On The Internals of Custom Ti*k*Z Path Operators

Digamma

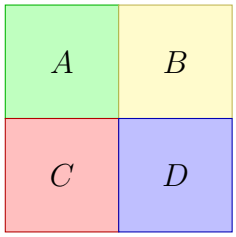
Abstract

In this article we will dive deep into how the notorious Ti*k*Z actually draws lines, circles, rectangles, Bézier curves, and so on using their path operators, and most importantly, how to make one of your own. We will dissect a few of them made by me while learning a thing or two (or nothing) from it.

Contents

1 A Special Case of `rectangle`

We implement a new path operator `square` to draw squares[1], obviously. We do this by some pure TEX necromancy, because screw expl3.




```
\begin{tikzpicture}
    \filldraw[fill=green!25, draw=green!70!black, invert] (0,0)
    ↪  square (-1cm) node at (-0.5,0.5) {$A$};
    \filldraw[fill=yellow!25, draw=yellow!70!black] (0,0) square
    ↪  (1cm) node at (0.5,0.5) {$B$};
    \filldraw[fill=red!25, draw=red!70!black] (0,0) square
    ↪  (-1cm) node at (-0.5,-0.5) {$C$};
    \filldraw[fill=blue!25, draw=blue!70!black, invert] (0,0)
    ↪  square (1cm) node at (0.5,-0.5) {$D$};
\end{tikzpicture}
```

1.1 Implementation

To implement this, one must modify `\tikz@schar`, a Ti*k*Z internal that handles path operators beginning with the letter "s". In `tikz.code.tex`, it's defined as:

```
\def\tikz@schar{\pgfutil@ifnextchar
↪  i{\tikz@sine}{\tikz@svg@path}}%
```

One can change this to account for our `square` path operator:

```
\def\tikz@schar{%
    \pgfutil@ifnextchar q
        {\tikz@square}
        {\pgfutil@ifnextchar i
```

[1]Of course, one can do this with `(0,0) rectangle (1,1)`, but I don't want to.

```
            {\tikz@sine}
            {\tikz@svg@path}}%
}
```

We add a line that checks whether the next character is "q". If it is, it executes `\tikz@square`. If not fall back to the next character checker.

For `\tikz@square`, we define it as follows:

```
\def\tikz@square quare{%
    \tikz@flush@moveto%
    \pgfutil@ifnextchar({\tikz@do@square}{\tikzerror{Expected
    ↪  parenthesis after square}}%
}
```

The `\tikz@flush@moveto` macro flushes any "pending" move-to operation, i.e., it ensures that the current point is actually set before drawing the square. Ti*k*Z uses this to synchronize path parsing.

After that, we invoke the last `\pgfutil@ifnextchar` which checks for a parenthesis. If it's there, proceed to `\tikz@do@square` (which we'll define in a second). If not, then Ti*k*Z will complain about your coding skills.

And finally the construction process. Of course, one must define the macro `\tikz@do@square` from the earlier code snippet, otherwise your code is just nonsense (and utterly erroneous). As the name suggests, you want this to draw the actual square. Let's start things off with some dimensions:

```
\def\tikz@do@square(#1){%
    \pgfmathsetlength{\pgf@xa}{#1}%
    \pgf@xb=\tikz@lastx
    \pgf@yb=\tikz@lasty
```

Since we want a syntax like `(<coord>) square (<dim>)`, we convert `\pgf@xa` to a dimension. This will be the side length of the square. We also set `\pgf@xb` and `\pgf@yb` to `\tikz@lastx` and `\tikz@lasty` respectively. Why? This saves the last known coordinate in the path to `\pgf@xb` and `\pgf@yb`. Think of it as the starting corner of the square. With that in mind, we may proceed to the next part of the definition:

```
    \iftikz@square@invert
        \advance\pgf@yb by -\pgf@xa%
    \else
        \advance\pgf@yb by \pgf@xa%
    \fi
```

Actually, this is one part that I'll explain in more detail after fully unpacking the definition of `\tikz@do@square`, but essentially it kind of inverts the square direction. Moving on, with the starting corners already fancifully set up, we proceed to draw the square:

```
    \advance\pgf@xb by \pgf@xa%
    \pgfpathrectanglecorners{\pgfqpoint{\tikz@lastx}{\tikz@last
    ↪  y}}{\pgfqpoint{\pgf@xb}{\pgf@yb}}
    \tikz@lastx=\pgf@xb
    \tikz@lasty=\pgf@yb
    \tikz@scan@next@command
}
```

The line `\advance\pgf@xb by \pgf@xa` always shifts to the right by the side length (for positive `\pgf@xa`). We adopt the rectangle construction process in `\tikz@rect` for simplicity (even though defining a path operator is not at all simple unless you know what you're doing, or just a lunatic in general like me).

You could say we are technically drawing a rectangle with equal side lengths from the old point to the new one since we are using PGF macros like `\pgfpathrectanglecorners`, but that is if you never paid attention to your elementary geometry classes.

These lines in particular:

```
\tikz@lastx=\pgf@xb
\tikz@lasty=\pgf@yb
```

might have caught your attention. Earlier, we set this the other way around, now we're setting it back to `\pgf@xb` and `\pgf@yb`. Confusing, right? Well, unless you know how to read TEX, yes, but not really. It just tells Ti*k*Z to update the last position so the path continues from the new corner. And the cherry on top — `\tikz@scan@next@command` — is just a hacker's way of saying "yeah I'm finished with this, continue checking for commands". I promise you this is NOT an April Fool's article. This is in

fact real.

There is one last thing I haven't gone in detail yet. As promised, we'll go into how this snippet actually works:

```
\iftikz@square@invert
    \advance\pgf@yb by -\pgf@xa%
\else
    \advance\pgf@yb by \pgf@xa%
\fi
```

First, we define a new conditional:

```
\newif\iftikz@square@invert
```

Notice how I used the `invert` key on the code of the diagram in the first page. This brings us back to `\tikz@do@square`. What it does is that for positive dimensions, it places the square downwards, and for negative dimensions, it places the square upwards.

