Folding Sheets for a Modular Origami Dodecalendar

Richard Hirsch
richard.hirsch at gmx dot net

Abstract
Twelve square sheets of paper can be folded in such a way that they can be assembled to a pentagon dodecahedron (*origami)*. The single units are called modules, hence the name *modular*. If the sheets bear calendrical information at the right places, the dodecahedron shows the calendar for each month on its faces: the *dodecalendar*.
In this article we let MetaPost calculate piece by piece the information that needs to be printed on the module paper to enable us to fold the modules and assemble the dodecahedron.

Keywords
MetaPost, tutorial

Introduction

MetaPost

MetaPost is a programming language and an interpreter that produces PostScript output. It is well suited to produce technical drawings and – being derived from Metafont with basically the same capabilities and just a few extensions – suggests itself to illustrate TEX documents. As we will see, MetaPost provides a very natural way of describing relations of points in the plane (3D extensions do exist also).

However, the richness of features and concepts makes it hard for the beginner to start. The user's manual from John Hobby [1] and the Metafontbook from Donald Knuth [2] want to be read and for the fine touch even a glimpse into the source code (`plain.mp`) may be necessary.

Examples, however, are a good way to start with. There are excellent sources in the Internet (c.f. `http://www.tug.org/metapost.html` for an extensive list), and I hope, the origami dodecalendar will be another one that might attract you to MetaPost.

Dodecalendar

At `http://www.origami-cdo.it/modelli/` instructions for folding a 12-piece modular origami dodecalendar can be found. Twelve square paper sheets are folded into twelve modules that can be assembled to a pentagon dodecahedron. If the paper is properly printed, each face shows a calendar for one month, see figure 1.

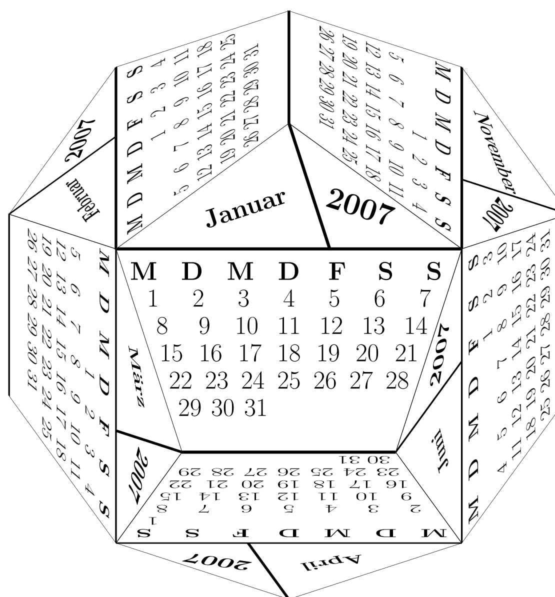


Figure 1: the modular origami dodecalendar

Aim of this Article

We want to print those sheets on our own and use MetaPost for this purpose. Traditional origami doesn't allow preprinted guides that show where the creases must go, but since we have to print the calendars anyway, we can as well print those marks – not for cheating, of course, but to avoid ugly creases on the faces of our dodecalendar.

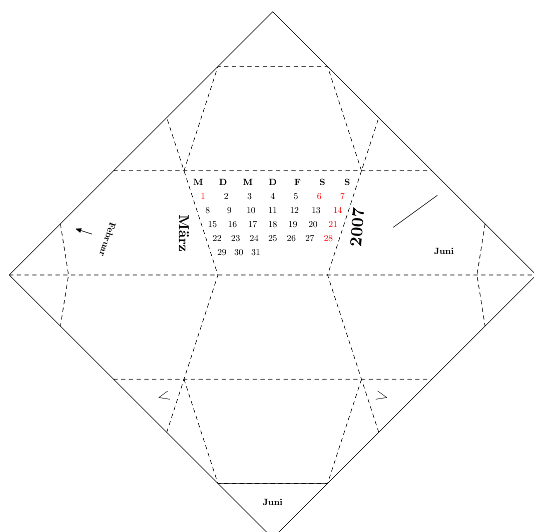


Figure 2: folding sheet with creases shown for January 2007 (sic!, see figure 1)

Folding Sheets

Figure 2 shows how the preprinted sheets of paper should look like. We number the relevant points as shown in figure 3.

