awt-util.jar"/>
  <jar href="lib/batik-bridge.jar"/>
  <jar href="lib/batik-codec.jar"/>
  <jar href="lib/batik-css.jar"/>
  <jar href="lib/batik-dom.jar"/>
  <jar href="lib/batik-ext.jar"/>
  <jar href="lib/batik-gvt.jar"/>
  <jar href="lib/batik-parser.jar"/>
  <jar href="lib/batik-script.jar"/>
  <jar href="lib/batik-svg-dom.jar"/>
  <jar href="lib/batik-svggen.jar"/>
  <jar href="lib/batik-transcoder.jar"/>
  <jar href="lib/batik-util.jar"/>
  <jar href="lib/batik-xml.jar"/>
  <jar href="lib/forms.jar"/>
  <jar href="lib/gluegen-rt.jar"/>
  <jar href="lib/itextpdf-5.1.0.jar"/>
  <jar href="lib/j3dcore.jar"/>
  <jar href="lib/j3dutils.jar"/>
  <jar href="lib/jaxb2-basics-runtime-0.6.0.jar"/>
  <jar href="lib/log4j-1.2.16.jar"/>
  <jar href="lib/pdf-transcoder.jar"/>
  <jar href="lib/vecmath.jar"/>
  <jar href="lib/xml-apis-ext.jar"/>
</resources>
<applet-desc name="Origami viewer" main-class="cz.cuni.mff.peckam
   .java.origamist.gui.viewer.OrigamiViewerBootstrapper" width
   ="800" height="600" documentbase=".">
  <param name="files" value="diagrams/listing.xml"/><!-- These
     are the files to be displayed at startup. -->
  <param name="displayMode" value="PAGE"/><!-- Startup mode of
     the page, either "page" or "diagram". -->
</applet-desc>
```

```
</jnlp>
```

Contents of the attached CD

Here is the structure of the files located on the attached CD:

- *.git* The complete Git repository of this project (may be a hidden file)
- *bin* The compiled .class files.
- *diagrams* Sample diagrams and export outputs of Origamist.
- *doc* Documentation.
  - *thesis* Sources of this bachelor thesis.
  - *odd_things_and_how_to_fix_them.txt* Some interesting solutions to various Java-related problems.
  - *podrobna_specifikace.pdf* The detailed specification of this project, in Czech.
  - *roadmap.odt* The roadmap of the project, in Czech.
  - *specifikace.pdf* The rough specification of this project, in Czech.
- *javadoc* JavaDoc per-class documentation.
- *lib* Runtime libraries.
- *lib-compile-time* Compile-time libraries.
- *src* The source codes of Origamist.
- *tests* Resource files for JUnit tests.
- *.htaccess* A sample Apache server configuration file that enables serving JNLP files with the correct MIME type.
- *browser.bat.example, browser.sh.example* Templates of shell scripts that run a web browser.
- *build.xml* The build file (is run by Apache Ant).
- *java.policy.applet* A needed security policy file.
- *origami_editor.jnlp* JNLP file designed for distribution.
- *origami_editor.test.jnlp* JNLP files designed for local testing.
- *origami_viewer.jnlp* JNLP file designed for distribution.
- *origami_viewer.test.jnlp* JNLP files designed for local testing.
- *Origamist.jar* The main Origamist archive.
- *OrigamistEditor.jar* The Origamist Editor additional archive.
- *OrigamistViewer.jar* The Origamist Viewer additional archive.

- *preview.html* An HTML page running Origamist Viewer as a webpage applet.
- *preview-editor.html* An HTML page running Origamist Editor as a webpage applet.
- *README* Instructions on how to run Origamist applications from this CD.
- *signapplet.bat.example, signapplet.sh.example* Templates of shell scripts that digitally sign JAR files.