

JMathAnim Cookbook

Recipes for the Groovy scripting API and DSL



David Gutiérrez Rubio

Version: 0.7

03 September 2026

Copyright © 2026 David Gutiérrez Rubio.

The text, figures, and screenshots of this manual are licensed under the Creative Commons Attribution-ShareAlike 4.0 International License ([CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/)). The code examples are dedicated to the public domain under [CC0 1.0](https://creativecommons.org/licenses/publicdomain/1.0/) and may be used without restriction or attribution.

JMathAnim itself is distributed under the GNU General Public License v3.

Contents

Before you start	1
Chapter 1 Basic Usage	2
Create and execute a script	2
Define objects using DSL and autocomplete	2
Chapter 2 Creating Objects	6
Create a point and position it on screen	6
Create basic shapes	6
Create a shape using LOGO commands	7
Draw a function graph	8
Create a parametric curve	9
Shade the area under a graph	10
Draw a Riemann sum and refine it	12
Draw the tangent line at a point of a graph	13
Show the sine and cosine of an angle on the unit circle	14
Write a LaTeX formula	15
Create arrows	17
Create delimiters attached to objects	17
Create lines and segments	19
Create axes with ticks and legends	20
Add a label or tip to a shape	21
Create copies of an object	22
Create a group of objects	23
Create a group of objects with varying properties	24
Apply the same method to a list of objects	26
Chapter 3 Styling Objects	28
Change colors and thickness of an object	28
Save and reuse styles	29
Apply a random color	30
Apply a gradient to an object	31
Change the dash style of a shape	34
Highlight one term of a formula	35
Dim everything except one object	36
Chapter 4 Positioning Objects	38
Center an object on the screen	38
Put at a given position	38
Move or transform an object without animation	39
Position an object relative to another	41
Move an object so that a corner matches a given location	45
Stack an object to a side or corner of the screen	45
Align a group of objects	46
Creating a mesh of objects	47
Distributing objects in a circle	48
Distributing objects over a path	49
Layout of elements of unequal size	51
Align LaTeX objects to their baselines	52
Create several boxes with titles in a box layout	53
Positioning randomly	55
Position an object over other objects	56
Add a little watermark in a corner	57
Chapter 5 Labelling Objects	58

Label a point	58
Labelling shapes	58
Add a label to a vertex or side	60
Modifying a label	61
Modifying multiple labels of an object	62
Adding congruence or parallel marks to a polygon	63
Adding labels as independent objects	64
Adding length labels to delimiters	65
Point at a detail with a callout	66
Chapter 6 Setting Up Your Scenes	68
Setting up a proper workspace	68
Using resources from other directory	68
Use a secondary resources location	69
Reuse a common setup across several scenes	69
Reusing personalized palettes	70
Chapter 7 Animating Objects	73
Chapter 8 Output and Input	74
Generate images	74
Chapter 9 Compound Objects	75
Decide between a group and a compound	75
Build an object out of named pieces	76
Reach a piece of a compound after creating it	78
Place a piece against another piece	78
Create a chessboard	79
Animate the geometry of a compound with a single value	84
Chapter 10 Constructible Objects	87
Draw the incircle of a triangle	87

Before you start

Every recipe in this book is a Groovy script for JMathAnim. Scripts are normally written and run inside the integrated editor (`jmathanimgui`), but the same code can be executed from a plain Java `main` method:

```
public static void main(String[] args) {
    JMathAnimScene scene = new SceneClass();
    scene.execute();
}
```

Most objects can be built in two equivalent ways: the DSL way (a named-parameter builder with editor autocompletion) and the Groovy way (static factory methods plus chained setters).

The DSL is not a separate language: it is an extra layer built on top of the JMathAnim API, whose only purpose is to make development simpler. Named parameters, autocompletion and ready-made snippets mean you can describe an object by what it is rather than by the sequence of calls that build it, and that is the recommended starting point for most users.

