The making of an object model for TikZ

Vincent Pit

2010-08-05

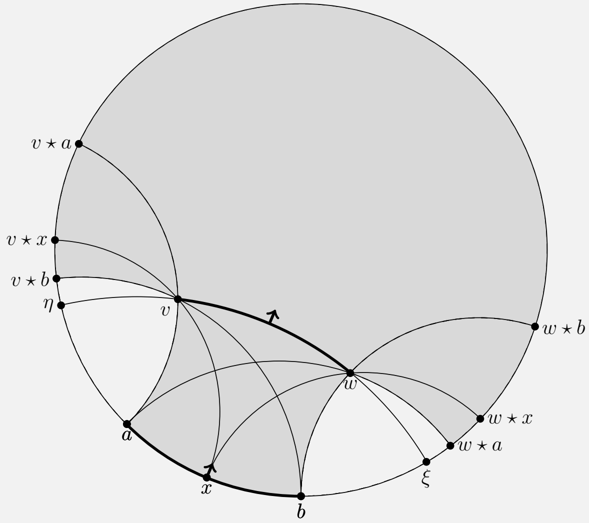
Setting

I'm a PhD student in pure Mathematics.

Setting

For my thesis, I have to draw geometric figures.

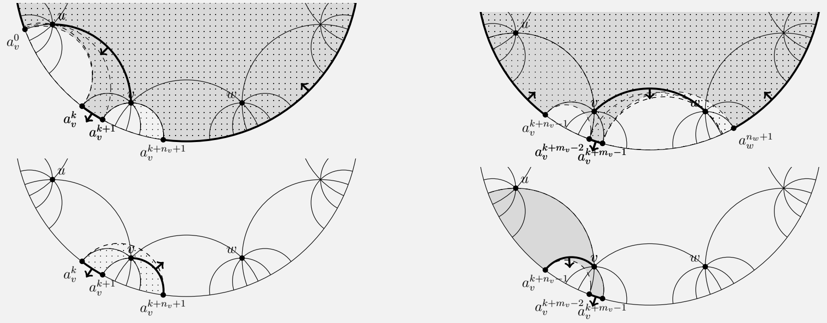
Setting



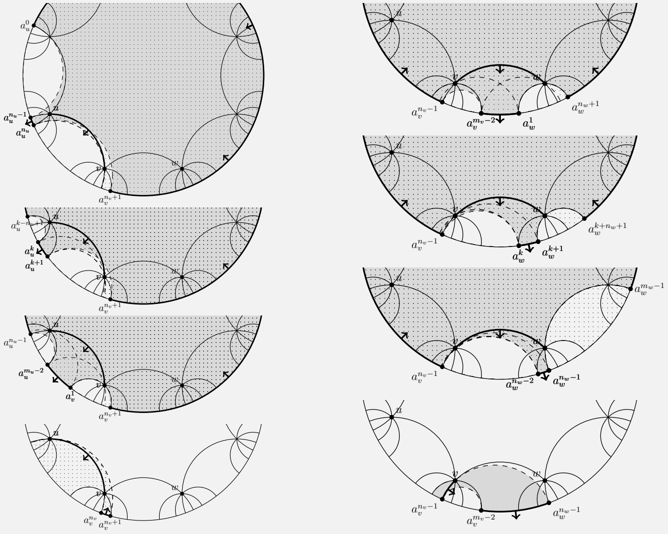
Setting

Lots of them, actually.

