10
Calculating with Coordinates and Paths

In *Chapter 1*, *Creating the First TikZ Images*, we started off using explicit values to choose coordinates. We achieved relative positioning by manually giving distances while drawing a path.

Now, we are about to take things to the next level by introducing a whole new set of techniques for calculating coordinates from other coordinates. We can add or subtract coordinates from each other, calculate a coordinate position between other coordinates at a certain distance, find a coordinate as a projection onto a line, and rotate coordinates.

And that's not all – we'll introduce loop commands that help repeat calculations and actions.

Get ready to dive deep into the following techniques:

- Repeating in loops
- Calculating with coordinates
- Evaluating loop variables
- Calculating intersections of paths

By the end of this chapter, you will be ultra-efficient in repeating similar commands and using calculations for perfect node and edge placement.

Technical requirements

The source code of the chapter's examples is available at `https://tikz.org/chapter-10`. The code can be downloaded from GitHub at `https://github.com/PacktPublishing/LaTeX-graphics-with-TikZ/tree/main/10-calculating-transforming`.

In this chapter, we will use the `calc` and `intersections` TikZ libraries; other features are loaded by default.

Repeating in loops

The easiest calculation is counting, so this will be our starting point. In a **for loop**, TikZ can count with a **variable** for us while it repeats a code segment using the variable. While this sounds simple, it's tremendously valuable for generating graphics with ease, especially with the TikZ `\foreach` command, which is incredibly flexible.

The basic syntax of this command is the following:

```
\foreach variable in {list of values} {commands};
```

Let's break down the highlighted code:

- `variable`: We name and use it like a macro, such as `\i`. The convention of using *i* as a loop variable dates back to the early programming languages and mathematics, when *x* and *y* were used for variables and *i* and *j* were used as indexing counters. However, we are free to choose any name as long as it starts with a backslash.
- `list of values`: This is a comma-separated list of values, such as `1,2,3`. You can omit values and write – for example, `1,...,10` – and then TikZ implicitly fills in the missing values – here, all numbers from 1 to 10. When you give more values, TikZ calculates the difference and uses it for filling in. So, with `2,4,...,10`, TikZ uses the even numbers until 10. That auto-filling works even with fractional steps, such as `0.1,0.2,...,1`. Plus, you can use alphabetic character sequences and patterns such as `A_1,...,F_1`.
- `commands`: This can be a sequence of commands that use the variable. If you use a single command, you can skip the braces around it.

Let's look at the syntax with some real examples. When we take the grid from *Figure 2.1*, we can add labels to the *x* axis like this:

```
\foreach \i in {-3,-2,-1,1,2,3} \node at (\i,-0.2) {\i};
```

As we can use several commands in a single loop, we can add *x* and *y* labels at the same time:

```
\foreach \i in {-3,-2,-1,1,2,3} {
  \node at (\i,-0.2) {\i};
  \node at (-0.2,\i) {\i};
}
```

Now our grid looks like this:

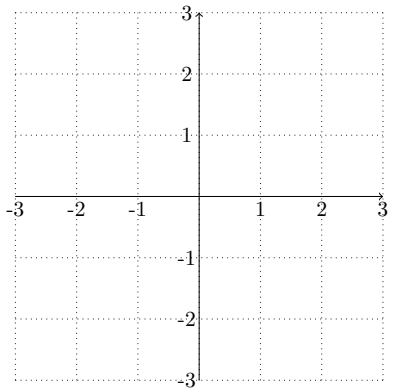

Figure 10.1 – A grid with axis labels

That's useful when we use this draft helper grid in more extensive drawings with a larger grid.

By using the dots auto-filling feature, we can write it shorter, such as the following:

```
\foreach \i in {-3,...,3} \node at (\i,-0.2) {\i};
```

This would, of course, also include the value 0.

