# RGeom shape-defining DSL: what it is and how it works

## Introduction

RGeom is a geometrically-aware (2D) Ruby library for creating mathematical images. With minimal specification, you can render segments, triangles, circles, squares, and more. Each of those shape types is naturally defined by a class, e.g. RGeom::Shapes::Square, which extends RGeom::Shape.

To *create* a square object, though, you use the square method with some parameters. The following examples demonstrate. This is what we call *specifying* a square.

square :base => 5

square :base => :CD # C and D must exist

square :diagonal => :AX # A and X must exist

points :M => p(1,1), :T => p(5,-3)

square :MTGH # defines G and H

square :PQRS, :base => 9 # only P (possibly) defined

square # default (side = 5 units)

All of these methods (except point) return a Square object which has registered itself for later rendering. (If you don’t want it rendered – e.g. if you just need to derive some points as part of some other construction – use the \_square method instead.)

Therefore, there needs to be some code that understands the variety of ways in which you can specify a square and convert the specification into the internal representation: four vertices. That code must also guard against invalid specifications like :diagonal => :ABC or :base => “foo” or even :base => :CD where C and/or D don’t exist.

Initially, all this code was hand-written, but as shape after shape was added to RGeom’s artillery, the task of converting the user’s specification into objects was leading to a *lot* of boilerplate code with important repeated themes. What parameter combinations are allowable? What types must they be? What if something else is given; can it be cast? Default values. Error messages. Over time, the feeling grew that these repeated themes should be dealt with once and for all. Or, if that wasn’t possible, that at least significant support should be provided to clarify the code.

Enter the **shape** command. The definition of what arguments can and cannot be given to the square method is now handled like this:

shape :square, :label => :HIJK,

:parameters => %{

-

side: n

base: segment

diagonal: segment

}

From this we can see that the label, if given, must be four characters long (it’s specified by example), and that there are four possible parameter “combinations”: blank, side (a number), base (a segment) and diagonal (also a segment).

The magic words n and segment need to be defined too:

datatype :number, :alias => [:n,:angle],

:match => lambda { |o| o.numeric? and o }

So a number is anything that satisfies o.numeric? (a self-explanatory predicate I added to Object). The “and o” means that the object itself forms the return value of the block.

The datatype segment is more complicated.

datatype :segment,

:match => [

lambda { |o| o.is\_a? Segment and o },

lambda { |o| o.symbol? and Segment.from\_symbol(o) },

lambda { |o| o.is\_a? Array and o.map { |x| x.class } \

== [Point, Point] and Segment.simple(\*o)}

]

There are *three* ways an object can “match” the type definition of a segment:

* by being a segment (instance of class Segment);
* by being a symbol like :AB and being able to be converted into a segment;
* by being an array of two points

In each case, the lambda returns an object of type Segment. This is where we handle *casting*: at the point of use. That means that all three square specifications below are equivalent.

points :A => p(5,2), :B => p(-1,4.9)

p1, p2 = Point[:A], Point[:B]

seg = Segment.simple(p1, p2) # or Segment.from\_symbol(:AB)

square :base => :AB # (1)

square :base => seg # (2)

square :base => [p1,p2] # (3)

We’ve seen a simple shape declaration (:square) and the datatypes that support it. How does it it actually get turned into a Square object, though? What things are created along the way? What does the **shape** command actually *do*? To answer those questions is the purpose of this document, not the introduction, but for now it is worth mentioning that shape :square, … creates the Square class, and the top-level methods square and \_square. To interpret the arguments (:base => :AB etc.) it is necessary to implement the method Square.construct(spec) where spec is an object containing the arguments provided and some other information. The person writing that method can do so in the knowledge that correct arguments have been provided and the values have been cast in the desired manner. It simplifies the task of constructing the square enormously.