You could say typing for example

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

is equivalent to

```
\draw[invert] (0,0) square (1cm);
```

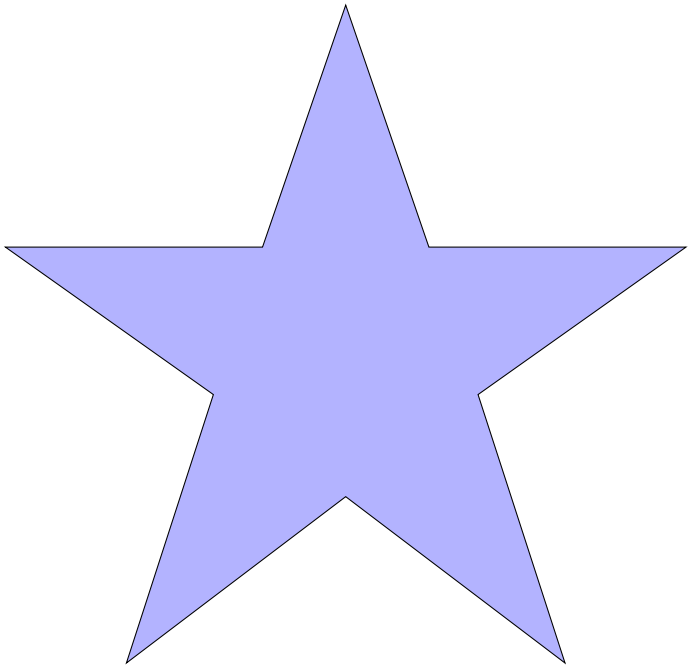
but the latter can be quite handy at times. I don't know if you actually understood all this and will probably just copy paste the code because you want to look cool in front of your friends, but yeah sometimes TEX can be pretty arcane. Anyway if you want just go take the full code and do some wild stuff with your new path operator.

```
% \catcode`\@=11 (or for the noobs out there, \makeatletter)

\newif\iftikz@square@invert
\tikzset{
    invert/.is if=tikz@square@invert,
    invert/.default=true,
}
```

```
\def\tikz@schar{%
    \pgfutil@ifnextchar q
        {\tikz@square}
        {\pgfutil@ifnextchar i
            {\tikz@sine}
            {\tikz@svg@path}}%
}

\def\tikz@square quare{%
    \tikz@flush@moveto%
    \pgfutil@ifnextchar({\tikz@do@square}{\tikzerror{Expected
    ↪  parenthesis after square}}%
}

\def\tikz@do@square(#1){%
    \pgfmathsetlength{\pgf@xa}{#1}%
    \pgf@xb=\tikz@lastx
    \pgf@yb=\tikz@lasty
    \iftikz@square@invert
        \advance\pgf@yb by -\pgf@xa%
    \else
        \advance\pgf@yb by \pgf@xa%
    \fi
    \advance\pgf@xb by \pgf@xa%
    \pgfpathrectanglecorners{\pgfqpoint{\tikz@lastx}{\tikz@last⌋
    ↪  y}}{\pgfqpoint{\pgf@xb}{\pgf@yb}}
    \tikz@lastx=\pgf@xb
    \tikz@lasty=\pgf@yb
    \tikz@scan@next@command
}

% \catcode`\@=12
```

2 Drawing Weird Polygons

Let's kick things up a notch by defining another path operator that draws a closed path crossing n points. We'll call this new path operator `polygon`. One can use this like `(start) polygon {(coord1),(coord2),...,(end)}`.



```
\begin{tikzpicture}
    \draw[fill=blue!30] (0.0, 9.5) polygon {
        (2.2, 3.1),
        (9.0, 3.1),
        (3.5, -0.8),
        (5.8, -7.9),
        (0.0, -3.5),
        (-5.8, -7.9),
        (-3.5, -0.8),
        (-9.0, 3.1),
        (-2.2, 3.1),
        (0.0, 9.5)
    };
\end{tikzpicture}
```

Now, one might rightfully ask, "what the hell was the motivation behind this? you know you can easily do `(start) -- (coord1) -- (coord2) -- ... -- (end) -- cycle` right?". First of all, this was simply to show how much power one can yield with TEX (and consequentially Ti*k*Z). Second, I don't see *you* creating new path operators. Sorry, that was a bit too offensive, but anyway, yes you can do that but it gets a bit boring when you're doing one with more than 5 vertices.

2.1 Implementation

Moving on, we apply the same principle as earlier. Ti*k*Z has `\tikz@pchar`, handling path operators beginning with "p". Prior to our modification, this is defined as:

```
\def\tikz@pchar{\pgfutil@ifnextchar
↪  l{\tikz@plot}{\pgfutil@ifnextchar
↪  i{\tikz@subpicture}{\tikz@parabola}}}%
```

From here, you can easily modify the macro:

```
\def\tikz@pchar{%
    \pgfutil@ifnextchar o
        {\tikz@polygon}%
        {\pgfutil@ifnextchar l{\tikz@plot}{\pgfutil@ifnextchar
i{\tikz@subpicture}{\tikz@parabola}}}%
}
```

Nothing much to say here other than we need to define `\tikz@polygon`. Again, applying the same principles, one can define it like this:

```
\def\tikz@polygon olygon{%
    \tikz@flush@moveto
    \pgfutil@ifnextchar\bgroup
        {\tikz@do@polygon}%
        {\tikzerror{Expected \string\bgroup after polygon}}%
}
```

It might already be obvious, but I should still point out that `\bgroup` and `{` are the same thing here.

Now comes the hard part. We will now define `\tikz@do@polygon`. Remember that our desired behavior is to create a shape with the syntax `(start) polygon {(coord1),(coord2),...,(end)}`.

```
\def\tikz@do@polygon#1{%
    \pgfpathmoveto{\pgfqpoint{\tikz@lastx}{\tikz@lasty}}%
```

Recall that `\tikz@lastx` and `\tikz@lasty` store the last known x and y coordinates respectively. The `\pgfqpoint` takes the previous two macros

and converts them to PGF points (or coordinates, whatever you call them), similar to how you write them in high-level Ti*k*Z (i.e., `(x,y)`). Now, as the name suggests, `\pgfpathmoveto` starts a new subpath in the current path being constructed. It's like moving the "pen" to a new location without drawing anything.

For the next part, a `\foreach` loop might be more appropriate here since we can specify an arbitrary number of coordinates inside the group.

```
    \foreach \pt in {#1} {%
        \tikz@scan@one@point\pgfutil@firstofone\pt
        \pgfpathlineto{\pgfqpoint{\pgf@x}{\pgf@y}}%
    }%
```

In the loop, we define `\pt` as the argument. In this case, it's our list of coordinates inside the group. We look for a Ti*k*Z point syntax using `\ttikz@scan@one@point`. The `\pgfutil@firstofone` does nothing much other than expanding its argument. After that comes the line formation. This is essentially a low-level way of doing `\draw (A) -- (B)`. Then, you're almost done, just close the path, save last coordinates, and continue scanning for the next command:

```
    \pgfpathclose
    \tikz@lastx=\pgf@x  \tikz@lasty=\pgf@y
    \tikz@scan@next@command
}
```

Using `\foreach` loops are great when you're dealing with path operators handling a list of tokens (usually coordinates). To be honest I still find these pretty cursed but they work well so I can't complain. As always, here's the full code. Set the category code of @ to 11 or else TEX will be unhappy.

```
\def\tikz@pchar{%
    \pgfutil@ifnextchar o
        {\tikz@polygon}%
        {\pgfutil@ifnextchar l{\tikz@plot}{\pgfutil@ifnextchar
i{\tikz@subpicture}{\tikz@parabola}}
}%
}

\def\tikz@polygon olygon{%
```

```
    \tikz@flush@moveto
    \pgfutil@ifnextchar\bgroup
        {\tikz@do@polygon}%
        {\tikzerror{Expected \string\bgroup after polygon}}%
}

\def\tikz@do@polygon#1{%
    \pgfpathmoveto{\pgfqpoint{\tikz@lastx}{\tikz@lasty}}%
    \foreach \pt in {#1} {%
        \tikz@scan@one@point\pgfutil@firstofone\pt
        \pgfpathlineto{\pgfqpoint{\pgf@x}{\pgf@y}}%
    }%
    \pgfpathclose
    \tikz@lastx=\pgf@x  \tikz@lasty=\pgf@y
    \tikz@scan@next@command
}
```