Let's also see how the auto-filling of omitted values works. We want to draw 36 circles at a distance of 1 to the origin, every 10 degrees between 10 and 360 degrees. It's sufficient to tell TikZ to start the loop with 10, proceed with 20, and continue that way until 360 is reached:

```
\foreach \i in {10,20,...,360} \draw (\i:1) circle (1);
```

That single line gives us a set of circles with nice symmetry:

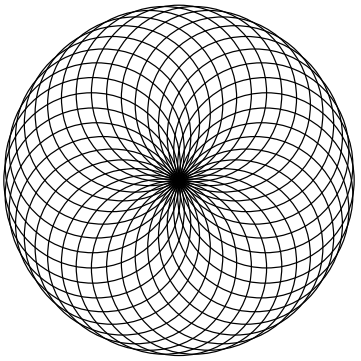

Figure 10.2 – Rotated circles

In the code line for *Figure 10.2*, we got 36 separate drawing paths. Let's say we want to fill all these circles, not simply black but alternating black and white. Remember the `even odd` filling rule from *Chapter 7*? We can apply that filling when we turn this example into a single path. Luckily, we can use `\foreach` within a single path. We can change our example as follows:

```
\filldraw[even odd rule] \foreach \i in {10,20,...,360}
  {(\i:1) circle (1)};
```

With the `even odd` filling rule, adjacent areas of a self-intersecting path have different colors, and so we get this amazing pattern as a result:

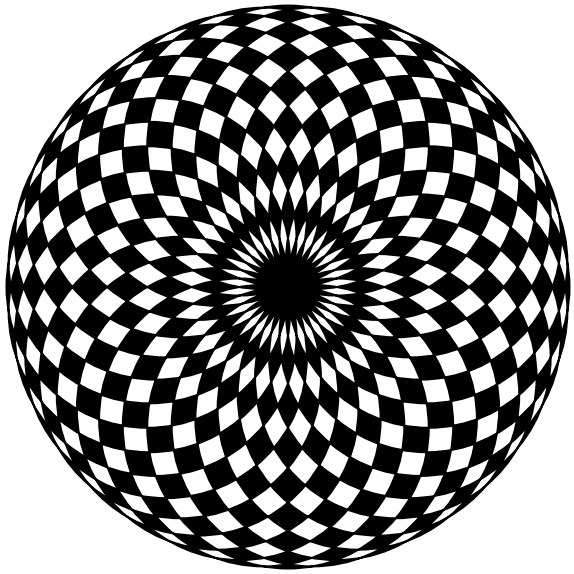

Figure 10.3 – Filled intersecting circles

You can see that with a single `\foreach` line, you can draw a lot with many iterations.

`\foreach` can take several loop variables and values, separated by forward slashes, as follows:

```
\foreach \i/\j in {A/1,B/2,C/3} \node at (\j,-0.2) {\i};
```

This prints **A**, **B**, and **C** instead of 1, 2, and 3 at the *x* axis shown in *Figure 10.1*, as follows:

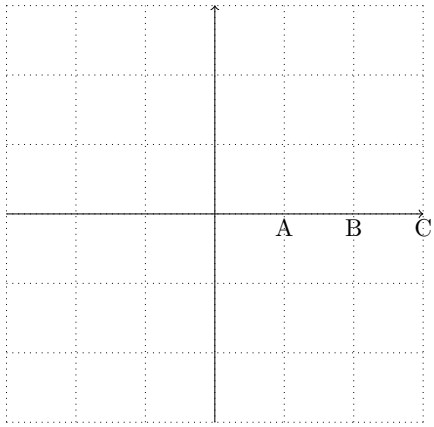

Figure 10.4 – Alphanumeric labels

Whenever you have a command that you would like to execute repeatedly for a specific set of values, pairs, or triples, you can use the `\foreach` command. And here's the best part – it's more than just handy in TikZ; you can use `\foreach` directly within LaTeX. To use it without loading TikZ, all you need to do is to load the `pgffor` package where it's defined.

Now that we've covered the basics of the `\foreach` command, let's quickly advance to the next topic because, in the following sections, we can see further examples. Loops are even more powerful when we combine them with calculations, and that's what we'll be exploring next.

Calculating with coordinates

The `calc` library allows us basic operations with coordinates. Load it first in the document preamble with `\usetikzlibrary{calc}`, and you're ready to go.

TikZ can now calculate what we write between `$` symbols within a coordinate. While it may look like TeX inline math mode, it actually enables us to perform calculations and math operations.

