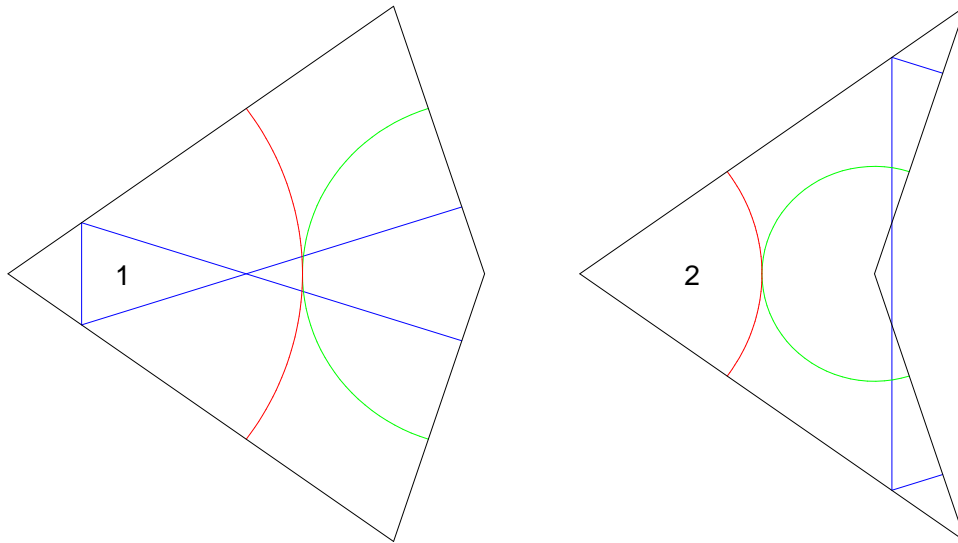


Penrose Kacheln

Michael Nüsken, Paderborn, 31.01.2003

Die beiden Penrose Kacheln Kite (engl. (Spielzeug-)Drachen, im Bild Nr. 1) und Dart (engl. Pfeil, im Bild Nr. 2) sind ganz simple Polygone. (Wir könnten sie auch Keil und Pfeil nennen, aber wir wollen hier bei den durch Penrose gewählten englischstämmigen Namen bleiben.)



Sie dürfen aneinandergesetzt werden wie Puzzleteile oder Dominosteine, wobei immer Kanten gleicher Länge aneinanderstossen und die roten und grünen Linien aneinanderpassen müssen. Das eigentlich schon alles. Verblüffend ist, was dabei herauskommt. Aber seht selbst...

```
> restart;
  `mod` := modp:
  changecoords := 'changecoords': with(plots):
  arrow := 'arrow': with(plottools):
```

Noch ein paar Farben für später.

```
> macro( pink=COLOR( HUE, 5/6 ) ):
  macro( lightgray=COLOR( RGB, .9,.9,.9 ) ):
```

Hier habe ich mich bemüht, in Maple eine einigermaßen einfache Möglichkeit zu schaffen, diese Kacheln aneinanderzulegen. Weiter unten erkläre ich einige Funktionen des `module` `penrose`.

```
> penrose := module()
  local z, ζ, Φ, complex2point, plotcomplexpolygon, num, plane, data, activetile, P, Pdot, A, B, i,
    smove, smoveinv, rels, fmove, dflcompose, dfl;
  export ζ, φ, wantradical, compose, longshift, shortshift, specialmove, move, composemoves,
    reset, redraw, tryabs, doabs0, doabs, undo, atrel, tryrel, dorel0, dorel, attr, atdo, deflate,
    completeaces;
    wantradical := false;
    ζ := exp(2*π*I/ 10);
    φ := (1 + sqrt(5))/ 2;
    complex2point := z → [ℜ(z), ℑ(z)];
    plotcomplexpolygon :=
      proc(L) polygonplot(map(complex2point, L), args[2 .. -1]) end proc;
  A := 1 / (2*ζ);
```

```

B :=  $\phi^2 + \phi \cdot \zeta^3 / 4$ ;
P[kite] := display(plotcomplexpolygon([0,  $\phi^2 \cdot \zeta^{-1}$ ),  $\phi^2$ ,  $\phi^2 \cdot \zeta$ ],
  arc(complex2point(0),  $\phi$ ,  $-\pi / 5 .. \pi / 5$ , color = red, thickness = 2),
  arc(complex2point( $\phi^2$ ), 1,  $3\pi / 5 .. 7\pi / 5$ , color = green, thickness = 2), curve([
  complex2point(B), complex2point(A), complex2point(conjugate(A)),
  complex2point(conjugate(B))], color = blue), scaling = constrained, axes = none);
A :=  $\phi^2 / \zeta - 1 / (2 \cdot \zeta)$ ;
B :=  $\phi^2 / \zeta + \phi \cdot \zeta^3 / 4$ ;
P[dart] := display(plotcomplexpolygon([0,  $\phi^2 \cdot \zeta^{-1}$ ),  $\phi$ ,  $\phi^2 \cdot \zeta$ ],
  arc(complex2point(0), 1,  $-\pi / 5 .. \pi / 5$ , color = red, thickness = 2),
  arc(complex2point( $\phi$ ),  $1 / \phi$ ,  $2\pi / 5 .. 8\pi / 5$ , color = green, thickness = 2), curve([
  complex2point(B), complex2point(A), complex2point(conjugate(A)),
  complex2point(conjugate(B))], color = blue), scaling = constrained, axes = none);
Pdot := disk(complex2point( $1 / \phi$ ),  $5 \cdot (1 / \phi - \Re(1 / (2 \cdot \zeta))) / 6$ , color = blue);
smove := proc(i, p0, p1, p2, p3, p4)
   $\zeta^{(i \bmod 10)} \cdot z + \text{add}(args[2 + j] \cdot \zeta^j, j = 0 .. nargs - 2) \cdot \Phi^2$ 
end proc;
smoveinv := proc(X)
  local A, B, j;
  B := collect(X, [z,  $\Phi$ ]);
  userinfo(2, penrose, B = collect(simplify(B, rels), [z,  $\Phi$ ]));
  A := coeff(B, z);
  if A = 1 then A := 0
  elif A =  $\zeta$  then A := 1
  elif A = -1 then A := 5
  elif type(A, '^') then A := op(2, A)
  else error "Huch: %1.", A
  end if;
  B := simplify(coeff(B,  $\Phi^2$ ), { $\zeta^4 - \zeta^3 + \zeta^2 - \zeta + 1$ });
  [A, seq(coeff(B,  $\zeta$ , j), j = 0 .. 3)]
end proc;
rels := { $\Phi^2 - \Phi - 1$ ,  $\zeta^4 - \zeta^3 + \zeta^2 - \zeta + 1$ };
compose := proc()
  local p, f, A, B;
  p := z;
  for f in args do
    p := simplify(expand(subs(z = smove(op(f)), p) *  $\zeta^{100}$ ), { $\zeta^{10} - 1$ })
  end do;
  smoveinv(p)
end proc;
move := proc(i, p0, p1, p2, p3, p4)
  if type(i, list) then return procname(op(i), args[2 .. -1]) end if;
   $\zeta^{(i \bmod 10)} \cdot (x + I \cdot y) + \text{add}(args[2 + j] \cdot \zeta^j, j = 0 .. nargs - 2) \cdot \phi^2$ ;
  expand(%);
  complex2point(%);
  (simplify(%) assuming real);
end proc;