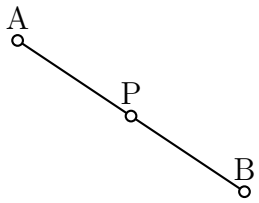
3 The Midpoint of Your Sanity

You thought we were done? Haha, no. Things just get more wicked here, whether you like it or not. And this time, we're going to introduce a new monster macro. Meet `\tikz@handle`:

```
% tikz.code.tex
% Central dispatcher for commands
\def\tikz@handle{%
    \pgfutil@switch\pgfutil@ifx\pgf@let@token{%
        {(}{\let\pgfutil@next\tikz@movetoabs}%)
        {+}{\let\pgfutil@next\tikz@movetorel}%
        {-}{\let\pgfutil@next\tikz@lineto}%
        {.}{\let\pgfutil@next\tikz@dot}%
        {r}{\let\pgfutil@next\tikz@rect}%
        {n}{\let\pgfutil@next\tikz@fig}%
        {[}{\let\pgfutil@next\tikz@parse@options}%]
        {c}{\let\pgfutil@next\tikz@cchar}%
        {\bgroup}{\let\pgfutil@next\tikz@beginscope}%
        {\egroup}{\let\pgfutil@next\tikz@endscope}%
        {;}{\let\pgfutil@next\tikz@finish}%
        {a}{\let\pgfutil@next\tikz@a@char}%
        {e}{\let\pgfutil@next\tikz@e@char}%
```

```
        {g}{\let\pgfutil@next\tikz@g@char}%
        {s}{\let\pgfutil@next\tikz@schar}%
        {|}{\let\pgfutil@next\tikz@vh@lineto}%
        {p}{\pgfsetmovetofirstplotpoint\let\pgfutil@next\tikz@p
        ↪  char}%
        {t}{\let\pgfutil@next\tikz@to}%
        {\pgfextra}{\let\pgfutil@next\tikz@extra}%
        {\foreach}{\let\pgfutil@next\tikz@foreach}%
        {f}{\let\pgfutil@next\tikz@fchar}%
        {\pgf@stop}{\let\pgfutil@next\relax}%
        {\par}{\let\pgfutil@next\tikz@scan@next@command}%
        {d}{\let\pgfutil@next\tikz@decoration}%
        {l}{\let\pgfutil@next\tikz@l@char}%
        {:}{\let\pgfutil@next\tikz@colon@char}%
        {\relax}{\relax\let\pgfutil@next\tikz@scan@next@command
        ↪  }%
    }{\tikz@resetexpandcount\pgfutil@next}{\tikz@expand}%
}%
```

We're going to have so much fun with this, as you'll see later. For now, consider the following diagram:



Can you guess what code snippet made this diagram? No? Obviously. Here:

```
\begin{tikzpicture}[thick]
    \coordinate (A) at (1,2);
    \coordinate (B) at (4,0);

    \draw (A) -- (B) midpoint (P);

    \foreach \p in {A,B,P} {
        \draw[fill=white] (\p) circle (2pt) node[above] {\p};
    }
\end{tikzpicture}
```

You probably see where I'm going with this one. We're going to define a "midpoint" operator. It's a little different this time though. Rather than drawing something, we define a coordinate based on the previous path operation.

3.1 Implementation

We will still apply the principles we introduced in the previous sections... except we can't, at least not yet. You see, Ti*k*Z does not have a path operator beginning with the letter "m". A bit problematic, isn't it? This is where `\tikz@handle` comes in. Suppose we're a hacker hijacking a communications system. That is exactly what we're gonna do: we're going to inject the letter "m" into the macro so Ti*k*Z will just accept it and move on. Now that we know our objective, it's not really that hard to do anymore:

```
\def\tikz@handle{%
    \pgfutil@switch\pgfutil@ifx\pgf@let@token{%
        {(}{\let\pgfutil@next\tikz@movetoabs}%
        {+}{\let\pgfutil@next\tikz@movetorel}%
        {-}{\let\pgfutil@next\tikz@lineto}%
        {.}{\let\pgfutil@next\tikz@dot}%
        {r}{\let\pgfutil@next\tikz@rect}%
        {n}{\let\pgfutil@next\tikz@fig}%
        {[}{\let\pgfutil@next\tikz@parse@options}%
        {c}{\let\pgfutil@next\tikz@cchar}%
        {\bgroup}{\let\pgfutil@next\tikz@beginscope}%
        {\egroup}{\let\pgfutil@next\tikz@endscope}%
        {;}{\let\pgfutil@next\tikz@finish}%
        {a}{\let\pgfutil@next\tikz@a@char}%
        {e}{\let\pgfutil@next\tikz@e@char}%
        {g}{\let\pgfutil@next\tikz@g@char}%
        {s}{\let\pgfutil@next\tikz@schar}%
        {|}{\let\pgfutil@next\tikz@vh@lineto}%
        {p}{\pgfsetmovetofirstplotpoint\let\pgfutil@next\tikz@p⌋
        ↪  char}%
        {t}{\let\pgfutil@next\tikz@to}%
        {\pgfextra}{\let\pgfutil@next\tikz@extra}%
        {\foreach}{\let\pgfutil@next\tikz@foreach}%
        {f}{\let\pgfutil@next\tikz@fchar}%
        {\pgf@stop}{\let\pgfutil@next\relax}%
        {\par}{\let\pgfutil@next\tikz@scan@next@command}%
```

```
        {d}{\let\pgfutil@next\tikz@decoration}%
        {l}{\let\pgfutil@next\tikz@l@char}%
        {:}{\let\pgfutil@next\tikz@colon@char}%
        {\relax}{\relax\let\pgfutil@next\tikz@scan@next@command
        ↪ }%
        {m}{\let\pgfutil@next\tikz@mchar}% Success!
    }{\tikz@resetexpandcount\pgfutil@next}{\tikz@expand}%
}
```

Now that we injected a new character into `\tikz@handle`, we can now safely apply what we've learned in Sections 1 and 2, if you still remember them that is. First and the most obvious one, define `\tikz@mchar`. Obviously, we can't just inject an undefined control sequence into `\tikz@handle`, that's just ridiculous. I can already sense the amount of errors you're gonna get when you do that. Anyway, you might want to define it like this:

```
\def\tikz@mchar{%
    \pgfutil@ifnextchar i{\tikz@midpoint}{\tikz@expand}%
}
```

Since our midpoint operator is literally the only path operator starting with "m", we use `\tikz@expand` as our third argument. The name is self-explanatory (it isn't, but it's nothing important). Anyway, we're now all geared up to define `\tikz@midpoint`:

```
\def\tikz@midpoint idpoint{%
    \tikz@flush@moveto%
    \tikz@@midpoint%
}

\def\tikz@@midpoint{%
    \pgfutil@ifnextchar({\tikz@do@midpoint}{\tikzerror{Expected
    ↪ parenthesis after midpoint}}%
}
```

I added an extra `\tikz@@midpoint` just to be fancy, but we simply apply what we did earlier, as well as adding an error for Ti*k*Z to complain about its users being noobs. And alas, the cherry on top, we end by coordinate construction using our beloved `\ttikzdo@midpoint`:

```
\def\tikz@do@midpoint(#1){%
    \edef\tikz@midpoint@name{\tikz@pp@name{#1}}%
```

Here, we define a macro inside the definition of a macro. Very fun. This is crucial in the later part of the definition though, and I know you definitely looked at `\tikz@pp@name`, I hacked into your webcam using TEX, unless you don't have one, in that case I installed one and hacked it. The macro with the peculiar name is just an internal used for setting the names of coordinates/nodes.