A good start for calculating all those points seems to be the base of the regular pentagon highlighted in figure 3. But first of all we have to perform some setup.

Setup

We are going to calculate the positions of the points shown in figure 3. Usually this is done in Metafont or MetaPost by collecting the x- and y-parts of the coordinate pairs in the arrays x and y and adress them as points z. We will use coordinate pairs P1 through P20 instead and won't bother with their components at all. This may be mainly a matter of taste, but there is at least one rationale too: For the dodecalendar we need twelve paper sheets for the modules, and for each of them the positions of the points in figure 3 must be known. Now MetaPost clears the contents of the coordinate pairs z for every new figure; thus all the calculations would have to be done over and over again. The positions in container P however, are valid through the whole MetaPost run. Therefore, first thing for us to do is to declare the container P for the coordinate pairs:

```
pair P[];       % positions of the points
```

(As you already may have guessed or known, just like in TEX everything after a %-sign in the MetaPost source is ignored by the interpreter.)

Furthermore we have to give MetaPost some information about the size of our figure. Therefore we define a basic unit of measurement for our drawing: the radius R of the escribed circle of the central pentagon (see figure 4) is set to be 2 cm:

```
R:= 2cm;        % scaling factor
```

The real MetaPost way to deal with the absolute size of the figure would be to tell MetaPost only something like `P2 + 11cm*right = P3` and let it figure out the actual value of R itself. Unfortunately MetaPost can not perform transformations unless all components of the transformation matrix are known. Since we want to make heavy use of transformations, it is better for us to fix the scale of the figure at this time.

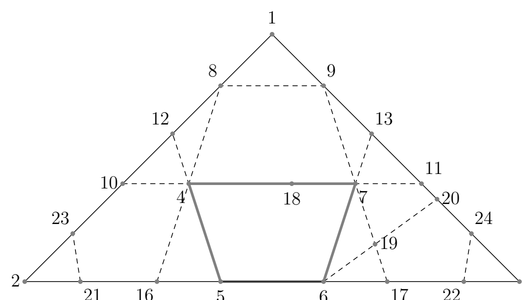
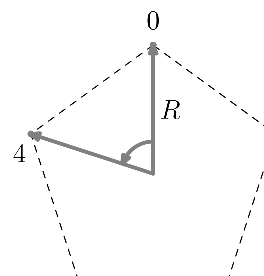
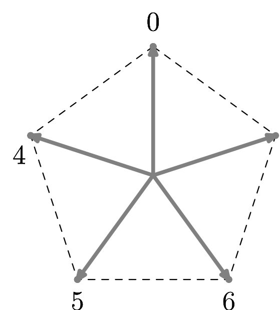


Figure 3: numbering of important points



(a) turning P_0 by 72°



(b) all vertices of the pentagon

Figure 4: central pentagon

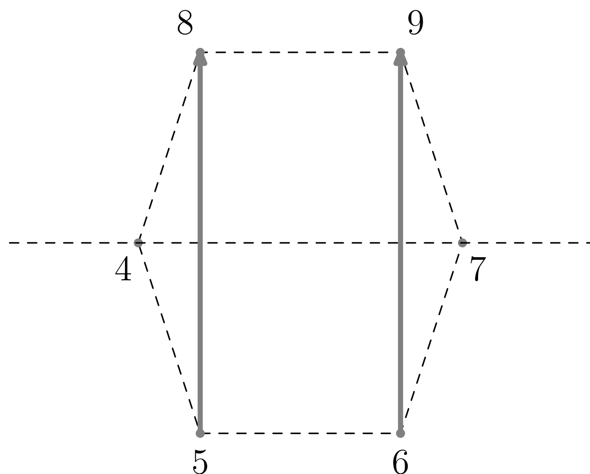
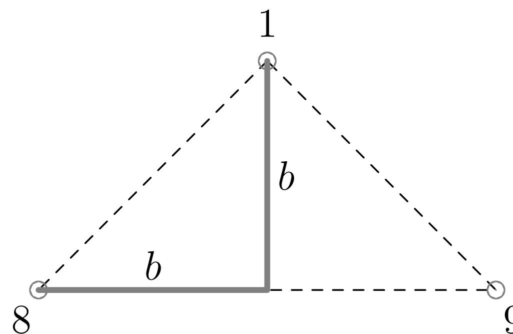
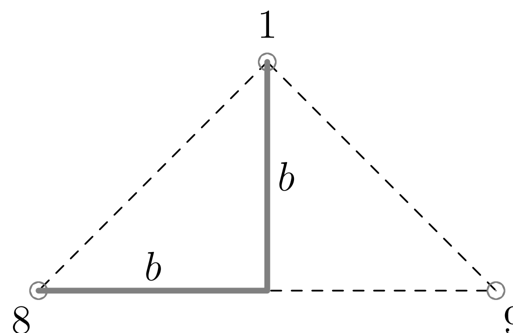