```

```

        if wanradical then convert(% , radical) end if;
        unapply(% , x, y)
    end proc;
fmove :=
    proc(i, p0, p1, p2, p3, p4) move(args); map(evalf, %(x, y)); unapply(% , x, y) end proc;
composemoves := proc(f)
    local g, i;
        g := f;
        for i from 2 to nargs do g := ((eval(g))@op)@(args[i]) end do;
        g(x, y);
        expand(%);
        (simplify(% ) assuming real);
        combine(% , trig);
        if wanradical then convert(% , radical) end if;
        map(collect, % , [x, y]);
        unapply(% , x, y)
    end proc;
reset := proc() num := 0; activetile := [0, kite, [0]]; plane := NULL; data := NULL end proc;
redraw := proc() display(plane, scaling = constrained, axes = none) end proc;
tryabs := proc(tile, mv)
    display(display(transform(move(op(mv)))(P[tile]), color = black, args[3 .. -1 ]),
    display(transform(move(op(activetile[3 ]))(Pdot)), plane, scaling = constrained,
    axes = none)
    end proc;
doabs0 := proc(tile, mv)
    local new;
        num := num + 1;
        new := display(P[tile], textplot(
            [op(evalf(complex2point(1 / phi))), sprintf("%d", num)], color = black),
            args[3 .. -1 ]);
        plane := transform(fmove(op(mv)))(new), plane;
        data := [num, args], data;
        activetile := [num, args];
        NULL
    end proc;
doabs := proc(tile, mv) doabs0(args); redraw( ) end proc;
undo := proc(n)
    local a, b, j;
        if nargs = 0 then return procname(activetile[1 ]) end if;
        for j to nops([data]) do
            if n = [data][j][1 ] then
                userinfo(2, penrose, 'removing ', [data][j],
                    'if'(n = activetile[1 ], 'WARNING: last position changes!', '');
                data := op(subsop(j = NULL, [data]));
                plane := op(subsop(j = NULL, [plane]));
                break
            end if
        end do
    end proc

```

```

        end if
    end do;
    if nops([data]) = 0 then reset( )
    elif n = activetile[1] then activetile := [data][1]
    end if;
    NULL
end proc;
atrel := proc(n)
    local j;
    if nargs = 0 and 0 < nops([data]) then activetile := [data][1]
    else for j to nops([data]) do
        if n = [data][j][1] then activetile := [data][j]; break end if
    end do
    end if;
    NULL
end proc;
tryrel := proc(tile, mv)
    local mv1;
    if mv in { greenleft, greenright, redleft, redright } then
        mv1 := specialmove[mv, tile, activetile[2]]
    else mv1 := mv
    end if;
    tryabs(tile, compose(activetile[3], mv1), args[3 .. -1])
end proc;
dorel0 := proc(tile, mv)
    local mv1;
    if mv in { greenleft, greenright, redleft, redright } then
        mv1 := specialmove[mv, tile, activetile[2]]
    else mv1 := mv
    end if;
    doabs0(tile, compose(activetile[3], mv1), args[3 .. -1])
end proc;
dorel := proc(tile, mv)
    local mv1;
    if mv in { greenleft, greenright, redleft, redright } then
        mv1 := specialmove[mv, tile, activetile[2]]
    else mv1 := mv
    end if;
    doabs(tile, compose(activetile[3], mv1), args[3 .. -1])
end proc;
attry := proc(n, tile, mv) atrel(n); tryrel(args[2 .. -1]) end proc;
atdo := proc(n, tile, mv) atrel(n); dorel(args[2 .. -1]) end proc;
dfcompose := proc()
    local p, f, A, B;
    p := z;
    for f in args do