Adding and subtracting coordinates

With just the simple notation of `$(A)+(B)$` we can add two coordinates. How can this be useful? It's an easy relative positioning when we use a particular coordinate and add a coordinate to have a shift in the *x* and *y* directions.

Let's start with a coordinate, `A`, at some arbitrary position and build what we can see in *Figure 10.5*:

```
\coordinate (A) at (1,2);
```

Now, we can create a coordinate that's just right of it, with an *x* distance of `1`, by adding a coordinate with suitable values, *x*=`1` and *y*=`0`:

```
\coordinate (B) at ($(A)+(1,0)$);
```

Note that we also have parentheses around the `$...$` expression to indicate that it's a coordinate. Admittedly, the nesting of parentheses and `$` symbols looks messy, but it's well structured.

We can also use polar coordinates. The following line creates a coordinate, `C`, which also has a distance of `1` to `A`, but with an angle of 60 degrees to the *x* axis, by adding a coordinate in polar notation:

```
\coordinate (C) at ($(A)+(60:1)$);
```

When we connect `A`, `B`, and `C`, we get an equilateral triangle:

```
\draw (A) -- (B) -- (C) -- cycle;
```

In the same way, we can subtract coordinates from each other. Furthermore, we can insert a factor expression before coordinates with a `*` symbol, which can be a number or even a more or less complex computation, such as in `2*(A)`, `sqrt(3)*(2,2)`. Such a calculation can be performed like this:

```
\coordinate (D) at ($sin(60)*sqrt(2)*(A)+0.5*(60:1)$);
```

You may rarely need it, but it's good to know that math tools are available if you need them.

Computing points between coordinates

We can save effort and let TikZ calculate the position of points between two coordinates. The basic syntax is like `(A)!factor!(B)`, which gives a coordinate on the line between `A` and `B`, with the factor between `0` and `1` deciding where – `(A)!0.1!(B)` is close to `A`, `(A)!0.9!(B)` is close to `B`, and `(A)!0.5!(B)` is precisely the midpoint between `A` and `B`. A factor of `0` would simply equate to `A`, and a factor of `1` would be equivalent to `B`.

We are allowed to use negative values and higher values than `1`; in that case, the resulting point will still be on the line between `A` and `B`, but not next to them. When we use negative factors, the new coordinate will have the same distance from `A` as the corresponding positive factor but lie in the other direction on the line, away from both `A` and `B`. So, `(A)!2!(B)` would be twice as far from `A` as `B` is.

TikZ calls this kind of expression a **partway modifier**.

To see the syntax in use, we can draw the inscribed circle of the `ABC` triangle from the previous section in *Figure 10.5*. A bit of math research reveals that the radius shall equal `sqrt(3)/6`; we use that fact to draw the circle above the middle point between `A` and `B`:

```
\draw ($(A)!0.5!(B)+(0,{sqrt(3)/6})$) circle({sqrt(3)/6});
```

Note how we used curly braces to encapsulate the math expression. Generally, adding braces helps us when the parser gets confused by additional syntax because, as you can see, we can have pretty complex math expressions. Here, it was particularly needed because the parser expects parentheses for coordinates and would get confused by the parentheses of the `sqrt` function.

This is the result of our drawing with calculated coordinates in this section:

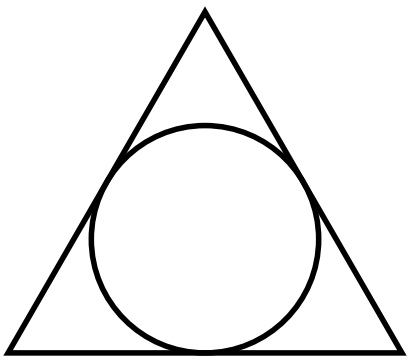

Figure 10.5 – A triangle with an inscribed circle

Instead of a factor, we can use a dimension; otherwise, the syntax stays the same. So, `(A)!1cm!(B)` is the point on the line between `A` and `B` with a distance of 1 cm. Similarly, `(A)!-1cm!(B)` is the point on the line connecting `A` and `B`, which is not between them but on the other side of `A`, with a distance of 1 cm. That's straightforward and called a **distance modifier**.