Figure 5: points P_8 and P_9



(a) Calculation by vector addition



(b) Calculation by means of the law of Pythagoras

Figure 6: P_1, the top corner of the paper sheet

Regular Pentagon

First of all, we introduce point P_0 at the top of the pentagon to complete it. We set the origin of our coordinate system to the center of the pentagon and then we can define the position of P_0 simply as

```
P0 = R*up;
```

The vector up is predefined in MetaPost as "up=-down=(0,1)".

Now, by means of MetaPost's rotation command, the position of the other vertices can be calculated as easily as

```
for i:= 1 upto 4:
  P[i+3]:= P0 rotated (i*360/5);
endfor
```

Mirroring of the Pentagon's Base

Now that MetaPost knows the positions of P_4 to P_7, we can define points P_8 and P_9: We get P_8 if we reflect P_5 about line $\overline{P_4P_7}$ and P_9 by doing the same with P_6. MetaPost understands this immediately:

```
for i:= 8,9:
  P[i]:= P[i-3] reflectedabout (P4, P7);
endfor
```

The Top Corner

The points P_8 and P_9 are of special interest since they lie on the edges of our folding sheet; until now all our drawing was completely independent of the actual size of the paper. But now we are going to deal with the boundaries of our sheet of paper; we start with the top (P_1).

We observe that the triangle $\triangle P_8P_9P_1$ is the top half of a square (see figure 2. We know the length of its base since we have already P_8 and P_9;we call it a and let $b = a/2$ (see figure 6a).

```
a:= length(P9-P8);
b:= .5a;
```

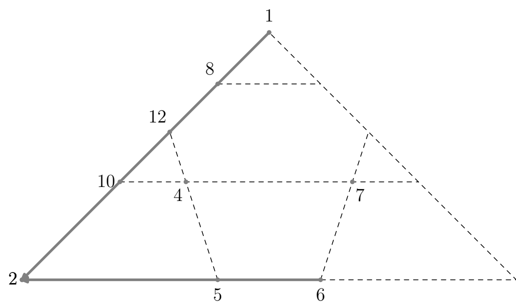
Now MetaPost can find P_1 simply by vector addition

```
P1:= P8 + b*(right + up);
```

(Alternatively we could have made use of the Pythagorean theorem (and introduce MetaPost's `dir` command and its ++-operator): We know that the angle $\angle P_1P_8P_9$ is half of a right angle (i.e. 45°). The length of the hypotenuse c is $\sqrt{b^2+b^2}$ (see figure 6b) and this root is just what the ++-operator calculates. So shifting P_8 by $b{+}{+}b$ in northeast direction yields P_1 – or in MetaPost's terms: `P1:= P8 shifted ((b++b)*dir 45).`)

Left Edge

Next we want to deal with the points on the left edge, P_2, P_{10} and P_{12}. Point P_2 lies somewhere on the line through points P_5 and P_6 (see figure 7) and also somewhere on the line going through P_1 and P_8. If we



(a) Point P_2 as intersection point

Figure 7: points on the left and right edge

want to tell MetaPost about these facts we can use the volatile numeric variable *whatever* for "somewhere" and the expression [p,q] for "on the line $\overline{p\,q}$". So we write:

```
P2  = whatever[P1,P8] = whatever[P5,P6];
```

This is all information MetaPost needs in order to know where point P_2 lies.