```

```

    p := simplify(expand(subs(z = smove(op(f)), p)*ζ^100), {ζ^10 - 1})
end do;
p := simplify(subs(Φ^2 = Φ^2*(ζ + ζ^9), p), {ζ^10 - 1});
smoveinv(p)
end proc;
dfl[kite] := proc(n, tile, mv)
    [kite, dflcompose(mv, [3, op(2 .. -1, longshift[9])]), args[4 .. -1]],
    [kite, dflcompose(mv, [-3, op(2 .. -1, longshift[1])]), args[4 .. -1]],
    [dart, dflcompose(mv, [1]), args[4 .. -1]],
    [dart, dflcompose(mv, [-1]), args[4 .. -1]]
end proc;
dfl[dart] := proc(n, tile, mv)
    [kite, dflcompose(mv), args[4 .. -1]],
    [dart, dflcompose(mv, [4, op(2 .. -1, longshift[9])]), args[4 .. -1]],
    [dart, dflcompose(mv, [-4, op(2 .. -1, longshift[1])]), args[4 .. -1]]
end proc;
deflate := proc()
    local newdata, oldplane;
    ListTools[Reverse]([data]);
    map(x → dfl[x[2]](op(x), %));
    newdata :=
        ListTools[MakeUnique](%, 1, (x, y) → evalb(x[1] = y[1] and x[2] = y[2]));
    reset();
    map(doabs0@op, newdata);
    redraw()
end proc;
completeaces := proc()
    local allkites, alldarts, X, mv;
    allkites, alldarts := selectremove(x → x[2] = kite, [data]);
    allkites := map(compose, map2(op, 3, allkites));
    for X in alldarts do
        mv := compose(X[3], specialmove[greenleft, kite, dart]);
        if not mv in allkites then doabs0(kite, mv, op(X[4 .. -1]), args) end if;
        mv := compose(X[3], specialmove[greenright, kite, dart]);
        if not mv in allkites then doabs0(kite, mv, op(X[4 .. -1]), args) end if
    end do;
    redraw()
end proc;
reset();
for i from 0 to 9 do
    longshift[i] := compose([i], [0, 1, 0, 0, 0], [-i]);
    shortshift[i] := compose([i], [0, 0, 0, 1, -1], [-i])
end do;
specialmove := table([
    (greenleft, dart, dart) = 'ERROR("Dart to dart can never be at green line")',
    (greenright, dart, dart) = 'ERROR("Dart to dart can never be at green line")',

```

```

(redleft, dart, dart) = compose([2]), (redright, dart, dart) = compose([-2]),
(greenleft, kite, dart) = compose(shortshift[0], longshift[0], [4]),
(greenright, kite, dart) = compose(shortshift[0], longshift[0], [6]),
(redleft, kite, dart) = compose(longshift[1], [5]),
(redright, kite, dart) = compose(longshift[9], [5]),
(greenleft, dart, kite) = compose([1], longshift[0], shortshift[0], [5]),
(greenright, dart, kite) = compose([-1], longshift[0], shortshift[0], [5]),
(redleft, dart, kite) = compose(longshift[1], [5]),
(redright, dart, kite) = compose(longshift[9], [5]),
(greenleft, kite, kite) = compose(shortshift[3], 2*longshift[0], [6]),
(greenright, kite, kite) = compose(shortshift[7], 2*longshift[0], [4]),
(redleft, kite, kite) = compose([2]), (redright, kite, kite) = compose([-2]))

```

end module

Jetzt laden wir den Modul und machen damit alle seine Exporte verfügbar.

```

> with(penrose);
  # kernelopts( opaquemodules=false );

```

Die Zeile, in der `opaquemodules` verändert wird, ist gut fürs Entwanzen (Debuggen), sie ermöglicht den Zugriff auf die lokalen Daten der Module.

```

reset(): doabs0( kite, [0] ): doabs( dart, [0,1.2] );

```

Diese Prozedur erlaubt uns später selektiv die blauen, roten oder grünen Linien und die Texte wieder zu entfernen.

```

> removecurves:=proc(c::set, p::specfunc(anything, PLOT))
  local C;
  C:=map( 'plot/color', c minus {TEXT} );
  p;
  if TEXT in c then
    subs( TEXT=NULL, % );
  end if;
  eval(subs(
    CURVES=proc(x,y)
      if y in C
      then NULL
      else CURVES(args)
      end if;
    end proc,
    % ));
end proc:

```

[*atdo, atrel, attry, completeaces, compose, composemoves, deflate, doabs, doabs0, dorel, dorel0, longshift, move, ϕ , redraw, reset, shortshift, specialmove, tryabs, tryrel, undo, wantradicall, ζ*]

Zurück zum Wesentlichen.

Zuerst soll eine wichtige Möglichkeit überprüft werden. Es werden einige Bilder erstellt, in denen jeweils eine Kachel festgelegt und ein weiteres, schwarz gezeichnetes probeweise angelegt ist. Wir werden gleich genauer auf Einzelheiten eingehen. Seht Euch zuerst einmal die 14 Bilder an, die alle Möglichkeiten darstellen, wie Kites und Darts aneinanderpassen.

```

> L := NULL;
  for f in [dart, kite] do
    reset();
    doabs(f, [rand(-5 .. 5)( )]);
  end for;

```

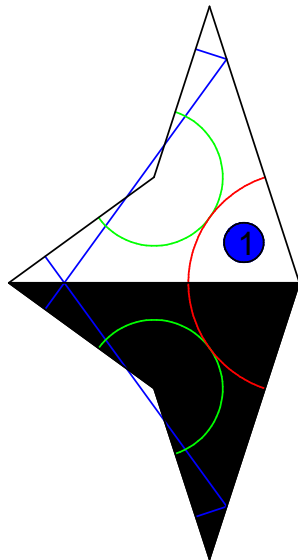
```

for t in [dart, kite] do for X in [greenleft, greenright, redleft, redright] do
  try L := L, tryrel(t, X, title = sprintf("A %s with a trial %s at %a.", f, t, X))
  catch :
  end try
end do
end do
end do;
display(L, insequence = true, scaling = constrained)

```

Der blaue Punkt zeigt immer die Kachel an, an die wir gerade anlegen wollen. Vor dort aus sollten wir auch schauen, wenn wir entscheiden, was links und rechts ist. Es ist für später nützlich, sich da jetzt klarzumachen. (Am besten führt man die obigen Programmanweisungen zwei- oder dreimal aus. Das erste Kite oder Dart wird jedesmal ein wenig anders hingelegt, so hat man Gelegenheit zum Üben.)

A dart with a trial dart at redleft.