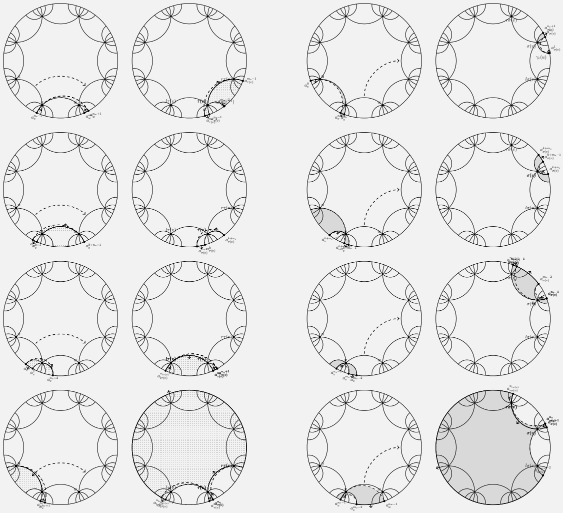
Setting


u
a_v^0
v
w
a_v^k
a_v^{k+1}
$a_v^{k+n_v+1}$
u
v
w
a_v^k
a_v^{k+1}
$a_v^{k+n_v+1}$
u
v
w
$a_v^{k+n_v-1}$
$a_v^{k+m_v-2}$
$a_v^{k+m_v-1}$
$a_w^{n_w+1}$
u
v
w
$a_v^{k+n_v-1}$
$a_v^{k+m_v-2}$
$a_v^{k+m_v-1}$

Setting


u
v
w
a_u^0
$a_u^{n_u-1}$
$a_u^{n_u}$
$a_v^{n_v+1}$
$a_u^{k-n_u+1}$
a_u^k
a_u^{k+1}
$a_u^{m_u-2}$
a_v^1
$a_v^{n_v}$
$a_v^{n_v-1}$
$a_v^{m_v-2}$
a_w^1
$a_w^{n_w+1}$
a_w^k
a_w^{k+1}
$a_w^{k+n_w+1}$
$a_w^{m_w-1}$
$a_w^{n_w-2}$
$a_w^{n_w-1}$

Setting



Setting

Only my advisor would think I could maintain this manually.

Drawings in LaTeX

- When only `latex` was available, people used `pstricks`.

Drawings in LaTeX

- When only `latex` was available, people used `pstricks`.
- Now that we have `pdflatex`, the canonical replacement is `PGF/TikZ`.

But what is TikZ ?

TikZ is a sublanguage for LaTeX for drawing graphs and figures.

Primitives

Points

```
(2cm,1cm) ;
```

Primitives

Lines

```
(-2cm,0cm) -- (2cm,0cm) ;
```

Primitives

Circles

```
(0cm,0cm) circle (1cm) ;
```

Primitives

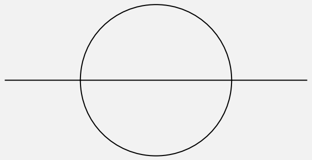
Paths

```
(-2cm,0cm) -- (2cm,0cm) (0cm,0cm) circle (1cm) ;
```

Primitives

Draw a path

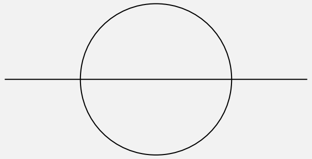
```
\draw (-2cm,0cm) -- (2cm,0cm) (0cm,0cm) circle (1cm) ;
```



Primitives

Draw a sequence

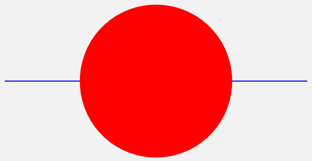
```
\draw (-2cm,0cm) -- (2cm,0cm) ;
\draw (0cm,0cm) circle (1cm) ;
```



Primitives

Path modifiers

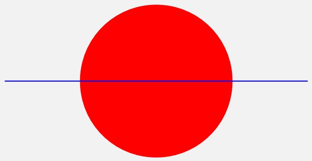
```
\draw [color=blue] (-2cm,0cm) -- (2cm,0cm) ;
\draw [fill,color=red] (0cm,0cm) circle (1cm) ;
```



Primitives

Path modifiers continued

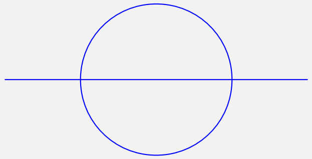
```
\draw [fill,color=red] (0cm,0cm) circle (1cm) ;
\draw [color=blue] (-2cm,0cm) -- (2cm,0cm) ;
```



Primitives

Scopes

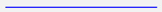
```
\begin{scope}[color=blue]
\draw (0cm,0cm) circle (1cm) ;
\draw (-2cm,0cm) -- (2cm,0cm) ;
\end{scope}
```



Primitives

Clips

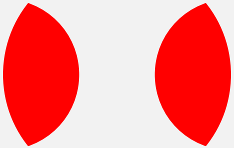
```
\begin{scope}[color=blue]
\clip (0cm,0cm) circle (1cm) ;
\draw (-2cm,0cm) -- (2cm,0cm) ;
\end{scope}
```