Please note, that we didn't use the := assignment, but instead add two equations to MetaPost's internal system of linear equations with =. This system is solved if we use P_2 in a drawing (or an immediate assignment with :=).

Points P_{10} and P_{12} are computed likewise:

```
P10 = whatever[P1,P8] = whatever[P7,P4];
P12 = whatever[P1,P8] = whatever[P5,P4];
```

Right edge

The positions of points P_3, P_{11} and P_{13} could be found the same way, but we don't want the MAPS to become dull.

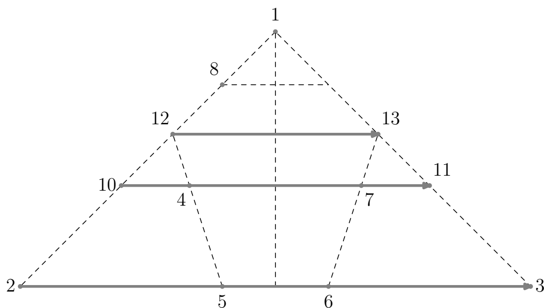


Figure 8: The points on the right edge

Instead we use another important feature of MetaPost: transformations. In figure 8 we note that point P_3 is just P_2 reflected about line $\overline{P_0P_1}$. In MetaPost we can express that as P3 = P2 reflectedabout (P0,P1) but since we need to transform more than one variable the same way, we'd rather have an own transformation for this purpose. We can have that in MetaPost by defining

```
transform flipright;
flipright:= identity reflectedabout(P0,P1);
```

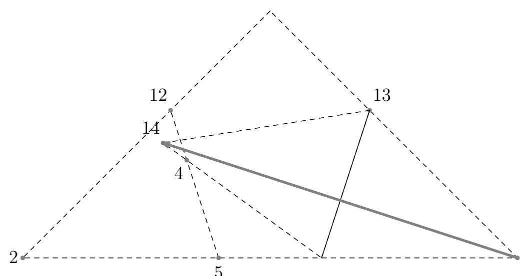
Now we could say P3 = P2 transformed flipright, but we go one step further and define the new macro flippedright:

```
def flippedright=transformed flipright enddef;
```

That enables us to write simply:

```
P3 = P2 flippedright;
P11 = P10 flippedright;
P13 = P12 flippedright;
```

Angle Marks



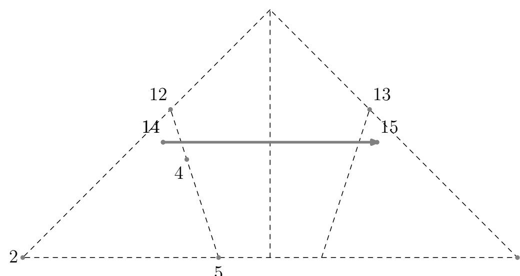
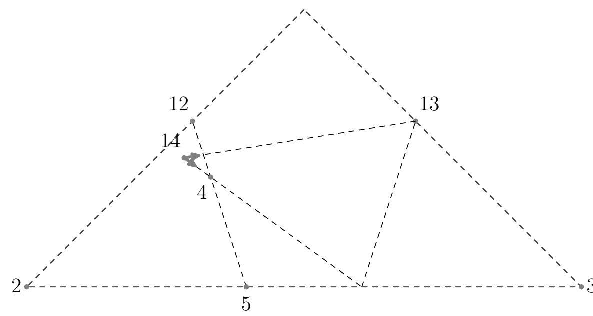


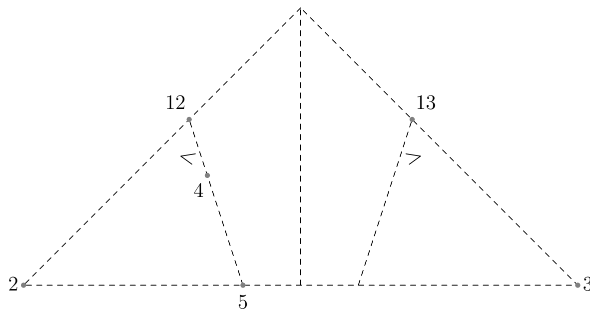
Figure 9: Points P_{14} and P_{15}

In the process of building a module, the corners P_2 and P_3 must be folded along $\overline{P_5P_{12}}$ and $\overline{P_6P_{13}}$ respectively. We must get the creases by having the angle marks at P_{14} and P_{15} preprinted on the sheets (see figure 9); this way P_2 can be folded to P_{15} and P_3 to P_{14} and the creases come up. Since we know already about transformations, defining P_{14} and P_{15} is too easy:

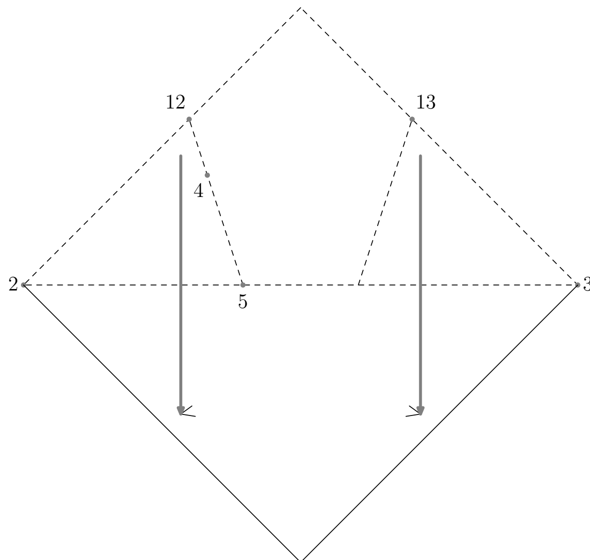
```
P14 = P3 reflectedabout (P6, P13);
P15 = P14 flippedright;
```



(a) the command `unitvector`



(b) the guides for the corners



(c) the guides on the lower half of the paper

Figure 10: Angles as guides for the corners

We don't want just points, however, but nice little angles of a certain length (`ticklength` say), into which the corners fit smugly (see figure 10). For this purpose we use MetaPost's `unitvector` command which reduces a given vector to length 1, but maintains its direction. With this tool we can build vectors with arbitrary length in a given direction, just by multiplying it with the proper dimension (see figure 10a). For the calculation of the endpoints of the angle's rays we define a *vardef* macro, which returns its last expression (like a function in certain other programming languages):

```
ticklength:= 0.1a;
vardef tkend(expr p, q) =
  p + ticklength*unitvector(q-p)
enddef;
```

The two rays of each angle are obtained by connecting P_{14} and the endpoints with the `--`-operator. It produces a path where the points are connected by straight lines. We can store these paths in suitable variables (of type `path`) and collect them in a container (`Line[]`). For the angle at point P_{15} we make use of the fact that transformations work on paths as well as on points:

```
path Line[];
Line14:= tkend(P14,P6)--P14--tkend(P14,P13);
Line15:= Line14 flippedright;
```

Finally we want the guidelines in the lower half of the paper. So we apply a final transformation. (Here the `:=` operator is mandatory, because the equation with = would be inconsistent.)

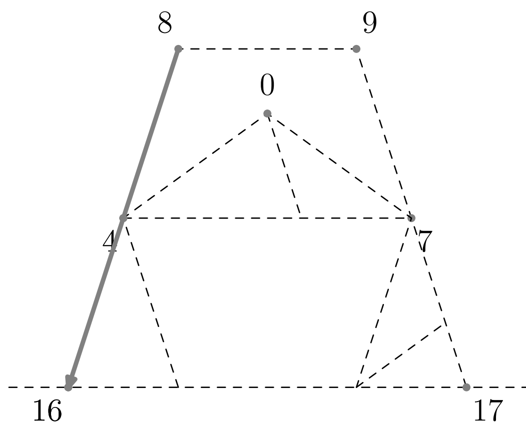
```
transform flipdown;
flipdown:= identity reflectedabout (P2,P3);
def flippeddown = transformed flipdown enddef;

for p:= 14,15:
  Line[p]:= Line[p] flippeddown;
endfor
```