The Groovy syntax sits one level below. It is the API itself, and it is aimed at advanced users: those writing loops and helper methods, those who need a method the DSL does not expose yet, or those interested in using JMathAnim as a plain Java library, where the DSL is not available at all.

Recipes show both side by side, with the DSL version on the left:

DSL

```
def c = shape(
    type: "circle",
    style: [drawColor: "blue",
           thickness: 2]
)
```

GROOVY

```
def c = Shape.circle()
        .drawColor("blue")
        .thickness(2)
```

Note

There are no “pure Groovy” or “pure DSL” ways. You mix them as you wish. Normally the DSL is well suited for creating objects with lots of arguments. For simple tasks, loops, or programming tasks, the Groovy syntax is more convenient.

Note

DSL syntax is only available from the Groovy interpreter. You cannot use it when using JMathAnim as a Java library.

Note

All DSL parameter keys are case-insensitive, and every keyword String value supports case-insensitive prefix matching: `"cen"` matches `CENTER`.

Basic Usage

Problem: Create and execute a script

You want to create your first animation.

Solution

The easiest way to create and execute a JMathAnim script is using the integrated editor. Once installed, open `jmathanimgui` in the menu and try the following simple code

DSL

```
def c = shape(type: "circle")
play.showCreation(c)
```

GROOVY

```
def c = Shape.circle()
play.showCreation(c)
```

and press F5. After saving the file, an animation drawing a circle will be shown in the preview window, and a log with the process will be printed in the log window, as in Figure 1.

JMathAnim uses themes to establish your color schemes. A theme is a predefined color scheme (background, shapes, etc.) and you can select one in the toolbar. Select a theme and press F5 again: the circle will be drawn on a different background color, and the circle itself will be a light color on a dark background, or a dark color on a light one.