### Circle

The square definition was quite simple; it’s worth examining a more complicated shape.

shape :circle, :label => :K,

:parameters => %{

radius: segment

diameter: segment

centre: point=origin, radius: length=1

centre: point=origin, diameter: length

}

There are four allowable parameter combinations, two of which require two parameters. But here, for the first time, we are seeing ***default values***. Here is a list of valid circle specifications with any default values indicated in the comment.

circle # centre = origin, radius = 1

circle :radius => :AB

circle :diameter => :AB

circle :radius => 8 # centre = origin

circle :diameter => 5 # centre = origin

circle :centre => :A, :radius => 19

circle :centre => :B, :radius => 13

circle :centre => :C, :radius => :XY # :XY can resolve to a length

circle :centre => :A # radius = 1

The code required to construct a circle is so simple it’s worth quoting. (Note: it’s slightly edited to remove one complicated aspect not worth discussing here.)

def Circle.construct(spec)

centre, radius =

case spec.parameters

when [:centre, :radius] # point, length

[spec.centre, spec.radius]

when [:centre, :diameter] # point, length

[spec.centre, spec.diameter.to\_f/2]

when [:radius] # segment

segment = spec.radius

[segment.p, segment.length]

when [:diameter] # segment

segment = spec.diameter

[segment.midpoint, segment.length.to\_f/2]

end

Circle.new(spec.label, centre, radius)

end

Not having to check the validity of the parameters or cast to desired types or supply default values lifts a massive weight from the task of constructing a shape.

Before moving on to the most complicated shape yet, it’s worth examining one interesting corner of specifying a circle. We saw above that :XY can resolve to a length, so does

circle :radius => :XY

have the obvious meaning where :XY is interpreted as a segment, or does it mean a circle with the default centre of the origin and radius equal to the length :XY? In short, which of these parameter specs is matched?

radius: segment

centre: point=origin, radius: length=1

They both *would* match if given the chance, but it’s simple: first match wins. For this reason, it’s important to consider the order in which you specify acceptable parameter combinations.

Finally, it’s also worth looking at the two new data types introduces in circle: point and length.

datatype :point,

:match => [

lambda { |o| o.is\_a? Point and o },

lambda { |o| o.symbol? and Point[o] }

]

datatype :length,

:match => [

lambda { |o| o.numeric? and o },

lambda { |o| o.symbol? and o.length == 2 and

Segment.from\_symbol(o).length },

lambda { |o| o.is\_a? Segment and o.length }

]

If we decided that an array like [5,2] should be allowed to be interpreted as a point, it would be easy to add the code to make it happen.

And the default values: origin and 1. (Remember, the whole spec is a string, so 1 doesn’t arrive as an integer.)

value "origin", :value => Point[0,0]

value /[0-9.-]+/, :value => lambda { |x| Integer(x) rescue Float(x) }

### Triangle

We reach the pinnacle of shape complexity. Triangles have a richer “specification geometry” about them that squares or circles. There are equilateral, isosceles, scalene and right-angled triangles, each of which have different properties. RGeom aims to be geometrically rich, allowing code such as the following to just work. (It draws a triangle atop a square, like a child’s drawing of a house.)

square :ABCD, :side => 10

triangle :DCX, :isosceles, :height => 4

After the first line of code, the points A–D are all defined. Line 2 references D and C to specify the base of the isosceles triangle. (Shapes are constructed anti-clockwise, so order is important.)

Here is an incomplete but indicative list of the ways in which triangles can be specified. Labels may be provided or not (Line 2 above could just as well be triangle :isosceles, :base => :DC, :height => 4).

triangle :ABC, :equilateral, :base => 4

triangle :equilateral, :base => :XY

triangle :isosceles, :base => :XY, :height => 9

triangle :isosceles, :base => :XY, :angle => 50.d

triangle :isosceles, :base => :XY, :side => 13

triangle :sides => [5,12,13]

triangle :DEF, :base => 8, :angles => [25.d, 101.d]

triangle :ABC, :right\_angle => :B, :base => 6, :height => 3.5

triangle :ABC, :right\_angle => :C

triangle :GHI, :sas => [13, 35.d, 16] # not implemented yet

This is a lot for one specification to handle. “base” can be a segment or a length; “side” and “sides” appear at different times, likewise “angle” and “angles”. There is the concept of a “fixed argument”, which is :equilateral, :isosceles, :scalene, or not provided at all.