[>

+ **Bewegungen**

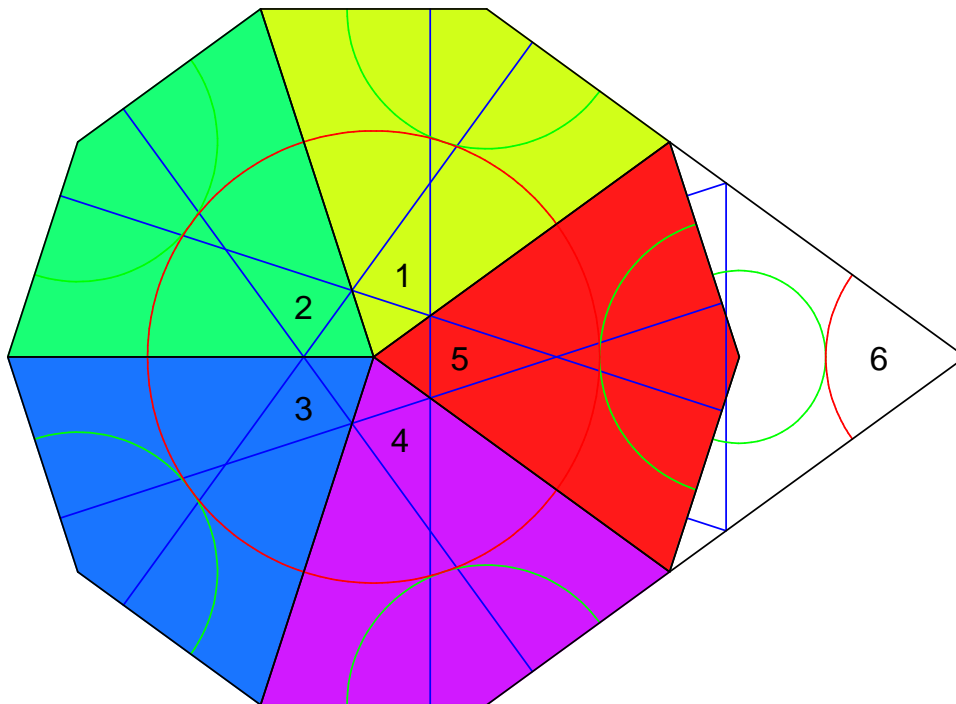
Und jetzt fangen wir richtig an.

Als erstes machen wir mit der Prozedur `reset` die Ebene leer.

Dann legen wir fünf Kites aus. Dazu verwenden wir das Kommando `dorel`, welches relativ zur aktiven Kachel eine neue auslegt. Das erste Argument gibt an, welches Kachel verwendet werden soll, ein `kite` oder ein `dart`. Das zweite Argument gibt eine Bewegung an. Das Kommando `dorel` erlaubt hier die Angabe eines der vier Schlüsselwörter `redleft`, `redright`, `greenleft` und `greenright`. Die Bewegung wird dann so bestimmt, dass die grüne bzw. rote Linie nach links bzw. rechts fortgesetzt wird. Ausserdem können wir auch gleich Farben mit angeben, es geht aber auch ohne.

Zum Schluss legen wir noch ein Kite aus. Allerdings verwenden wir diesmal das Kommando `doabs`, welches eine Kachel absolut plziert; es wird also von seiner Startposition (mit der Spitze am Ursprung in positive x-Richtung blickend) aus bewegt. Die Argumente haben ansonsten dieselbe Bedeutung wie bei `dorel`. (Die Schlüsselwörter können hier natürlich nicht verwendet werden.)

```
> reset():
  dorel( kite, redleft, color=COLOR(HUE,1/5) ):
  dorel( kite, redleft, color=COLOR(HUE,2/5) ):
  dorel( kite, redleft, color=COLOR(HUE,3/5) ):
  dorel( kite, redleft, color=COLOR(HUE,4/5) ):
  dorel( kite, redleft, color=COLOR(HUE,0) ):
  doabs( dart, [5,1,0,1,-1] );
```



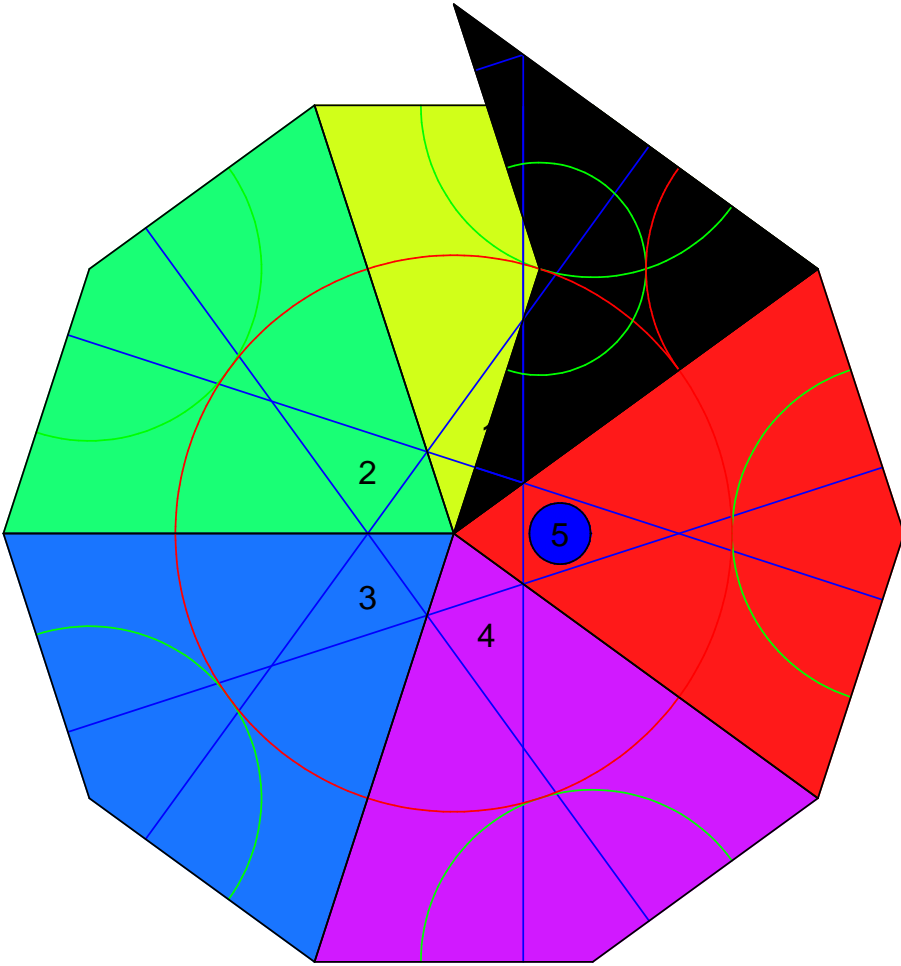
Das eben angelegte Kite passt da offenbar nicht hin. Wir möchten es wieder entfernen. Dazu gibt es das Kommando `undo`. Sein einziges Argument ist die Nummer der Kachel, welche wir entfernen wollen. (Wenn wir keine Nummer angeben, wird einfach die aktuelle Kachel, also normalerweise die zuletzt gelegte, entfernt.)

```
> infolevel[penrose]:=2:
  undo(6);
  infolevel[penrose]:=0:
undo: removing [6, dart, [5, 1, 0, 1, -1]] WARNING: last position changed!
```