Moving on, we proceed to the calculation part:

```
    \pgfgetpath\tikz@currentpath%
    \pgfprocesspathextractpoints\tikz@currentpath%
    \pgf@process{\pgfpointsecondlastonpath}%
    \pgf@xa=\pgf@x%
    \pgf@ya=\pgf@y%
    \pgf@process{\pgfpointlastonpath}%
    \pgf@x=\dimexpr .5\pgf@x + .5\pgf@xa\relax
    \pgf@y=\dimexpr .5\pgf@y + .5\pgf@ya\relax
```

if you're good at both math and TEX/Ti*k*Z internals, you can easily create a code that will calculate the midpoint of a line. First, we detect the current path, then extract the points, and process both those points and do simple math.

After that, constructing a coordinate internally using the results we had in the previous snippet is trivial:

```
    \expandafter\pgfcoordinate\expandafter{\tikz@midpoint@name}
    ↪  {\pgfqpoint{\pgf@x}{\pgf@y}}%
```

and of course, we can't forget about finishing this up with Ti*k*Z scanning for the next command like a good boy:

```
    \tikz@scan@next@command
}
```

Usually when you want to inject a new path operator whose first character is not part of `\tikz@handle`, you'd do something like

```
{<char>}{\let\pgfutil@next\tikz@<char>char}%
```

or

```
{<char>}{\let\pgfutil@next\tikz@<char>@char}%
```

I usually prefer the former, I don't know why Till Tantau preferred `\tikz@g@char` over `\tikz@gchar`. And if you've ever used `tkz-euclide`, you probably just want me to do `\tkzDefMidpoint(A,B) \tkzGetPoint{P}` and call it a day. In that case, screw you (not really, I'm sorry) and that package. It is cool and all but mixing two different syntaxes feel awkward. Even Alain Matthes said it in the package manual in p. 33:

> "It is of course possible to use the tools of Ti*k*Z but it seems more logical to me not to mix the different syntaxes."

Oh and it's not just `tkz-euclide`, screw the `calc` library[2] as well! I will not do `\coordinate (P) at ($A!0.5!B$)` in my projects when I have to add an extra `\usetikzlibrary{calc}` line in my preamble. Rantings aside, here's the full code[3] for lazy coders:

```
\def\tikz@handle{%
    \pgfutil@switch\pgfutil@ifx\pgf@let@token{%
        {(}{\let\pgfutil@next\tikz@movetoabs}%
        {+}{\let\pgfutil@next\tikz@movetorel}%
        {-}{\let\pgfutil@next\tikz@lineto}%
        {.}{\let\pgfutil@next\tikz@dot}%
        {r}{\let\pgfutil@next\tikz@rect}%
        {n}{\let\pgfutil@next\tikz@fig}%
        {[}{\let\pgfutil@next\tikz@parse@options}%
        {c}{\let\pgfutil@next\tikz@cchar}%
        {\bgroup}{\let\pgfutil@next\tikz@beginscope}%
        {\egroup}{\let\pgfutil@next\tikz@endscope}%
        {;}{\let\pgfutil@next\tikz@finish}%
        {a}{\let\pgfutil@next\tikz@a@char}%
```

[2] I showed that I have beef with expl3, `tkz-euclide`, and now the Ti*k*Z library useful for on-the-fly calculation. That is purely for commentary, I do not hate them. In fact, I use them as well, but our main focus here is my custom path operators. Nevertheless, I apologize for the offensive comments.

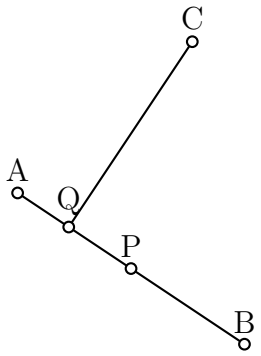
[3] The code was too long I had to add the option `breakable` in my `\newtcblisting{ltxbox}{<options>}`.

```
        {e}{\let\pgfutil@next\tikz@e@char}%
        {g}{\let\pgfutil@next\tikz@g@char}%
        {s}{\let\pgfutil@next\tikz@schar}%
        {|}{\let\pgfutil@next\tikz@vh@lineto}%
        {p}{\pgfsetmovetofirstplotpoint\let\pgfutil@next\tikz@p
        ↪  char}%
        {t}{\let\pgfutil@next\tikz@to}%
        {\pgfextra}{\let\pgfutil@next\tikz@extra}%
        {\foreach}{\let\pgfutil@next\tikz@foreach}%
        {f}{\let\pgfutil@next\tikz@fchar}%
        {\pgf@stop}{\let\pgfutil@next\relax}%
        {\par}{\let\pgfutil@next\tikz@scan@next@command}%
        {d}{\let\pgfutil@next\tikz@decoration}%
        {l}{\let\pgfutil@next\tikz@l@char}%
        {:}{\let\pgfutil@next\tikz@colon@char}%
        {\relax}{\relax\let\pgfutil@next\tikz@scan@next@command
        ↪  }%
        {m}{\let\pgfutil@next\tikz@mchar}%
    }{\tikz@resetexpandcount\pgfutil@next}{\tikz@expand}%
}

\def\tikz@mchar{%
    \pgfutil@ifnextchar i{\tikz@midpoint}{\tikz@expand}%
}

\def\tikz@midpoint idpoint{%
    \tikz@flush@moveto%
    \tikz@@midpoint%
}

\def\tikz@@midpoint{%
    \pgfutil@ifnextchar({\tikz@do@midpoint}{\tikzerror{Expected
    ↪  parenthesis after midpoint}}%
}

\def\tikz@do@midpoint(#1){%
    \edef\tikz@midpoint@name{\tikz@pp@name{#1}}%
    \pgfgetpath\tikz@currentpath%
    \pgfprocesspathextractpoints\tikz@currentpath%
    \pgf@process{\pgfpointsecondlastonpath}%
    \pgf@xa=\pgf@x%
    \pgf@ya=\pgf@y%
```

```
    \pgf@process{\pgfpointlastonpath}%
    \pgf@x=\dimexpr .5\pgf@x + .5\pgf@xa\relax
    \pgf@y=\dimexpr .5\pgf@y + .5\pgf@ya\relax
    \expandafter\pgfcoordinate\expandafter{\tikz@midpoint@name}
    ↪  {\pgfqpoint{\pgf@x}{\pgf@y}}%
    \tikz@scan@next@command%
}
```