Primitives

Clips continued

```
\begin{scope}
\clip (-1.5cm,0cm) circle (1cm)
      (1.5cm,0cm) circle (1cm) ;
\draw [fill,color=red] (0cm,0cm) circle (1.5cm) ;
\end{scope}
```



Process

- The drawing characteristics are produced by a Perl script ;

Process

- The drawing characteristics are produced by a Perl script ;
- Each geometric object (points, arcs, ...) are modeled by a class ;

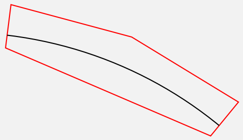
Process

- The drawing characteristics are produced by a Perl script ;
- Each geometric object (points, arcs, ...) are modeled by a class ;
- Each class has a method to serialize to `TikZ` code recursively.

TikZ limitations

There is also an arc primitive, but with angle accuracy of one degree.

TikZ limitations



Constraints

- Must handle clips ;

Constraints

- Must handle clips ;
- Must handle layers ;

Constraints

- Must handle clips ;
- Must handle layers ;
- Should optimize object `TikZ` code as much as possible ;

Constraints

- Must handle clips ;
- Must handle layers ;
- Should optimize object `TikZ` code as much as possible ;
- Easily pluggable with other geometric systems ;

Constraints

- Must handle clips ;
- Must handle layers ;
- Should optimize object `TikZ` code as much as possible ;
- Easily pluggable with other geometric systems ;
- Stay reasonably fast.

LaTeX::TikZ

Basics

```
use LaTeX::TikZ;
```

Basics

```
use LaTeX::TikZ;
my $p = Tikz->method(@args);
```

Basics

```
use LaTeX::TikZ;
my $p = Tikz->method(@args);
my $formatter = Tikz->formatter;
```

Basics

```
use LaTeX::TikZ;
my $p = Tikz->method(@args);
my $formatter = Tikz->formatter;
my ($head, $decl, $body) = $formatter->render($p);
```

Inclusion in a LaTeX document

```
\documentclass[12pt]{article}
...
@$head
...
\begin{document}
@$decl
...
@$body
...
\end{document}
```

Objects

(Any::)Moose-based.

Objects

`(Any::)Moose`-based.

Two main roles :

Objects

`(Any::)Moose`-based.

Two main roles :

- Geometric figures (sets) consume `LaTeX::TikZ::Set`.
  They must implement `draw`.

Objects

`(Any::)Moose`-based.

Two main roles :

- Geometric figures (sets) consume `LaTeX::TikZ::Set`.
  They must implement `draw`.
- Modifiers (mods) consume `LaTeX::TikZ::Mod`.
  They must implement `tag`, `covers`, `declare`, `apply`.

Interface

Points

Points are represented by `LaTeX::TikZ::Set::Point` objects.

```
my $origin = Tikz->point;
```

```
\draw (0cm,0cm) ;
```

Interface

Points continued

```
my $u_x = Tikz->point(1);
```

```
\draw (1cm,0cm) ;
```

Interface

Points continued again

```
my $u_y = Tikz->point(0, 1);
```

```
\draw (0cm,1cm) ;
```

Interface

Points continued yet again

```
my $xy = Tikz->point([1, 1]);
```

```
\draw (1cm,1cm) ;
```

Interface

Points continued and last

```
use Math::Complex;
my $xy_2 = Tikz->point((1 + i) / 2);
```

```
\draw (0.5cm,0.5cm) ;
```

Interface

Lines

Lines are `LaTeX::TikZ::Set::Line` objects, built from two `LaTeX::TikZ::Set::Points`.

```
my $l = Tikz->line([-1, 0] => [0, 1]);
```

```
\draw (-1cm,0cm) -- (0cm,1cm) ;
```

Interface

Numbers, array references and `Math::Complex` objects are automatically coerced into `LaTeX::TikZ::Point` objects.