Anstatt ständig mit `undo` zu arbeiten, ist es viel einfacher, die Kachel ersteinmal probeweise anzulegen. Dazu gibt es die Kommandos `tryrel` und `tryabs`. Sie haben dieselben Argumente wie ihre Namensvettern `dorel` und `doabs`. Das Probesteil wird normalerweise schwarz gezeichnet. Der blaue Punkt kennzeichnet die aktuelle Kachel bezüglich dessen die nächste plaziert wird.

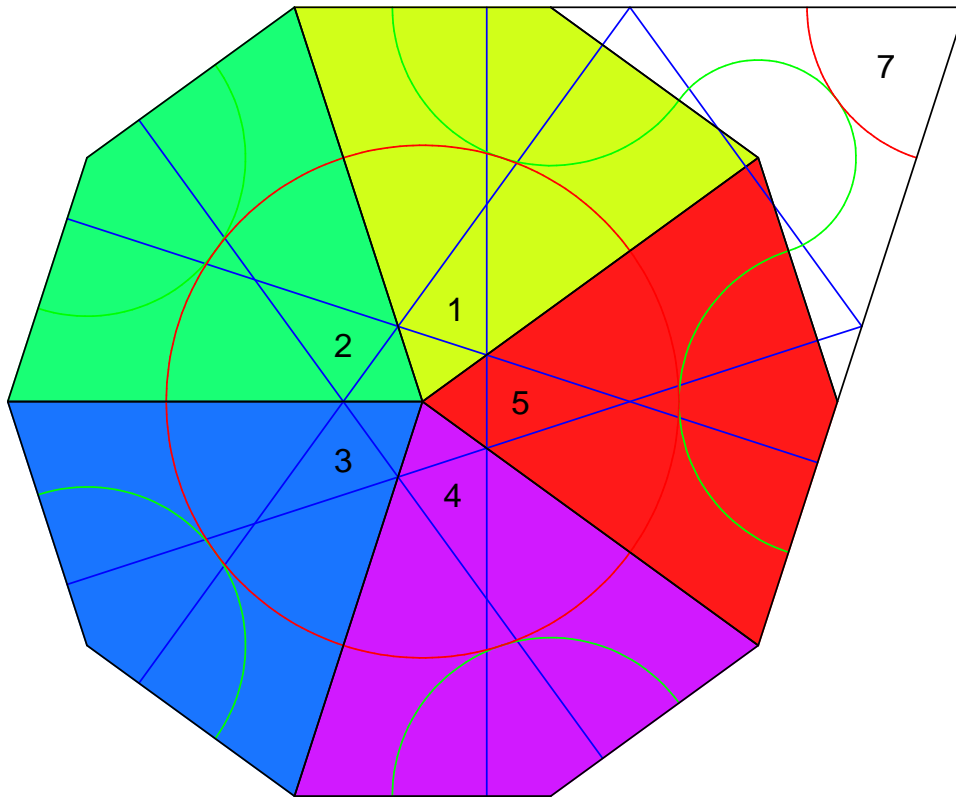
```
> tryrel(dart, redleft);
```

Dieser Versuch ist offenbar keine gute Idee, wir werden den Dart wohl woanders hinlegen müssen.



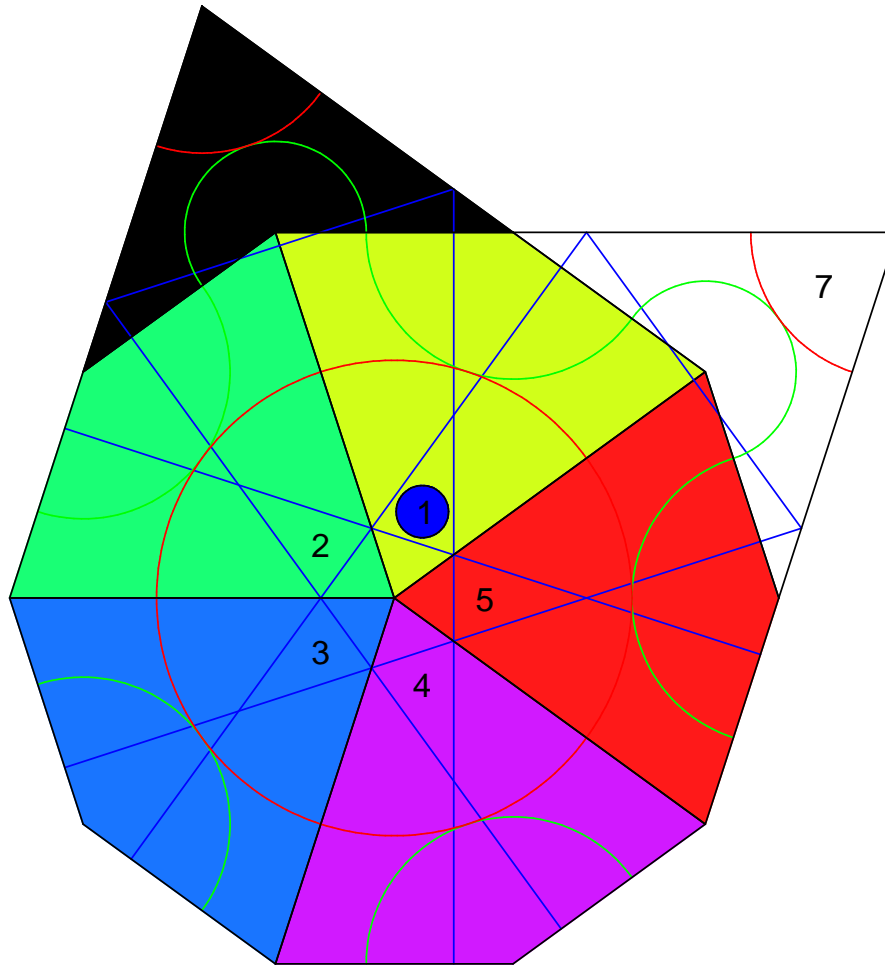
Das ist ein guter Zug:

```
> dorel(dart,greenleft);
```