4 Goofy Orthogonal Projections

Now comes the climax. This is my most complex implementation to date, and is actually the first one I've done. In this section we will dissect my `oproj` path operator. But before that, I want to tell you about the sponsor of this article, NordVPN! NordVPN is the most reliable VPN, giving you access to 50... Ahem, anyway, showcase time!




```
\begin{tikzpicture}[thick]
    \coordinate (A) at (1,2);
    \coordinate (B) at (4,0);
    \coordinate (C) at (3.31,4);

    \draw (A) -- (B) midpoint (P);
    \draw (C) oproj {(A)--(B)} (Q) -- (Q);

    \foreach \p in {A,B,C,P,Q} {
        \draw[fill=white] (\p) circle (2pt) node[above] {\p};
    }
\end{tikzpicture}
```

We now have a more complex syntax, `(coord1) oproj {(coord2)--(coord3)} (coord4)`[4]. It's possible (as demonstrated by the diagram), but it will be a very long and painful ride through the treacherous lands of TEX wizardry.

4.1 Implementation

Recall that Ti*k*Z once again does not have a handler for path operators beginning with "o", so we hook it ourselves using what we did in Section 3.1:

```
{o}{\let\pgfutil@next\tikz@ochar}%
```

Then, we're ready to define `\tikz@ochar`:

```
\def\tikz@ochar{%
    \pgfutil@ifnextchar p{\tikz@oproj}{\tikz@expand}%
}
```

Just a quick tip: if you're going to make another path operator beginning with "o" or "m", replace the `\tikz@expand` with something like `\tikz@oofmyself` or whatever you want to name it. Anyway, we now proceed to define `\tikz@oproj`:

```
\def\tikz@oproj proj{%
    \tikz@flush@moveto
    \tikz@@oproj%
}

\def\tikz@@oproj{%
    \pgfutil@ifnextchar\bgroup{\tikz@@oproj@line}{\tikzerror{Ex⌋
    ↪  pected \string\bgroup after
    ↪  oproj}}%
}
```

Just like in Section 2, when we're dealing with path operators whose syntax involve groups/curly braces, we do `\bgroup`. By the way, the following snippet involves parsing the line and coordinates:

[4]My dumbass made it so that `... oproj {(coord2) -- (coord3)} ...` throws an error. Will fix it soon. Apologies!

```
\def\tikz@@oproj@line#1{%
    \edef\tikz@oproj@line{#1}%
    \pgfutil@ifnextchar({\tikz@@oproj@coord}{\tikzerror{Expected
    ↪  parenthesis for coordinate}}%
}
\def\tikz@@oproj@coord(#1){%
    \edef\tikz@oproj@target{#1}%
    \tikz@do@oproj%
}
```

We add another character checking for the coordinate, and yet another error for the dummy that forgot it. These `\edef`s will be essential when we proceed to the construction.

If you think it's time to define `\tikz@do@oproj`, not so fast. We won't do that just yet. We must do some stripping and splitting first before moving on to the calculation and the coordinate definition (which has a lot of trickery involved!).

```
\def\tikz@oproj@split@line#1--#2\@nil{%
    \edef\tikz@oproj@A{\expandafter\strip@parentheses\expandaft
    ↪  er{#1}}%
    \edef\tikz@oproj@B{\expandafter\strip@parentheses\expandaft
    ↪  er{#2}}%
}
```

Recall that `\tikz@oproj@line` holds something like `(coord1)--(coord2)`. This macro basically chops this at the `--`, so `#1` is `(coord1)`, and `#2` is `(coord2)`. After that, we proceed to the following:

```
\def\strip@parentheses#1{%
    \expandafter\strip@parentheses@aux#1\@nil%
}
\def\strip@parentheses@aux(#1)#2\@nil{#1}
```

The `\strip@parentheses` removes the parentheses, leaving just `coord1` or `coord2`.

Now, we can proceed to the calculation! It's really long, so bear with me.

```
\def\tikz@do@oproj{%
    \pgfgetlastxy{\currentx}{\currenty}%
    \expandafter\tikz@oproj@split@line\tikz@oproj@line--\@nil%
    \pgfextract@process\tikz@oproj@A{\tikz@scan@one@point\pgfut
    ↪  il@firstofone(\tikz@oproj@A)}%
    \pgfextract@process\tikz@oproj@B{\tikz@scan@one@point\pgfut
    ↪  il@firstofone(\tikz@oproj@B)}%
```

That first line after the beginning of the group retrieves the x and y coordinates (in sp units) of the most recent path point into the macros you're seeing as `\currentx` and `\currenty` respectively. For reference, we will call this vector C. The second line is where we needed `\tikz@oproj@split@line` for. When you do `oproj {(A)--(B)}`, Ti*k*Z saves the text between the braces as `\tikz@oproj@line`[5]. The split macro peels off everything before and after the `--` path operator, giving you two parenthesized tokens.

Each `\pgfextract@process` runs the coordinate parser on (1,2) and stores the result in the registers `\pgf@x`, `\pgf@y`; we then copy those into `\pgf@xa` and `\pgf@ya` for A, and `\pgf@xb` and `\pgf@yb` for B. So now we have this:

- $A = (x_A, y_A)$
- $B = (x_B, y_B)$
- $C = (x_{\text{current}}, y_{\text{current}})$ (or (x_C, y_C))

Now that we're done with processing coordinates, we proceed with the calculation. We first compute the vectors AB and AC:

```
    \advance\pgf@xb by -\pgf@xa \advance\pgf@yb by -\pgf@ya%
    \pgf@xc=\currentx \advance\pgf@xc by -\pgf@xa%
    \pgf@yc=\currenty \advance\pgf@yc by -\pgf@ya%
```

After this, we denote (x_B, y_B) as the vector $\mathbf{v} = B - A$, and (x_C, y_C) as the vector $\mathbf{w} = C - A$. Now, here comes the messy but satisfying part:

```
    \loop%
        \ifdim\pgf@xb>1pt
            \divide\pgf@xb by2 \divide\pgf@yb by2
            \divide\pgf@xc by2 \divide\pgf@yc by2
```

[5]Please do not confuse this with the earlier macro we tackled, `\tikz@@oproj@line`.

```
        \multiply\c@pgf@counta by2%
    \repeat%
    \loop%
        \ifdim\pgf@xb<-8192pt
            \divide\pgf@xb by2 \divide\pgf@yb by2
            \divide\pgf@xc by2 \divide\pgf@yc by2
            \multiply\c@pgf@counta by2%
    \repeat%
```

TEX's arithmetic uses fixed-point dimensions and chokes if you do huge multiplies or squares. So there are two `\loop...\repeat` blocks that repeatedly halve both $\mathbf{v}$ and $\mathbf{w}$ (and count how many times they've halved) until $\mathbf{v}$ is safely within about ± 8192 pt. The integer `\c@pgf@counta` keeps track of the total factor of 2 we removed.