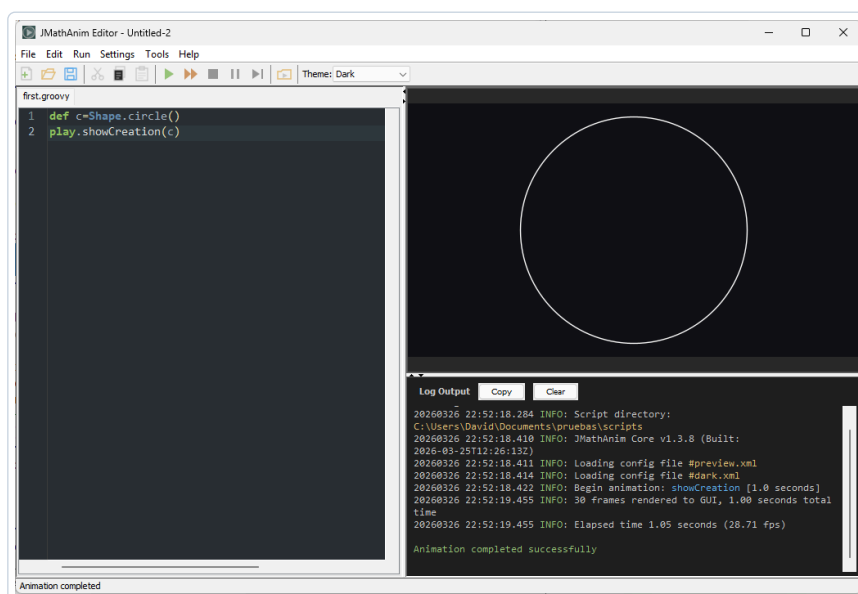


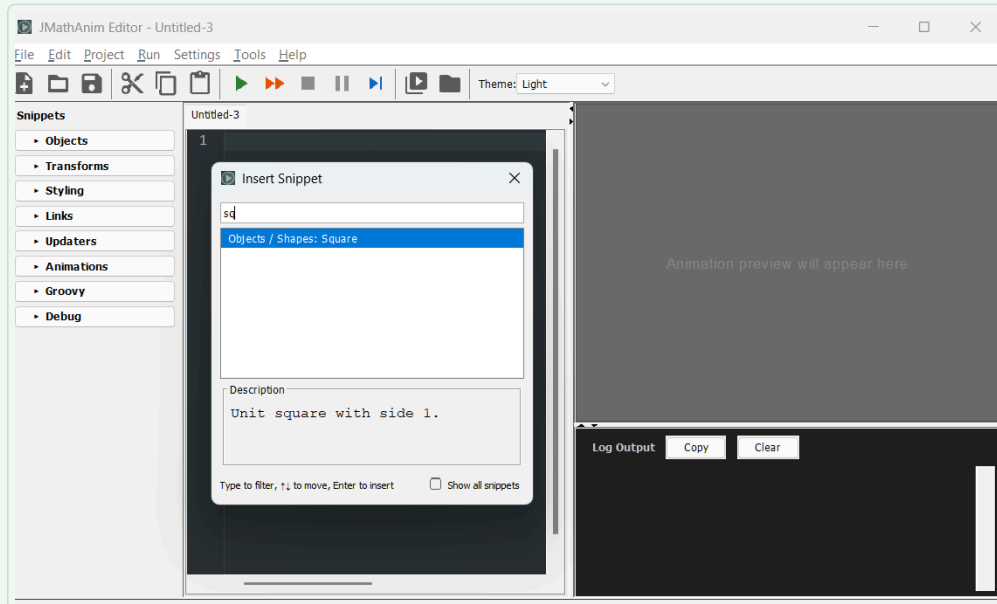
Figure 1: First simple animation

Problem: Define objects using DSL and autocomplete

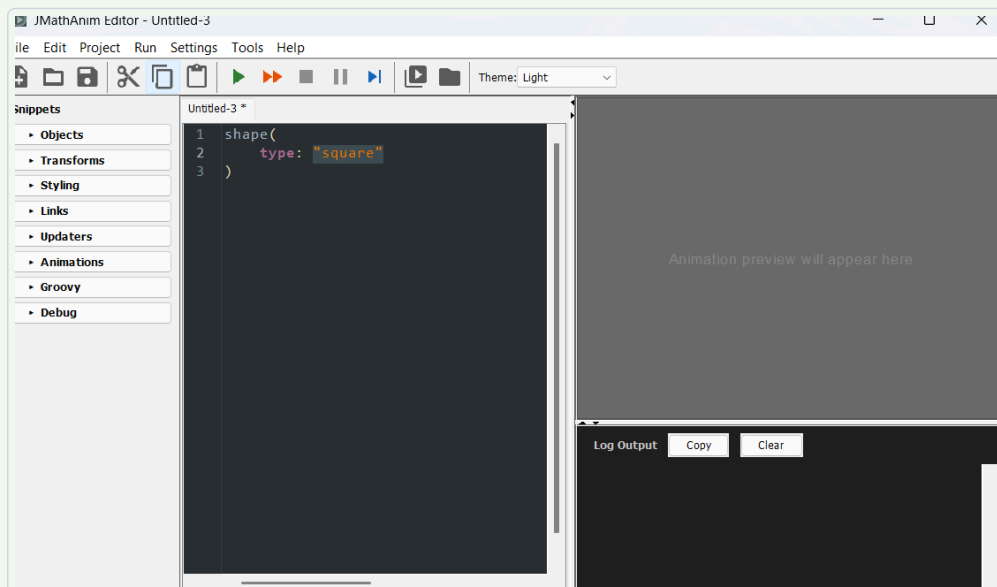
For example, you want to create a square with a fill color, scale it, and rotate it, but you don't remember the exact syntax or (like me) you are too lazy to write it.