Projecting on a line

The third expression with very similar syntax is the **projection modifier**. Instead of a factor or a distance, we can insert a third coordinate. Let's say the third coordinate is `C`; then, `(A)!(C)!(B)` is the orthogonal projection from `C` onto the line connecting `A` and `B`. It doesn't have to be between `A` and `B`.

Here, you can see it in action with the previous example, drawing a dotted line from `C` to the orthogonal projection from `C` on the line between `A` and `B`:

```
\draw[densely dotted] (C) -- ($(A)!(C)!(B)$);
```

Figure 10.5 with the additional line now looks like this:

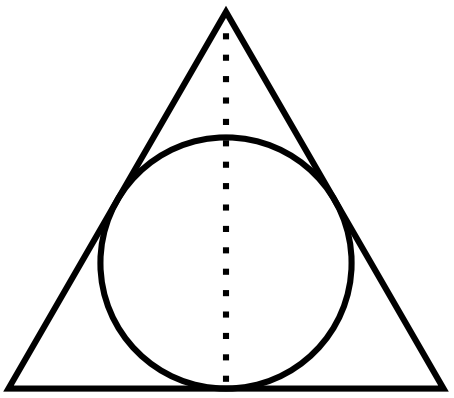

Figure 10.6 – The projection on a line

Of course, you can combine a projection with further calculations, such as factors and angles, which we will see in the next section.

Adding angles

With all modifiers, we can insert an angle. That's a value in degrees prefixed to the second coordinate, separated by a colon. At first, the line from `A` and `B` would be rotated by that angle around `A`, and then the modifier would be applied.

So, with our example, the full expression, `($(A)!0.5!60:(B)$)`, equals the coordinate right in the middle between `A` and `B`, rotated by 60 degrees around `A`.

We can apply it to our equilateral triangle in the following way:

```
\filldraw ($(A)!0.5!60:(B)$) circle (0.03);
```

As each angle of the triangle happens to be 60 degrees, it's equal to the middle point between `A` and `C`, as we can see here:

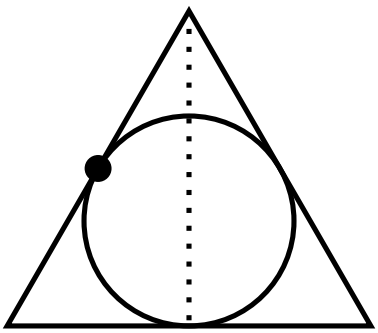

Figure 10.7 – Using a partway modifier with an angle

Let's practice the partway modifier with angles and a `\foreach` loop to get a glimpse of practical code. We will arrange circles in an Archimedean spiral. That is a spiral of polar coordinates where the radius is proportional to the angle. We start with an angle of zero, having a radius of zero. At half of the full 360 degrees, we have a radius of 0.5. At 360 degrees, we have a radius of 1. That can continue, so with 720 degrees, we will have a radius of 2, and so on.

We will make a `\foreach` loop with tiny steps to get small circles; we will iterate over a `\i` variable that shall be the fraction of the angle. `\i` will be our partway modifier between the origin, `(0,0)`, and the coordinate, `(1,0)`. `\i` will also serve as the fraction of 360 degrees. The circle radius shall also grow with `\i`; we will add a suitable factor so that it's small enough. That's the plan, and now here's the code; try to understand it with the preceding explanation:

```
\foreach \i in {0,0.025,...,1}
  \draw ($(0,0)!\i!\i*360:(1,0)$) circle(0.08*\i);
```

We get one rotation of the spiral:

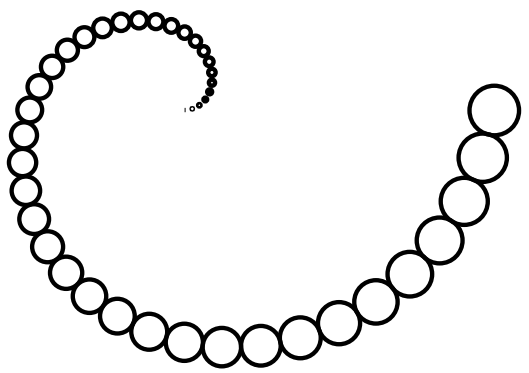

Figure 10.8 – A spiral of circles

To see the six spiral rotations, we can let `\i` run in the loop until `6`. Add shading to get colored balls:

```
\foreach \i in {0,0.025,...,6}
  \draw[shading=ball] ($(0,0)!\i!\i*360:(1,0)$)
  circle(0.08*\i);
```