Now, we compute the dot product $\mathbf{w} \cdot \mathbf{v}$ and the length of $\mathbf{v}$ squared $\|\mathbf{v}\|^2$:

```
    \pgfmathparse{(\pgf@sys@tonumber{\pgf@xc})*(\pgf@sys@tonumb⌋
    ↪  er{\pgf@xb}) +
    ↪  (\pgf@sys@tonumber{\pgf@yc})*(\pgf@sys@tonumber{\pgf@yb⌋
    ↪  })}%
    \let\dotprod=\pgfmathresult%
    \pgfmathparse{(\pgf@sys@tonumber{\pgf@xb})^2 +
    ↪  (\pgf@sys@tonumber{\pgf@yb})^2}%
    \let\lensq=\pgfmathresult%
```

The macro `\pgf@sys@tonumber` is used to convert the stored x and y values to numbers so that PGF can calculate it properly. We add error checking for when `\lensq` is equal to 0. We can't have a line that is 0pt long, then that would be devastating.

```
    \ifdim\lensq pt=0pt%
        \tikzerror{Line segment has zero length}%
    \else%
        \pgfmathparse{\dotprod/\lensq}%
        \pgf@x=\pgf@xa%
        \pgf@y=\pgf@ya%
        \advance\pgf@x by
        ↪  \dimexpr\pgfmathresult\pgf@xb*\c@pgf@counta\relax%
        \advance\pgf@y by
        ↪  \dimexpr\pgfmathresult\pgf@yb*\c@pgf@counta\relax%
```

```
        \expandafter\pgfcoordinate\expandafter{\tikz@oproj@targ⌋
        ↪  et}{\pgfqpoint{\pgf@x}{\pgf@y}}%
    \fi%
    \tikz@scan@next@command
}
```

If `\lensq` is not zero, then great, we compute the projection scalar. This is expressed mathematically as

$$t = \frac{\mathbf{w} \cdot \mathbf{v}}{\|\mathbf{v}\|^2}.$$

You weren't expecting math, yet here we are. In coding terms, we start at A's coordinates, then move `t(original B-A)`. We multiply back by `\c@pgf@counta` to undo all our earlier halvings, so we get exactly the correct point. Finally, we name our coordinate as usual and scan for the next command.

Actually, before moving on to the full code, maybe I should explain in more detail how we just bypassed TEX's arithmetic limits when we were constructing the `oproj` operator. As a quick refresher, TEX uses fixed-point arithmetic (not floating-point), and its dimension registers (`\dimen`, `\pgf@x`, etc.) can only safely represent values in the approximate range of $-16,384$ to $16,384$ pt. So if you square or multiply large values (like long vectors), TEX can easily overflow or crash, resulting in "Dimension too large" errors. I myself annoyingly dealt with this quite a few times but thankfully I found a clever workaround.

Go back to the two `\loop...\repeat` blocks. That does automatic normalization. If $\mathbf{v}$ is too large in absolute value, it halves all involved vectors until they're small enough for safe math. Every time it halves, it keeps track of how many times with `\c@pgf@counta`. This means instead of computing

$$P = A + t(B - A),$$

we compute:

$$P = A + t \cdot \mathbf{v}_s \cdot S$$

where $\mathbf{v}_s$ is the scaled down vector and S is the scaling factor, which in code is `\c@pgf@counta`. Now, with that, all intermediate values are small enough to avoid overflow, and at the end you multiply back by S so the final result is correct.

Now, one might ask, "how on earth does this magically work?", well, you just witnessed the wonders of TEX. TEX's `\divide`, `\multiply`, and `\advance` are precise (within range). PGF math can handle floats via macros

like \pgfmathparse, those don't overflow because they happen inside a custom math parser. The real danger is mixing large \pgf@x values with \dimexpr or squaring, which is what this avoids by scaling first. It's even crazy when you realize Ti*k*Z wasn't designed for arbitrary vector math, PGF doesn't protect you from overflow, and TEX has no double type, just scaled points and macros.

Now, as promised, here's the full code. I included the updated \tikz@handle in case you forgot or don't know how to hook the letter "o" and \tikz@ochar.

```
\def\tikz@handle{%
    \pgfutil@switch\pgfutil@ifx\pgf@let@token{%
        {(}{\let\pgfutil@next\tikz@movetoabs}%
        {+}{\let\pgfutil@next\tikz@movetorel}%
        {-}{\let\pgfutil@next\tikz@lineto}%
        {.}{\let\pgfutil@next\tikz@dot}%
        {r}{\let\pgfutil@next\tikz@rect}%
        {n}{\let\pgfutil@next\tikz@fig}%
        {[]}{\let\pgfutil@next\tikz@parse@options}%
        {c}{\let\pgfutil@next\tikz@cchar}%
        {\bgroup}{\let\pgfutil@next\tikz@beginscope}%
        {\egroup}{\let\pgfutil@next\tikz@endscope}%
        {;}{\let\pgfutil@next\tikz@finish}%
        {a}{\let\pgfutil@next\tikz@a@char}%
        {e}{\let\pgfutil@next\tikz@e@char}%
        {g}{\let\pgfutil@next\tikz@g@char}%
        {s}{\let\pgfutil@next\tikz@schar}%
        {|}{\let\pgfutil@next\tikz@vh@lineto}%
        {p}{\pgfsetmovetofirstplotpoint\let\pgfutil@next\tikz@p
        ↪  char}%
        {t}{\let\pgfutil@next\tikz@to}%
        {\pgfextra}{\let\pgfutil@next\tikz@extra}%
        {\foreach}{\let\pgfutil@next\tikz@foreach}%
        {f}{\let\pgfutil@next\tikz@fchar}%
        {\pgf@stop}{\let\pgfutil@next\relax}%
        {\par}{\let\pgfutil@next\tikz@scan@next@command}%
        {d}{\let\pgfutil@next\tikz@decoration}%
        {l}{\let\pgfutil@next\tikz@l@char}%
        {:}{\let\pgfutil@next\tikz@colon@char}%
        {\relax}{\relax\let\pgfutil@next\tikz@scan@next@command
        ↪  }%
        {m}{\let\pgfutil@next\tikz@mchar}%
```