Solution

The snippet sidebar is your friend! There you can find (almost) all DSL block definitions. If you are more of a “keyboard” user, try the shortcut **alt+S** to show a fast snippet search. Open the dialog and write “sq”:



The compatible candidates will appear. In this case, just one. Press **enter** and the DSL code is inserted (sometimes with example parameters):



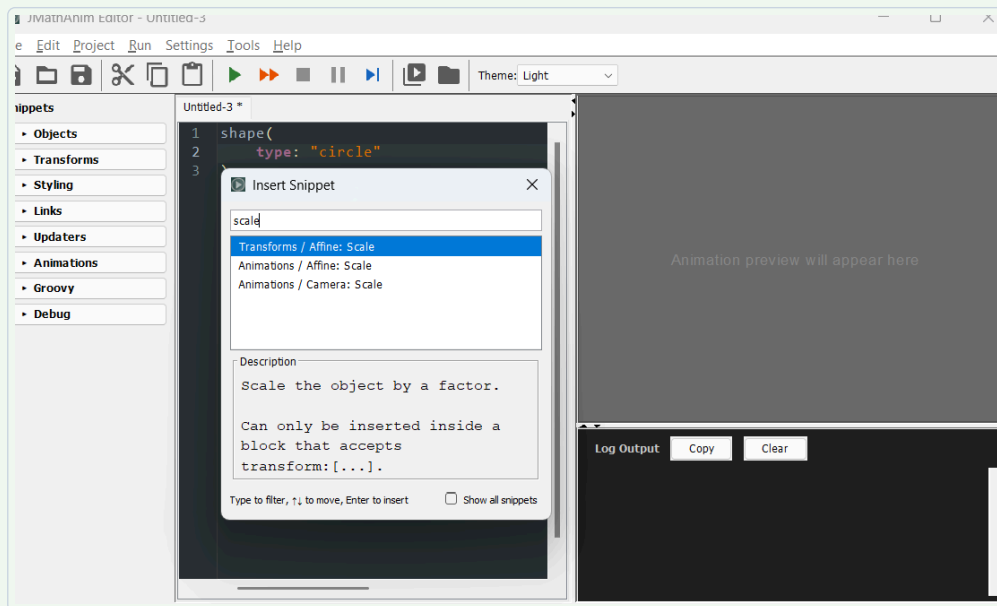
Note

Suppose you immediately regret your decision and want to create a circle instead. You can change it in two ways without deleting everything. You can put the cursor at the beginning of "square" and press **ctrl+space**. An autocomplete dialog will appear with all available options and a small help text. Another way is to invoke the snippet dialog selector again with **alt+S** and select "circle". The editor will replace "square" with "circle".

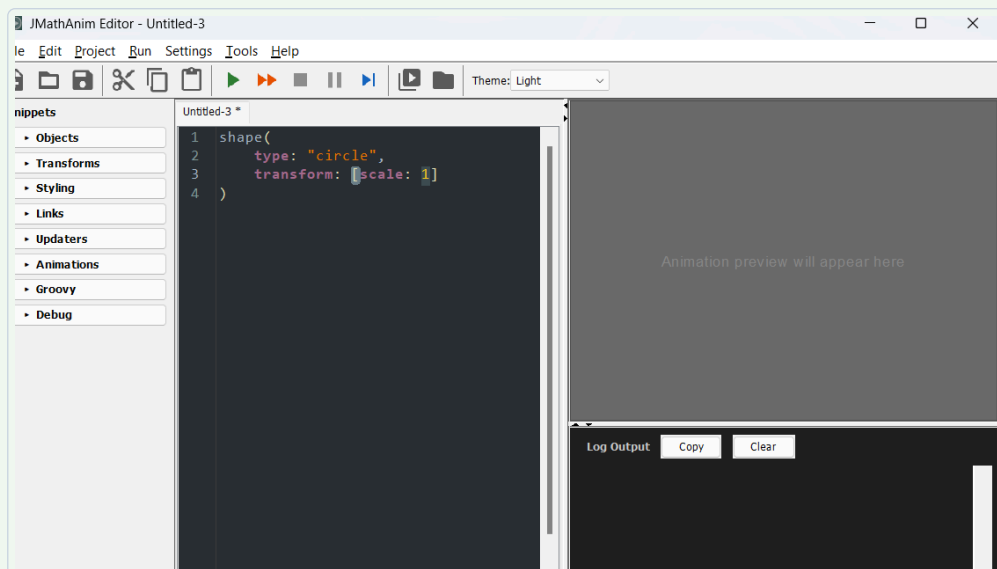
Note

Some parameters are optional and some are not. For example, if you specify "regularpolygon" in the `shape` block, a parameter called `sides` has to be set. Otherwise an error will be thrown asking for that missing parameter.