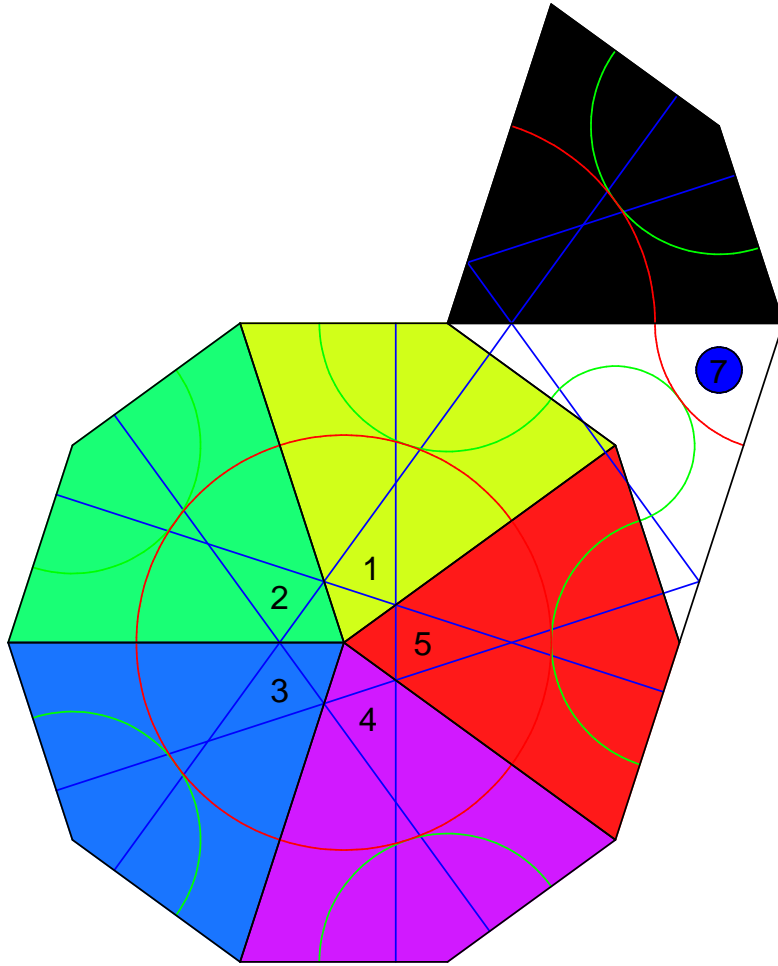
Jetzt wollen wir aber nicht an die zuletzt gelegte Kachel anlegen, sondern an die Kachel mit der Nummer 1. Dabei hilft uns das Kommando `atrel`. Sein Argument gibt die Nummer der Kachel an, das nun aktuell sein soll. Wir machen gleich auch einen Versuch, ein Dart anzulegen.

```
> atrel(1): tryrel(dart,greenleft);
```



Wir können aber auch anders weitermachen. (Ohne Argument macht `atrel` die jüngste Kachel zur aktuellen.)

```
> atrel(): tryrel( kite, redright );
```