Without much work, we get an impressive output thanks to the loop:

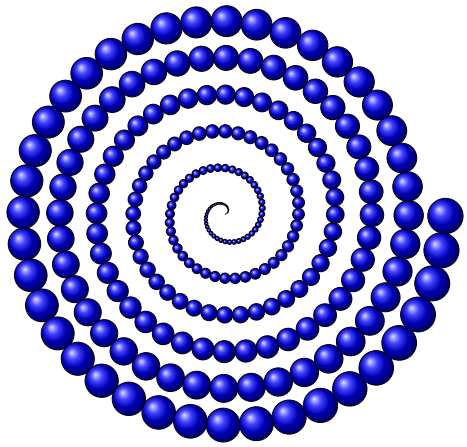

Figure 10.9 – A spiral of balls

In the next section, we will see how to calculate within `\foreach` options.

Evaluating loop variables

Let's take a closer look at the code for *Figure 10.8*. Could there possibly be even more variable options to adjust? How about iterating colors and interconnecting between loop repetitions? Yes, you can achieve this within the same loop. So, let's look at further advanced `\foreach` options – at first, some syntax with short examples, and then a more extended example.

Counting loop repetitions

A `\foreach` value list can contain alphanumeric values and patterns to be iterated through and utilized within the loop commands. However, we may want to use values based on their position in the list, such as using the `(\j,0)` coordinate at position `j` in the iterative order of list values.

This is where the `count` option comes into play. Let's say we have `\i` as the loop variable iterating through letters. We introduce the `\j` counter as follows:

```
\foreach \i [count=\j] in {A,...,Z} {commands};
```

Now, while `\i` iterates from `A` to `Z`, `\j` goes from 1 to 26, and we can use both `\i` and `\j` in the commands.

We don't have to start with 1. By writing `count=\j from 10`, we let `\j` begin at 10 instead of 1.

Of course, we can choose any name instead of `\i` and `\j`.

Evaluating the loop variable

As the loop variable can be some pattern, it is not evaluated to a number by default. It is used in the commands as it is, without pre-calculating its value. We can force it or even do a complicated custom computation. It works as follows; again, `\i` and `\j` are chosen as names:

```
\foreach \i [evaluate=\i as macro using formula]
  in {values} {commands};
```

`macro` is our additional variable name, such as `\j`, and `formula` can be a math expression.

If we only say `evaluate=\i`, that value is used when we use `\i` in the commands. If we only say `evaluate=\i as \j`, then `\i` stays as its original pattern, and `\j` is the evaluated value for using both in the commands.

When we use the full syntax with `formula`, this formula will be used for evaluation `\j`, with some math expression applied to `\i`. We will practice it at the end of this section.

Remembering the loop variable

When we have any repetition in a loop, we may want to remember the variable value from the previous repetition, such as to connect points. That's done as follows:

```
\foreach \i [remember=\i as macro initially value]
  in {values} {commands};
```

`macro` can be a name that we choose, such as `\j`. Now, `\j` will have the value of `\i` from the previous repetition. The initial `value` is the value at the first repetition when there is no last value.

To practice these evaluations, we will modify the example for *Figure 10.8* as follows:

```
\foreach \i [remember=\i as \j (initially 6),
  evaluate=\i as \c using 20*\i] in {5.95,5.9,...,0}
  \fill[fill=black!60!blue!\c!white]
    ($(0,0)!\i!\i*180:(1,0)$) --
    ($(0,0)!\j!\j*180:(1,0)$) -- (0,0);
```

The following happens:

- `\i` is our loop variable, this time demonstrating that we can do negative steps. We do tiny steps of 0.05, starting lower than 6 and going to 0.
- `\j` is the remembered previous value of `\i` in each loop repetition, starting from 6.
- `\c` is the color we use for filling the evaluation based on the value of `\i`; the blue value gets lighter in each repetition.
- The loop commands fill a triangle based on the `\i` and `\j` values as corners, with the origin `(0,0)` as the third corner, in a calculation like that shown in *Figure 10.8*.