Interface

Cicles

Circles are `LaTeX::TikZ::Set::Circle` objects, built from a `LaTeX::TikZ::Set::Point` and a length.

```
use Math::Complex;
my $c = Tikz->circle((1+i) => 1);
```

```
\draw (1cm,1cm) circle (1cm) ;
```

Interface

Polylines

Those are `LaTeX::TikZ::Set::Polyline` objects, built out of a list of `LaTeX::TikZ::Set::Points`.

```
my $U = Tikz->polyline(
 [ 0, 1 ] => 0 => 1 => [ 1, 1 ]
);
```

```
\draw (0cm,1cm) -- (0cm,0cm) -- (1cm,0cm)
                -- (1cm,1cm) ;
```

Interface

Polylines continued

```
use Math::Complex;
my $diamond = Tikz->closed_polyline(
 i() => -1 => -2*i() => 1
);
```

```
\draw (0cm,1cm) -- (-1cm,0cm) -- (0cm,-2cm)
                -- (1cm,0cm) -- cycle ;
```

Interface

Sequences

```
my $seq = Tikz->seq(
 Tikz->line(-2 => 2),
 Tikz->seq(
  Tikz->circle(-2 => 1),
  Tikz->circle(2 => 1),
 );
);
```

```
\draw (-2cm,0cm) -- (2cm,0cm) ;
\draw (-2cm,0cm) circle (1cm) ;
\draw (2cm,0cm) circle (1cm) ;
```

Interface

Modifiers

`LaTeX::TikZ` abstracts modifiers, layers and clips into the same concept.

Interface

Modifiers

Modifiers are applied with the mod method, which returns the set object.

```
my $red = Tikz->color('red');
$diamond->mod($red);
```

```
\draw [color=red] (0cm,1cm) -- (-1cm,0cm)
-- (0cm,-2cm) -- (1cm,0cm) -- cycle ;
```

Interface

Modifiers continued (clips)

```
my $unit_circle = Tikz->circle(0 => 1);
my $boundary = Tikz->clip($unit_circle);
my $arc = Tikz->circle([1, 1] => 1)->mod($boundary);
```

```
\begin{scope}
\clip (0cm,0cm) circle (1cm) ;
\draw (1cm,1cm) circle (1cm) ;
\end{scope}
```

Interface

Modifiers continued (clips)

```
my $arc = Tikz->circle([1, 1] => 1)
              ->clip(Tikz->circle(0 => 1));
```

```
\begin{scope}
\clip (0cm,0cm) circle (1cm) ;
\draw (1cm,1cm) circle (1cm) ;
\end{scope}
```

Interface

Modifiers continued (layers)

```
my $top = Tikz->layer('top');
my $bottom = Tikz->layer('bottom', below => [ 'top' ]);
```

```
\usetikzlibrary{patterns}
\pgfdeclarelayer{bottom}
\pgfdeclarelayer{top}
\pgfsetlayers{bottom,main,top}
```

Interface

Modifiers continued (layers)

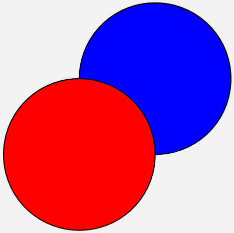
```
my $discs = Tikz->seq(
 Tikz->circle(0 => 1)
     ->mod(Tikz->fill('red'), $top),
 Tikz->circle([1, 1] => 1)
     ->mod(Tikz->fill('blue'), $bottom),
);
```

```
\begin{pgfonlayer}{top}
\draw [fill=red] (0cm,0cm) circle (1cm) ;
\end{pgfonlayer}
\begin{pgfonlayer}{bottom}
\draw [fill=blue] (1cm,1cm) circle (1cm) ;
\end{pgfonlayer}
```

Interface

Modifiers continued (layers)

```
my $discs = Tikz->seq(
 Tikz->circle(0 => 1)
     ->mod(Tikz->fill('red'), $top),
 Tikz->circle([1, 1] => 1)
     ->mod(Tikz->fill('blue'), $bottom),
);
```



Interface

Functors

A functor takes a `LaTeX::TikZ::Set` tree and clones it according to certain rules.

Interface

Functors - examples

The default set of rules gives you a clone functor :

```
my $clone = Tikz->functor;
my $dup = $set->$clone;
```

Interface

Functors - examples

A translation functor :

```
my $translate = Tikz->functor(
 'LaTeX::TikZ::Set::Point' => sub {
  my ($functor, $set, $x, $y) = @_;
  $set->new(
   point => [ $set->x + $x, $set->y + $y ],
  );
 },
);
my $shifted = $set->$translate(1, 1);
```

Optimizations

Using an object model allows for some interesting optimizations.