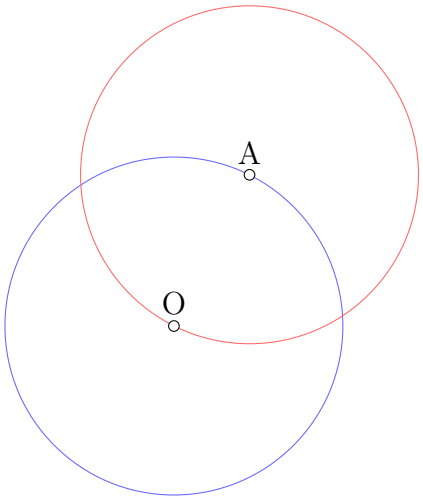
```
        {o}{\let\pgfutil@next\tikz@ochar}%
    }{\tikz@resetexpandcount\pgfutil@next}{\tikz@expand}%
}

\def\tikz@ochar{%
    \pgfutil@ifnextchar p{\tikz@oproj}{\tikz@expand}%
}

\def\tikz@oproj proj{%
    \tikz@flush@moveto
    \tikz@@oproj%
}

\def\tikz@@oproj{%
    \pgfutil@ifnextchar\bgroup{\tikz@@oproj@line}{\tikzerror{Ex
    ↪  pected \string\bgroup after
    ↪  oproj}}%
}
\def\tikz@@oproj@line#1{%
    \edef\tikz@oproj@line{#1}%
    \pgfutil@ifnextchar({\tikz@@oproj@coord}{\tikzerror{Expected
    ↪  parenthesis for coordinate}}%
}
\def\tikz@@oproj@coord(#1){%
    \edef\tikz@oproj@target{#1}%
    \tikz@do@oproj%
}

\def\tikz@oproj@split@line#1--#2\@nil{%
    \edef\tikz@oproj@A{\expandafter\strip@parentheses\expandaft
    ↪  er{#1}}%
    \edef\tikz@oproj@B{\expandafter\strip@parentheses\expandaft
    ↪  er{#2}}%
}

\def\strip@parentheses#1{%
    \expandafter\strip@parentheses@aux#1\@nil%
}
\def\strip@parentheses@aux(#1)#2\@nil{#1}

\def\tikz@do@oproj{%
    \pgfgetlastxy{\currentx}{\currenty}%
```

```
\expandafter\tikz@oproj@split@line\tikz@oproj@line--\@nil%
\pgfextract@process\tikz@oproj@A{\tikz@scan@one@point\pgfut
↪  il@firstofone(\tikz@oproj@A)}%
\pgfextract@process\tikz@oproj@B{\tikz@scan@one@point\pgfut
↪  il@firstofone(\tikz@oproj@B)}%
\pgf@process{\tikz@oproj@A}%
\pgf@xa=\pgf@x \pgf@ya=\pgf@y%
\pgf@process{\tikz@oproj@B}%
\pgf@xb=\pgf@x \pgf@yb=\pgf@y%
\advance\pgf@xb by -\pgf@xa \advance\pgf@yb by -\pgf@ya%
\pgf@xc=\currentx \advance\pgf@xc by -\pgf@xa%
\pgf@yc=\currenty \advance\pgf@yc by -\pgf@ya%
\c@pgf@counta=1%
\loop%
    \ifdim\pgf@xb>1pt
        \divide\pgf@xb by2 \divide\pgf@yb by2
        \divide\pgf@xc by2 \divide\pgf@yc by2
        \multiply\c@pgf@counta by2%
\repeat%
\loop%
    \ifdim\pgf@xb<-8192pt
        \divide\pgf@xb by2 \divide\pgf@yb by2
        \divide\pgf@xc by2 \divide\pgf@yc by2
        \multiply\c@pgf@counta by2%
\repeat%
\pgfmathparse{(\pgf@sys@tonumber{\pgf@xc})*(\pgf@sys@tonumb
↪  er{\pgf@xb}) +
↪  (\pgf@sys@tonumber{\pgf@yc})*(\pgf@sys@tonumber{\pgf@yb
↪  })}%
\let\dotprod=\pgfmathresult%
\pgfmathparse{(\pgf@sys@tonumber{\pgf@xb})^2 +
↪  (\pgf@sys@tonumber{\pgf@yb})^2}%
\let\lensq=\pgfmathresult%
\ifdim\lensq pt=0pt%
    \tikzerror{Line segment has zero length}%
\else%
    \pgfmathparse{\dotprod/\lensq}%
    \pgf@x=\pgf@xa%
    \pgf@y=\pgf@ya%
    \advance\pgf@x by
    ↪  \dimexpr\pgfmathresult\pgf@xb*\c@pgf@counta\relax%
```

```
        \advance\pgf@y by
        ↪  \dimexpr\pgfmathresult\pgf@yb*\c@pgf@counta\relax%
        \expandafter\pgfcoordinate\expandafter{\tikz@oproj@targ⌋
        ↪  et}{\pgfqpoint{\pgf@x}{\pgf@y}}%
    \fi%
    \tikz@scan@next@command%
}
```

5 Circles and Coordinates

To finish things off, we're gonna do something different in this section. Instead of creating a new path operator, we will be modifying an existing one. The culprit is `circle`. Now, you might ask, "what are you gonna do with `circle`?". Well, I'll make it accept coordinates as the radius. The `tkz-euclide` package has this feature, but Ti*k*Z doesn't. Ridiculous.




```
\begin{tikzpicture}
    \coordinate (O) at (0,0);
    \coordinate (A) at (1,2);

    \draw[blue!60] (O) circle (A);
    \draw[red!60] (A) circle (O);

    \foreach \p in {O,A} {
        \draw[fill=white] (\p) circle (2pt) node[above] {\p};
```

```
    }
\end{tikzpicture}
```

Honestly, Ti*k*Z could've added this feature, but here we are now. Usually the `tkz-euclide` way of replicating the above snippet would be something like this:

```
\begin{tikzpicture}
    \tkzDefPoints{0/0/O,1/2/A}
    \tkzDrawCircle[blue!60](O,A)
    \tkzDrawCircle[red!60](A,O)
    \tkzLabelPoints[above](O,A)
\end{tikzpicture}
```

or, in Ti*k*Z, you can do this with the `calc` library:

```
\begin{tikzpicture}
    \coordinate (O) at (0,0);
    \coordinate (A) at (1,2);

    \draw let \n2 = {veclen(\x1,y1)} in
        (O) circle (\n2)
        (A) circle (\n2);

    \foreach \p in {O,A} {
        \draw[fill=white] (\p) circle (2pt) node[above] {\p};
    }
\end{tikzpicture}
```

But nah, I don't want to use either of them.