We get the following as output:

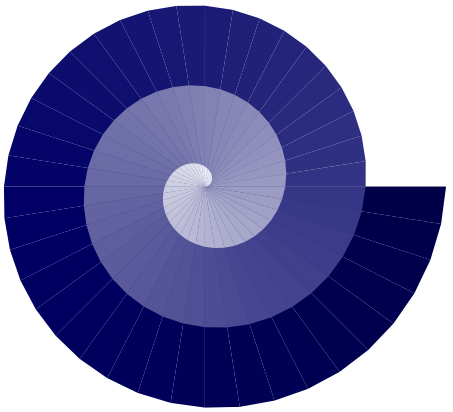

Figure 10.10 – A colored segmented spiral

That was already pretty complex. If you need even more flexibility, remember that you can have several loop variables and nest the `\foreach` loops. Experiment with it, and share your examples on the TikZ community gallery site: `https://tikz.net`.

In the next and final section, we will generate coordinates from existing paths.

Calculating intersections of paths

TikZ drawings are often built step by step. We choose coordinates and draw lines, curves, and shapes. At some point, we may need to know the intersection of such paths to proceed with further drawing steps, such as adding text or arrows at such positions.

We could calculate the intersection point of two lines ourselves by solving a system of two linear equations. To get the intersection points of a circle and a line, we can solve a quadratic equation. Remember polygons or shapes consisting of curvy paths such as bent lines? It can become hard to compute a point on such a path that overlaps with another path.

TikZ provides the `intersections` library that solves such challenges. You can load it in the usual way:

```
\usetikzlibrary{intersections}
```

Now, TikZ can do all the hard work and calculate all intersection points of arbitrary paths, generating named coordinates for them.

Let's dive into a basic example to see how it works. We'll need to use **named paths**, which means we will declare the name as an option for each path. The following code draws two lines called `l1` and `l2`:

```
\draw[name path = l1] (-2,-2) -- (3,3);
\draw[name path = l2] (-1,3)  -- (3,-3);
```

The `name intersections` option generates the intersection coordinates, which are named `intersection`, followed by a dash and a number starting from 1. We need to specify the paths using the `of` keyword, as follows:

```
\fill[name intersections = {of = l1 and l2}]
  (intersection-1) circle(1mm) node[right] {here};
```

That command draws a bullet at the intersection of lines `l1` and `l2`, with text next to it. Together with our helper grid from *Figure 2.1*, it looks like this:

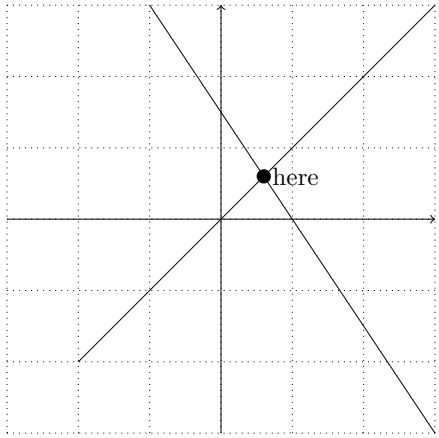

Figure 10.11 – A point at the intersection of two lines

This is the list of available `intersection` library keys and options:

- `name path`: This is the name we give to the ongoing path in the current scope. Use `name path global` if you need the path names beyond scopes.
- `name intersections`: That's a list of options in curly braces.

- `of`: Here, you specify the names of paths, together with the `and` keyword.
- `name`: You can select an optional prefix to replace the default intersection prefix.
- `total`: This is a macro name that stores the total number of intersections TikZ found, useful for iterating through in a `foreach` loop.
- `by`: Here, you can write a comma-separated list of coordinates that you want to use for the intersection points, such as `a`, `b`, and `c`, instead of `intersection-1`, `intersection-2`, and `intersection-3` respectively. Like in a `foreach` list, you can use the `...` notation.
- `sort by`: You can state the name of the path that shall be the reference for sorting the intersection coordinates, instead of the order in which TikZ found them.