Optimizations

Don’t repeat mods

```
my $circles = Tikz->seq(
 Tikz->circle([1, 0] => 1)->mod(Tikz->color('red')),
 Tikz->circle([-1, 0] => 1),
)->mod(Tikz->color('red'));
```

```
\begin{scope}[color=red]
\draw (1cm,0cm) circle (1cm) ;
\draw (-1cm,0cm) circle (1cm) ;
\end{scope}
```

Optimizations

Don’t repeat mods

```
my $circles = Tikz->seq(
 Tikz->circle([1, 0] => 1)->mod(Tikz->color('red')),
 Tikz->circle([-1, 0] => 1),
)->mod(Tikz->color('blue'));
```

```
\begin{scope}[color=blue]
\draw [color=red] (1cm,0cm) circle (1cm) ;
\draw (-1cm,0cm) circle (1cm) ;
\end{scope}
```

Optimizations

Don't repeat mods - how it works

- When calling `->draw` on a composite type, its mods are marked as "applied", so that they aren't emitted for its elements.

Optimizations

Don't repeat mods - how it works

- When calling `->draw` on a composite type, its mods are marked as "applied", so that they aren't emitted for its elements.
- `LaTeX::TikZ` considers that two mods `$m1` and `$m2` are equivalent when :
  `$m1->tag eq $m2->tag and $m1->covers($m2)`

Optimizations

Factor contiguous similar clips

```
my $unit_disc = Tikz->circle(0 => 1);
my $arcs = Tikz->seq(
 Tikz->circle([1, 0] => 1)->clip($unit_disc),
 Tikz->circle([-1, 0] => 1)->clip($unit_disc),
);
```

```
\begin{scope}
\clip (0cm,0cm) circle (1cm) ;
\draw (1cm,0cm) circle (1cm) ;
\draw (-1cm,0cm) circle (1cm) ;
\end{scope}
```

Optimizations

Factor contiguous similar clips - How it works

- Calling `->draw` on each clipped circle returns a lazy representation of the object `TikZ` code in the form of a `LaTeX::TikZ::Scope` object.

Optimizations

Factor contiguous similar clips - How it works

- Calling `->draw` on each clipped circle returns a lazy representation of the object `TikZ` code in the form of a `LaTeX::TikZ::Scope` object.
- When gathered in one sequence, `LaTeX::TikZ::Scope::fold` is called to perform the optimization.

Autoloading of type coercions

- When the user provides an object of type `User::Point` instead of a `LaTeX::TikZ::Point`, `LaTeX::TikZ::Point::User::Point` is loaded in hope that it contains a type constraint from `User::Point` to `LaTeX::TikZ::Point`.

Autoloading of type coercions

- When the user provides an object of type `User::Point` instead of a `LaTeX::TikZ::Point`, `LaTeX::TikZ::Point::User::Point` is loaded in hope that it contains a type constraint from `User::Point` to `LaTeX::TikZ::Point`.
- All the black magic is contained in `LaTeX::TikZ::Meta::TypeConstraint::Autocoerce`.

Autoloading of type coercions

Example

```
package LaTeX::TikZ::Point;
use Any::Moose;
has "x"; has "y";
use LaTeX::TikZ::Meta::TypeConstraint::Autocoerce;
use Any::Moose 'Util::TypeConstraints';
register_type_constraint(
 LaTeX::TikZ::Meta::TypeConstraint::Autocoerce->new(
  name => 'LaTeX::TikZ::Point::Autocoerce',
  parent => find_type_constraint(__PACKAGE__),
 );
);
```

Autoloading of type coercions

Example

```
package LaTeX::TikZ::Set::Point;
use LaTeX::TikZ::Point;
has "point" => (
 isa => 'LaTeX::TikZ::Point::Autocoerce',
 coerce => 1,
);
```

Autoloading of type coercions

Example

```
package LaTeX::TikZ::Point::Math::Complex;
use Math::Complex;
use LaTeX::TikZ::Point;
use Any::Moose 'Util::TypeConstraints';
my $mc_tc = class_type 'Math::Complex';
coerce 'LaTeX::TikZ::Point::Autocoerce'
    => from 'Math::Complex'
    => via {
     LaTeX::TikZ::Point->new(x => $_->Re, y => $_->Im)
};
```

Thanks!

Thank you for your attention!

Questions?