Here then, without further ado, is the specification for the shape *triangle*.

shape :triangle, :label => :ABC,

:fixed\_parameter => :type,

:declaration => "base: (segment,n=nil)",

:parameters => {

:isosceles => %{

(base)

(base), height: length

(base), angle: n

(base), side: length

},

:equilateral => %{

(base)

},

:scalene => %{

(base)

sides: [length,length,length]

(base), angles: [n,n]

(base), height: length

},

:\_ => %{

(base)

(base), right\_angle: symbol, height: length=nil, slant: symbol=nil

sides: [length,length,length]

(base), angles: [n,n]

(base), height: length

sas: [length,angle,length]

}

}

The special features we see here are, in order:

* :fixed\_parameter => type. This tells us that a fixed parameter is permissable for a triangle, and that it should be recorded in the specification as the “type” of the triangle. It’s like a shorthand for the user having to write triangle :type => :equilateral, ....
* :declaration => "base: (segment,n=nil)". Nearly every way of specifying a triangle optionally includes the keyword :base, and it can be a segment (:base => :AB) or a number (:base => 5). The overall spec would look ugly if this complex parameter were declared in full each time, so it is declared once and referred to throughout. If multiple declarations are needed, they can be put in an array.
* The :parameters keyword is not a *string* defining the acceptable parameter combos, as in the square and circle shapes. It is a *hash*, specifying a different set of parameter combos for each type of triangle. The special key :\_ means “no fixed parameter was specified”.

That concludes our introduction to RGeom’s important and powerful shape command. We’ve seen three examples of its use and touched on its side-effects. Now we go under the hood.

## Section title: How the shape command works

The shape command has this signature:

def shape(name, args={})

As an example:

shape :circle, :label => :K,

:parameters => %{

radius: segment

diameter: segment

centre: point=origin, radius: length=1

centre: point=origin, diameter: length

}

These are the objects produced by the Shape command:

* A ShapeCommandRawData object that teases the hash of arguments into the attributes label, declarations, fixed\_parameter, parameters. For an example involving a declaration and a fixed parameter, see the triangle spec in the introduction.
* An ArgumentParser object, fed the raw data, which, when required, will handle the user’s arguments (:R, :centre => :A, :radius => 5) and generate a ConstructionSpec object like:

spec.label == Label[:R]

spec.centre == Point[8,1]

spec.radius == 5

spec.parameters == [:centre, :radius]

It’s called a ConstructionSpec because this information is used to *construct* the Circle object in Circle.construct(spec). Notice how the centre :A has been turned into a Point object, and the parameters attribute tells us what parameters the user specified, which is useful for the construction logic.

* A ShapeProperties object containing the name (:circle), label size (1) and argument parser (see above). This object goes into an index where it can be retrieved by ShapeProperties[:circle], for instance. Anyone can access any shape’s vital properties any time, but that’s not the intention. We just need the Circle class to access them.
* Finally, the class Circle and the methods circle and \_circle are dynamically created. Here is the complete code for these things, as per our circle example:

class ::RGeom::Shapes::Circle < ::RGeom::Shape

CATEGORY = :circle

def self.shape\_properties

@\_properties ||= ShapeProperties[:circle]

end

end

module ::RGeom::Commands

def \_circle(\*args)

::RGeom::Shapes::Circle.create(\*args)

end

def circle(\*args)

\_circle(\*args).register

end

end

Because eval is used to create the class and methods, a mechanism was needed for the new class Circle to have knowledge of the ShapeProperties object created for it by the shape method. That mechanism is easily seen above.

To round out this example, it’s worth seeing how it works from the starting point of the circle method.

circle(:K, :radius => :AB)

→ Circle.create(:K, :radius => :AB)

→ prop = Circle.shape\_properties

spec = prop.generate\_construction\_spec([:K, :radius => :AB])

Circle.construct(spec)

It’s reasonably simple, but that’s because a lot of work is done by the method ShapeProperties#generate\_construction\_spec and its dependencies. We will examine that work in the next section.

The method Circle.create is actually defined in the Shape class and is shared by all subclasses present (Circle, Triangle, …) and future (Polygon, Bezier, Histogram, …).

Circle.shape\_properties is defined as a result of calling the shape method, and Circle.construct is circle-specific code written as part of the RGeom library.

To summarise:

* The shape method creates a class and two “command” methods based on the name of the shape it is defining.
* To support that class, a ShapeProperties object is created and stored in an index. That object:
  + can validate a label (:ABC is fine for triangles but :ABCD is not);
  + can generate a construction spec given the user’s arguments, handling all casting and default values logic, and raising appropriate errors if the arguments are invalid.
* The class uses that ShapeProperties object as part of its creation code (the Shape.create method).
* The classes ShapeCommandRawData and ArgumentParser are important implementation details but they exist only to serve ShapeProperties.

## Section title: A complete run through the ‘circle’ method