We can create a more sophisticated example with complex paths and more intersection points to see that syntax come alive. In *Figure 7.9*, we had a circle intersecting triangle paths. Let's use this and choose two circles that overlap with the two triangles, as follows:

```
\fill[name path=triangle, orange]
  (90:2) -- (210:2) -- (330:2) -- cycle
  (90:1) -- (330:1) -- (210:1) -- cycle;
\draw[name path=circle, dashed, gray]
  circle(1.5) circle(0.65);
```

This gives us the following:

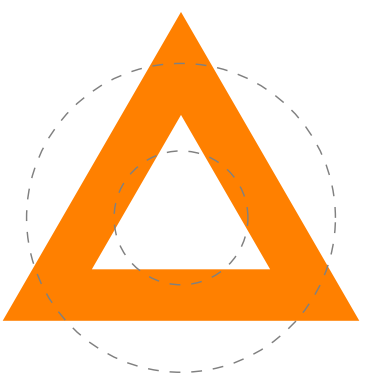

Figure 10.12 – A filled triangle path with intersecting circles

Now, we use the preceding keys and options to find, sort, and number all intersection points:

```
\fill[blue,
  name intersections = {of = triangle and circle,
  total=\max, name=c, sort by = circle}]
```

```
\foreach \i in {1,...,\max} {
  (c-\i) circle(0.5mm)
  node[above left=0.5mm,font=\tiny, inner sep=0]{\i}};
```

Now, TikZ finds `\max=12` intersection coordinates, sorted in the order of the circle's path, counterclockwise, and we can use the short `c` prefix for compact notation. That command adds the points and labels as follows:

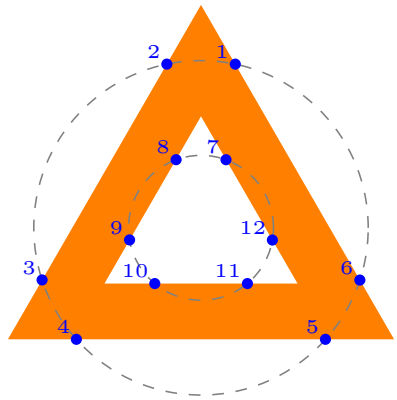

Figure 10.13 – Intersections of circles and triangles

You can see that even though our paths are not continuous but instead consist of multiple shapes, TikZ finds all intersection points with ease, regardless of the path complexity.

Summary

You saw that TikZ's loops are incredibly flexible. You can use loops whenever you see stuff repeating and want to save yourself from writing repetitive code.

Calculating with coordinates can now make your life easier. Just add a coordinate to move in the *x* or *y* direction or with an angle or distance. Use factors to place something in between nodes or coordinates. That's not just for geometry; this handy syntax is helpful for any lines, arrows, or positioning nodes in complex diagrams in a perfectly controlled manner.

Letting TikZ calculate intersection points of lines, curves, and complex paths helps you create more intricate shapes based on simpler ones.

In the next chapter, you will learn how to transform coordinates, paths, and scopes, such as by transposition and rotation.

Further reading

The following sections in the TikZ manual at `https://texdoc.org/pkg/tikz` are the reference for the commands, syntax, and libraries used in this chapter:

- *Part VII, Section 88, Repeating Things: The Foreach Statement*, gives all formal details of the `\foreach` command and its syntax. You can also find it at `https://tikz.dev/pgffor`.
- *Part III, Section 13.5, Coordinate Calculations*, is the reference for the `calc` library. The direct online link is `https://tikz.dev/tikz-coordinates#sec-13.5`.
- *Part III, Section 13.3, Coordinates at Intersections*, explains working with path intersections and is available online at `https://tikz.dev/tikz-coordinates#sec-13.3`.

As you already know, the TikZ galleries contain many examples relevant to this chapter. You can visit the following:

- `https://texample.net/tikz/examples/feature/foreach`
- `https://tikz.net/tag/foreach`
- `https://texample.net/tikz/examples/feature/coordinate-calculations`
- `https://tikz.net/tag/calc`

The Archimedean spiral is explained on Wikipedia at `https://wikipedia.org/wiki/Archimedean_spiral`.