5.1 In the Eyes of the Parser

Before we modify the path operator, we must first understand how Ti*k*Z "understands" `circle`. Under the hood, it's handled by the macro `\tikz@cchar`:

```
\def\tikz@cchar{%
    \pgfutil@ifnextchar i{\tikz@circle}{
        \pgfutil@ifnextchar h{\tikz@children}{\tikz@cochar}
```

```
    }
}%
```

If it detects “i” as the next character after “c”, it moves on to `\tikz@circle`:

```
\def\tikz@circle ircle{\tikz@flush@moveto\tikz@@circle}%
```

Just like in our path operators from the previous sections, the definition contains the macro `\tikz@flush@moveto` and `\tikz@@circle`. If you forgot what the former command does then go look at Section 1. Moving on, `tikz.code.tex` will then proceed to:

```
\def\tikz@@circle{%
    \let\tikz@tangent\relax%
    \pgfutil@ifnextchar(\tikz@@@circle
    {\pgfutil@ifnextchar[\tikz@circle@opt{%])
        \advance\tikz@expandcount by -10\relax% go down quickly
        \ifnum\tikz@expandcount<0\relax%
            \let\pgfutil@next=\tikz@@circle@normal%
        \else%
            \let\pgfutil@next=\tikz@@circle@scanexpand%
        \fi%
    \pgfutil@next%
    }}%
}%
\def\tikz@@circle@scanexpand{\expandafter\tikz@@circle}%
\def\tikz@@circle@normal{\tikz@circle@opt[]}%
```

Basically, if it detects a parenthesis, it moves to `\tikz@@@circle`, or if it sees a bracket, it goes to `\tikz@circle@opt[...]`, otherwise it decrements an ”expand count” and either re-loops (which is what the macro `\tikz@@circle@scanexpand` does) or falls back to empty option mode (which is what `\tikz@@circle@normal` does). Ti*k*Z handles the new syntax as follows:

```
\def\tikz@circle@opt[#1]{%
    {%
        \def\tikz@node@at{\tikz@last@position}%
        \let\tikz@transform=\pgfutil@empty%
```

```
        \tikzset{every circle/.try,#1}%
        \pgftransformshift{\tikz@node@at}%
        \tikz@transform%
        \tikz@do@ellipse{\pgfkeysvalueof{/tikz/x
        ↪  radius}}{\pgfkeysvalueof{/tikz/y radius}}
    }%
    \tikz@scan@next@command%
}%
```

In this branch, Ti*k*Z expects you've done something like a bracket after `circle`, like `... circle[radius=1cm]`. It grabs `/tikz/x radius` and `/tikz/y radius` (usually set by the `radius` key), shifts to the center point, applies any transforms, and calls the macro `\tikz@do@ellipse{<x>}{<y>}`. Note that a circle is just an ellipse with equal radii, if you hadn't realized that by now.

For the deprecated syntax (`... circle (<r>)`), Ti*k*Z has `\tikz@@@circle` to handle that:

```
\def\tikz@@@circle(#1){%
    {%
        \pgftransformshift{\tikz@last@position}%
        \pgfutil@in@{ and }{#1}%
        \ifpgfutil@in@%
            \tikz@@ellipseB(#1)%
        \else%
            \tikz@do@circle{#1}%
        \fi%
    }%
    \tikz@scan@next@command%
}%
```

This then moves on to either `\tikz@@ellipse@B` when it sees an "and" after a dimension, or `\tikz@do@circle` for pure radius, like `circle (1cm)`. We won't be dealing with ellipses, so we'll proceed to `\tikz@do@circle`:

```
\def\tikz@do@circle#1{%
    \pgfmathparse{#1}%
    \let\tikz@ellipse@x=\pgfmathresult
    \ifpgfmathunitsdeclared
        \pgfpathellipse{\pgfpointorigin}%
```

```
            {\pgfqpoint{\tikz@ellipse@x pt}{0pt}}%
      {\pgfpoint{0pt}{\tikz@ellipse@x pt}}%
    \else
        \pgfpathellipse{\pgfpointorigin}%
            {\pgfpointxy{\tikz@ellipse@x}{0}}%
            {\pgfpointxy{0}{\tikz@ellipse@x}}%
    \fi
}
```

The top line evaluates the radius expression, which is either dimensionless or with units. Then, it stores the result in `\tikz@ellipse@x`. If you're familiar with PGF, then you might recognize `\pgfpathellipse` as a macro that draws an ellipse. In the definition above, it draws the ellipse centered at the current origin (shifted earlier) with x-radius and y-radius both equal to `\tikz@ellipse@x`.

That's about it. Now, for our modification, our target will be the macro `\tikz@do@circle`, since it parses and draws the circle. Our goal is to hook a handler for coordinates, so Ti*k*Z won't complain about some arithmetic error when it sees for example `circle (A)`.

5.2 Implementation

Our goal is this: the handler will check if the argument is a coordinate first. If there's no coordinate with that name (in other words undefined), then you're in the plain radius case where it tries to parse it numerically. If it is defined, then we will fetch the node's center, compute the vector difference from the origin, take its length, and use that as the radius.

The first step would be to find a macro that is defined when naming a node. If you read `tikz.code.tex` like bedtime stories, then you'll know we're going to use `\pgf@sh@ns<name>`. With that in mind, our modification will now look something like this:

```
\def\tikz@do@circle#1{%
    \pgfutil@ifundefined{pgf@sh@ns@#1}{%
        % first 2 lines of original \tikz@do@circle
    }{%
        % something that handles coordinate case
    }%
    % the rest of the original \tikz@do@circle
}
```

We'll now insert the plain radius case and the rest of the original definition code:

```
\def\tikz@do@circle#1{%
    \pgfutil@ifundefined{pgf@sh@ns@#1}{%
        \pgfmathparse{#1}%
        \let\tikz@ellipse@x=\pgfmathresult
    }{%
        % something that handles coordinate case
    }%
    \pgfpathellipse{\pgfpointorigin}%
        {\pgfqpoint{\tikz@ellipse@x pt}{0pt}}%
        {\pgfpoint{0pt}{\tikz@ellipse@x pt}}%
}
```

Now, the slightly tricky part would be to handle the case where our argument is a coordinate, but all it takes is 3 lines of code and $3 \cdot 10^9$ cups of coffee.

To fetch the node center, we usually use some macro from PGF like `\pgfpointanchor{<node>}{center}`, then calculate the vector distance from the origin. So, something like this will suffice:

```
\pgfpointdiff{\pgfpointorigin}{\pgfpointanchor{#1}{center}}%
```

One line typed, two more to go. The next step would be to figure out the result of that operation, then use it as the length. Once we have the length, we will use that as the radius. One can do this with `\pgfmathveclen`, so let's place that real quick:

```
\pgfmathveclen{\pgf@x}{\pgf@y}%
```

Nice. Now, since after the "if undefined" statement we used `\tikz@ellipse@x`, our last line would therefore be:

```
\let\tikz@ellipse@x=\pgfmathresult
```

And that's basically it. Congrats! You just made Ti*k*Z accept coordinates as the radius of a circle. Now, the full code would be this:

```
\def\tikz@do@circle#1{%
    \pgfutil@ifundefined{pgf@sh@ns@#1}{%
        \pgfmathparse{#1}%
        \let\tikz@ellipse@x=\pgfmathresult
    }{%
        \pgfpointdiff{\pgfpointorigin}{\pgfpointanchor{#1}{cent
        ↪  er}}%
        \pgfmathveclen{\pgf@x}{\pgf@y}%
        \let\tikz@ellipse@x=\pgfmathresult
    }%
    \pgfpathellipse{\pgfpointorigin}%
        {\pgfqpoint{\tikz@ellipse@x pt}{0pt}}%
        {\pgfpoint{0pt}{\tikz@ellipse@x pt}}%
}
```

6 Conclusion

That was a wild ride through TEX and Ti*k*Z's parser. Honestly, I still find it baffling how complicated it is. Even with all this I feel like I merely scratched the surface. However, this is only the beginning. I might do some more of this soon.