Boundaries for Calendrical Information

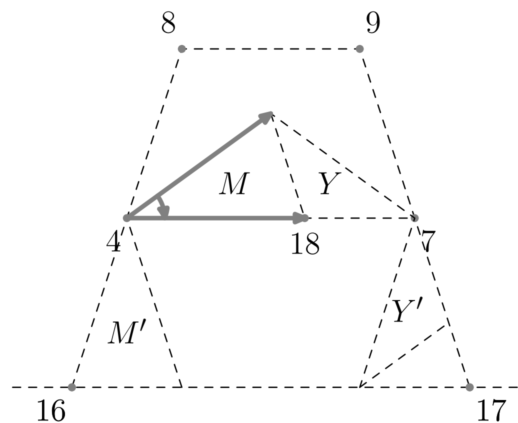
In order to print the calendar, the month and year information at the correct places, we need to calculate the positions of points P_{16} to P_{19} (see figure 3).

Since we obtained P_8 by reflecting P_5 about $\overline{P_4P_7}$ (see 'Mirroring of the Pentagon's Base'), we know from the interception theorems that the vector $P_8 - P_4$ is the same as $P_4 - P_{16}$ (see figure 12a). The position of P_{17} is again found by reflection (figure 11b).

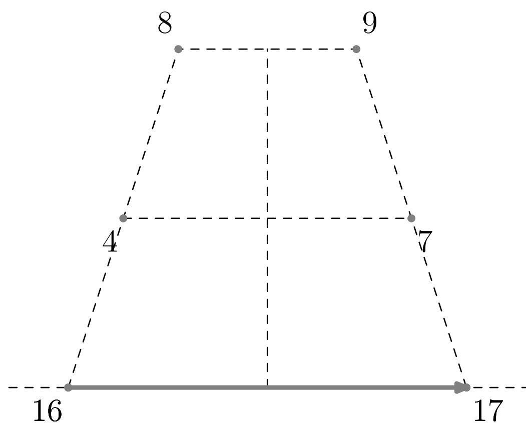
```
P16:= 2[P8,P4];
P17:= P16 flippedright;
```



(a) position of P_{17}

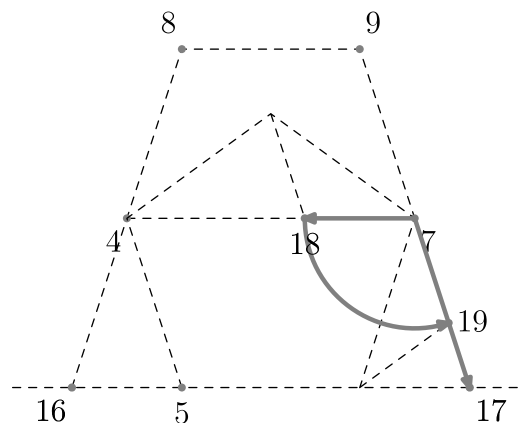


(a) position of P_{18}



(b) position of P_{16}

Figure 11: boundaries for calendrical information



(b) position of P_{19}

Figure 12: boundary of the year-field

In order to determine the position of P_{19} we look at P_{18} first: The triangle M is isosceles, so $|P_0 - P_4| = |P_{18} - P_4|$ and we can obtain the position of P_{18} by rotating P_0 around P_4:

```
w:= angle(P0-P4) - angle(P7-P4);
P18 = P0 rotatedaround (P4, -w);
```

(Yes, the expression `angle(P7-P4)` is indeed 0 and could have been left out, and yes, the use of `unitvector` may have been appropriate here too; indeed, we are going to return to using `unitvector` soon when calculating P_{19}.)

Now, that we know the position of P_{18} we can focus again on P_{19}. The line $\overline{P_0P_{18}}$ divides the triangle $\triangle P_4P_7P_0$ into two smaller triangles M and Y that contain the information about the month and the year respectively (see figure 12a). We observe that the triangles Y and Y′ are identical, only that the latter falls onto another face of the dodecahedron. Thus $|P_7 - P_{18}| = |P_7 - P_{19}|$ and we get the position of P_{19} by applying the universal example of constructing a vector by multiplying its length with the unitvector of its direction:

```
P19 = P7 + unitvector(P17-P7)*length(P7-P18);
```