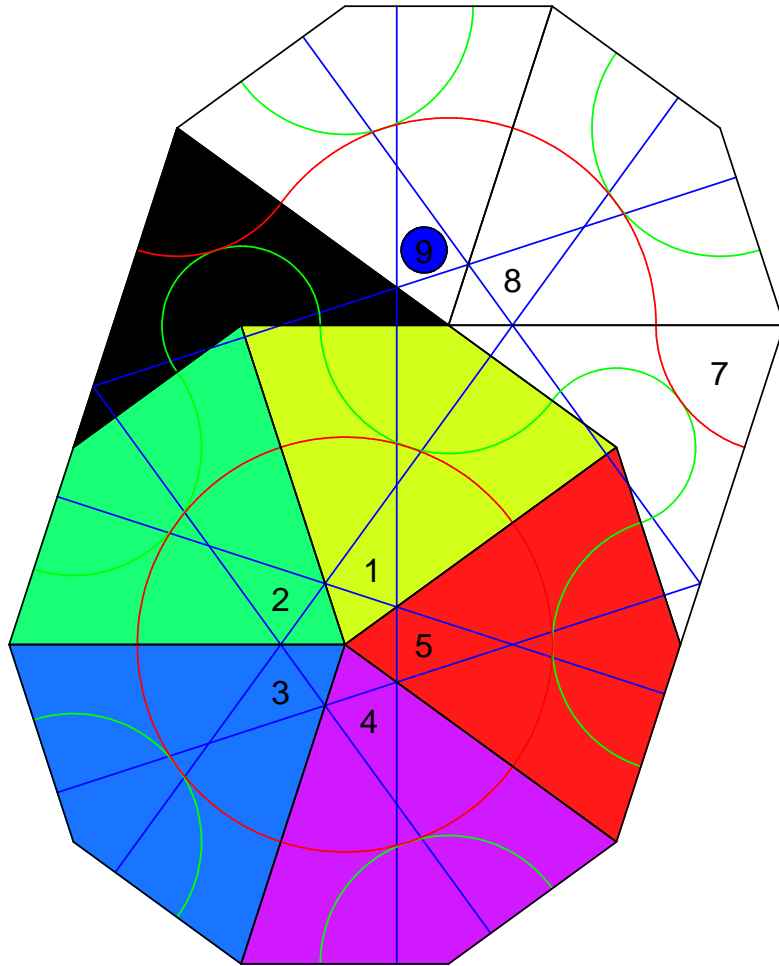


Legen wir dieses und ein weiteres Kite an.

```
> dorel( kite, redright );  
    dorel( kite, redleft );
```

Jetzt müsste wieder ein Dart folgen. Und richtig ...

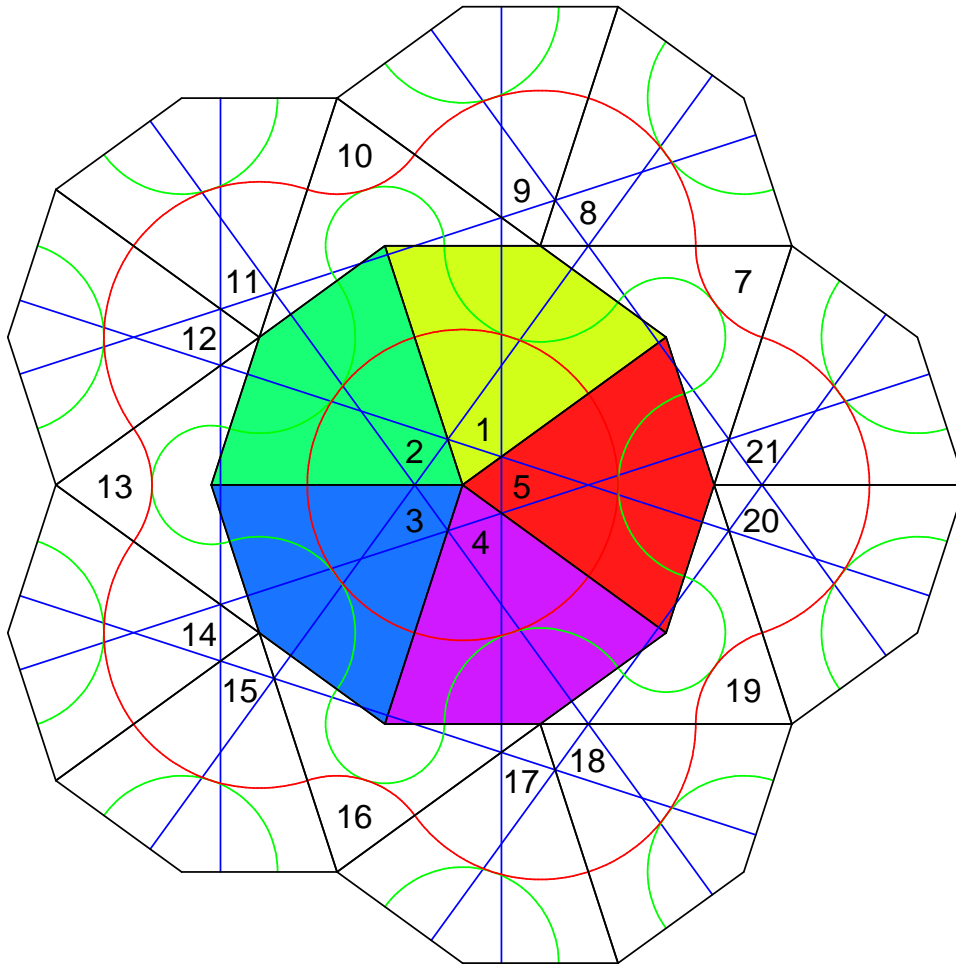
```
> tryrel( dart, redleft );
```



Das scheint so weiterzugehen:

```
> dorel( dart, redleft ): dorel( kite, redright ): dorel( kite,
  redleft ):
  dorel( dart, redleft ): dorel( kite, redright ): dorel( kite,
  redleft ):
  dorel( dart, redleft ): dorel( kite, redright ): dorel( kite,
  redleft ):
  dorel( dart, redleft ): dorel( kite, redright ): dorel( kite,
  redleft );
```

Ja gut!



Nach etwas Probieren finden wir eine weitere Runde.

```
> to 5 do
  dorel( kite, greenleft ):
  dorel( kite, redright ):
  dorel( dart, greenright ):
  dorel( dart, redright ):
  dorel( dart, redright ):
end do:
```