Guideline for aligning two modules

When all the modules have been folded, every two of them at a time are to be combined to a double module. To get the correct angle between the faces of the dodecahedron, the two modules must be aligned along line $\overline{P_6P_{20}}$. (As it happens, this is just the bisector of angle $\angle P_6P_3P_7$, but we pretend not to know about that.) Since we know already about the position of P_{19}, we can apply the same technique as in 'Left Edge' and have

```
P20 = whatever[P1,P3] = whatever[P6,P19];
```

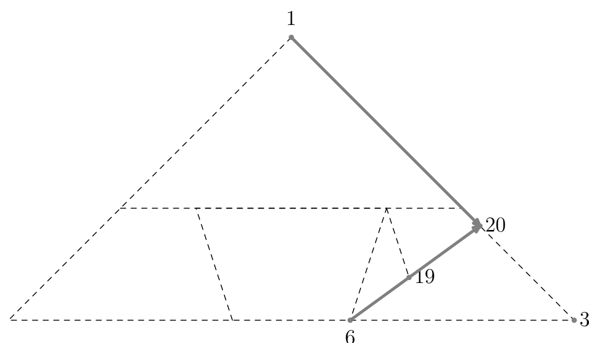


Figure 13: The guideline for aligning two modules

On the final sheets we don't want the whole line to show up; some part (20 % say) at the beginning and the end should be left out (see figure 2). We find the new starting point 20 % from P_6 on the line $\overline{P_6P_{20}}$. Sounds familiar? Yes, see section 'Mirroring of the Pentagon's Base' – we can apply the [p,q]-expression again and store the resulting path in our `Line[]` container.

```
Line[20]:= .2[P6,P20]--.8[P5,P20];
```

Drawing the folding sheets

Now MetaPost knows about all the relevant points and we can let it draw the folding sheets. The result, together with some labels, is presented in figure 14. Alas, the calendar is still missing (see below).

```
beginfig(1);
  draw P1--P2--P1 flippeddown--P3--cycle;
  draw P8--P9;
  draw Line[20];
  for l:= 14, 15: draw Line[l]; endfor
endfig;
end.
```

Summary and Outlook

For now we have learned how to make use of some important features of MetaPost, among them

- addition of vectors,
- affine transformations like *shifting*, *rotation*, and *reflection*,
- solving linear equations (the *whatever*-statement),
- and last but not least how to draw *straight lines*.

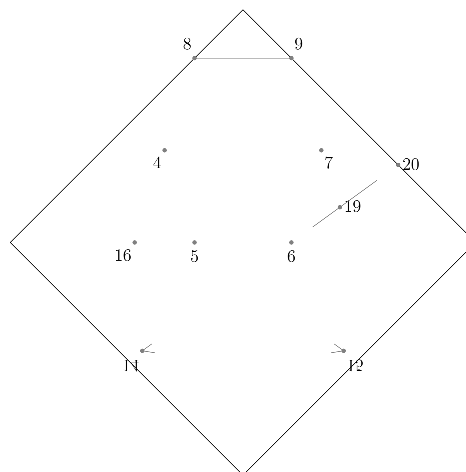


Figure 14: the printed information (with labels)

In the next issue of the MAPS we will have a look on MetaPost's algebraic capabilities and the possibilities to add labels and TEX content to the picture when we are going to make MetaPost calculate and draw the calendrical information for the dodecalendar too.

In the meantime you can prepare a set of twelve folding sheets and assemble a dodecahedron as shown at the above mentioned web page. (If you don't know how to run MetaPost on your system, you can try the MetaPost Previewer at `http://www.tlhiv.org/MetaPostPreviewer`, just put the code without "`beginfig(1);`", "`endfig;`" and "`end.`" into the textfield and press the preview button.)

Have fun!

References

[1] John D. Hobby. A User's Manual for MetaPost. Technical Report 162, AT&T Bell Laboratories, Murray Hill, New Jersey, April 1992. Also available at `http://www.tug.org/docs/metapost/mpman.pdf`.

[2] D. E. Knuth. *Computers and Typesetting*, volume C. Addison Wesley, Reading, Massachusetts, 1986.

Richard Hirsch
richard.hirsch at gmx dot net