Hoppla, das ist wohl nicht erlaubt:

```
> tryrel( dart, greenright );
```

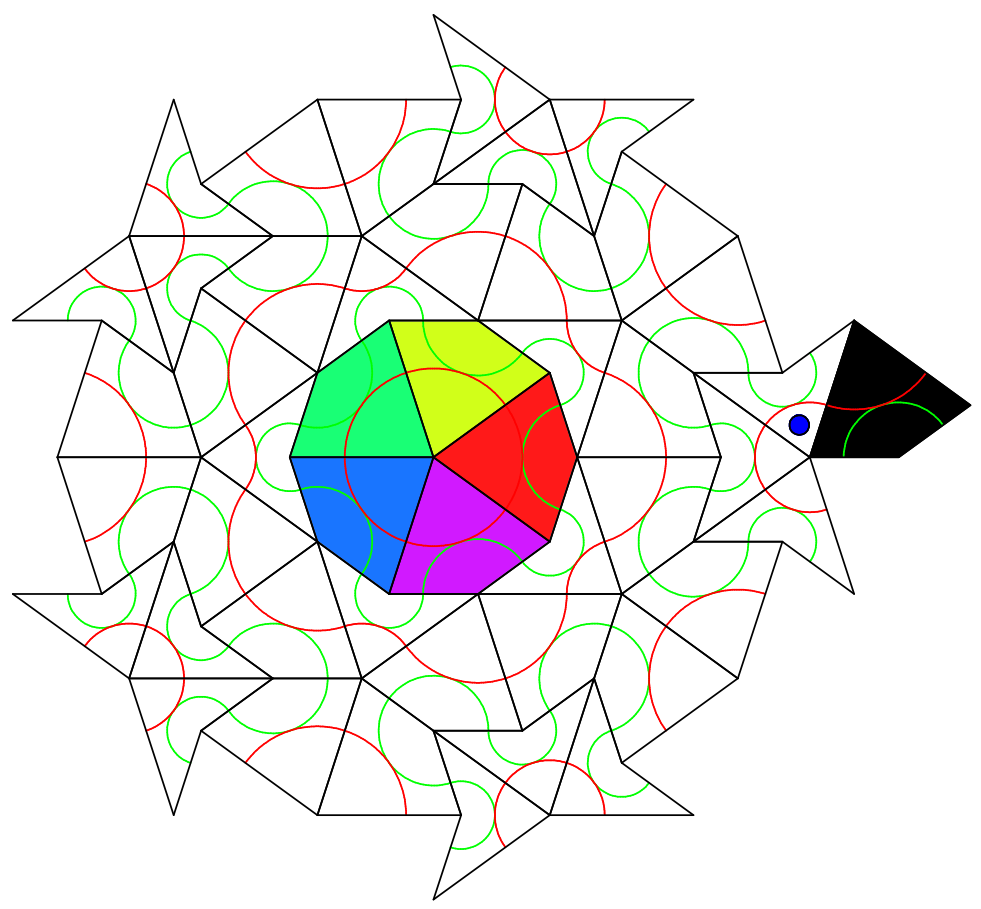
Wir erhalten eine Fehlermeldung.

```
Error, (in tryrel) Dart to dart can never be at green line
```

Aber so könnte es weitergehen. (Das Kommando `removecurves` ermöglicht uns, die roten, grünen und blauen Linien auszublenden. Das erste Argument ist eine Menge von Farben, das zweite eine Zeichnung.)

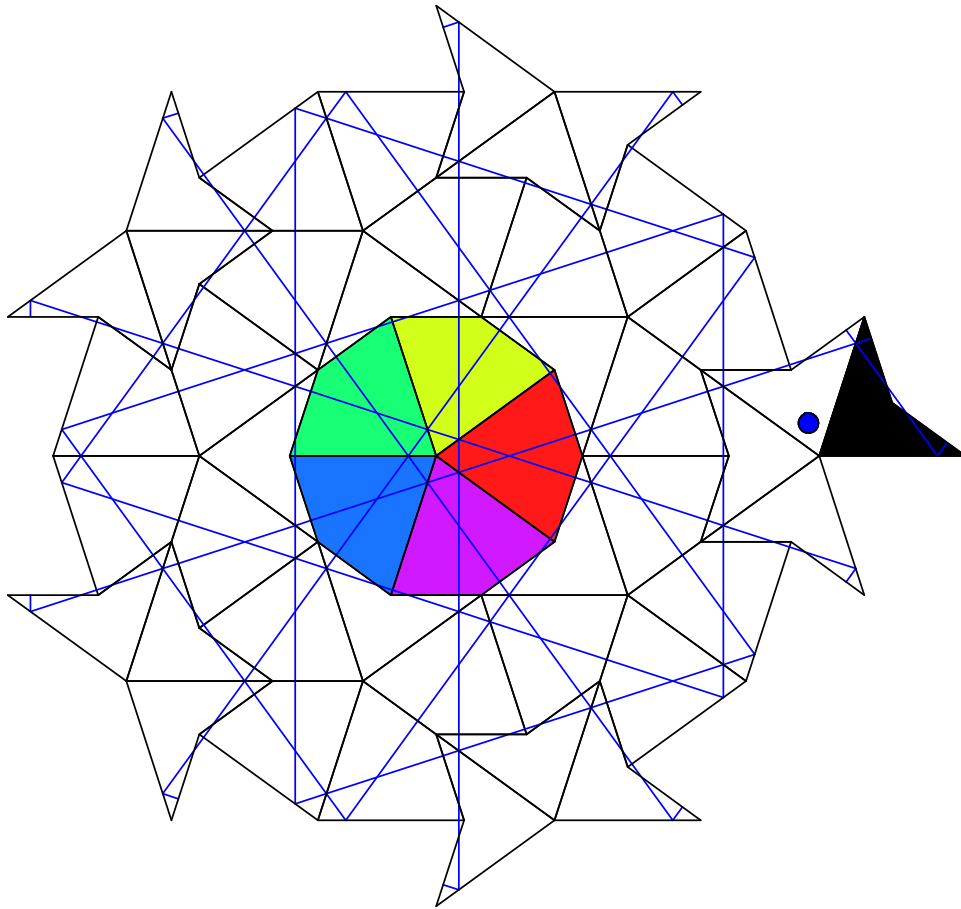
```
> removecurves(
  {blue,TEXT},
```

```
tryrel( kite, redright )  
);
```



Oder so:

```
> removecurves(  
  {red,green,TEXT},  
  tryrel( dart, redright )  
);
```



Versucht selbst weiterzubauen. Vielleicht lernt Ihr ja sogar soviel, dass Ihr ein Muster automatisch (beliebig weit) erzeugen könnt?

>

[

+ Der Stern

[>

+ Die Königin

[>

+ Batman

[>

+ Deflation

[>

+ Deflation der Sonne

[>

+ Sieben Punktfigurationen ...

[>

+ Eine Beobachtung

[>

[>

[>