Ok, now let's transform it. Be sure that the cursor is inside the DSL block. As before, invoke the snippet dialog (or browse in the sidebar), and look for `scale`. This time you will see more available options:



We are interested in "Transforms/Affine: Scale". Press `enter` and the proper DSL parameters will be inserted in their place:



Let's change the value of `scale`. It can be a number (uniform scale) or a 2-list `[a, b]` for the x-scale and y-scale respectively. Put `[2, .75]`.

At this point you may want to experiment: apply a `rotate` transform of `PI/4` radians, for example, apply a `fillColor` and press `ctrl+shift+K` to invoke the color chooser dialog, change the `thickness` of the object, or just `center` it on the screen.

Finally, if you want this object to be added automatically to the screen, just add the parameter `addToScene: true` to the DSL block. This parameter is not in the snippet dialog, so you need to write it. Fortunately, the built-in autocomplete will suggest the word for you. You can also autocomplete the possible values (`true` / `false`).

Press F5 and save the file, and your beautiful square will be shown in the preview window.

Note

Remember that in a DSL block all parameters have the form `name: value` and are comma-separated. A common mistake is to forget the commas between parameters!

```
shape(  
  type: "circle", //A comma, ok!  
  transform: [scale: 1] //Error!, put a comma here!  
  addToScene: true //No comma here, but if you put one, no problem!  
)
```

Creating Objects

Problem: Create a point and position it on screen

You want to place a point at specific coordinates.

Solution

Create a `Point` object and add it to the scene.

DSL

```
def P = point(
  at: [0, 0],
  style: [drawColor: 'darkgreen',
    thickness: 80,
    dotStyle:
      "triangle_up_hollow"],
  addToScene: true
)

def Q = point(
  at: [1, 0],
  style: [drawColor: 'firebrick',
    thickness: 60,
    dotStyle: "ring"],
  addToScene: true
)
```

GROOVY

```
def P = Point.at(0, 0)
  .drawColor("darkgreen")
  .thickness(80)
  .dotStyle(DotStyle.TRIANGLE_UP_HOLLOW)

def Q = Point.at(1, 0)
  .thickness(60)
  .drawColor("firebrick")
  .dotStyle(DotStyle.RING)

//Add them to the scene
scene.add(P, Q)
```



Two points

Problem: Create basic shapes

You want to create circles, squares, regular polygons, and other basic shapes.

Solution

The `Shape` class has several static methods to build the most common shapes. Alternatively, you can use the DSL format. Autocomplete inside a DSL block shows you the available options.

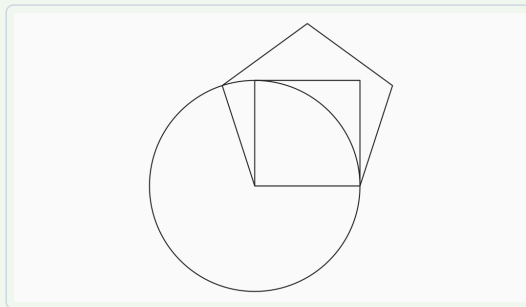
DSL

```
def circle = shape(  
  type: "circle")  
def square = shape(  
  type: "square")  
def regPol = shape(  
  type: "regularpolygon",  
  sides: 5)
```

GROOVY

```
//Unit circle centered at origin  
def circle = Shape.circle()  
//Unit square, corner at origin  
def square = Shape.square()  
//Pentagon with side (0,0)-(1,0)  
def regPol = Shape.regularPolygon(5)
```

```
//Add them to the scene  
scene.add(circle, square, regPol)  
//Ensure that everything is visible on screen  
camera.adjustToAllObjects()
```



Problem: Create a shape using LOGO commands

You want to define a shape using turtle graphics commands like **FD**, **RT**, **REPEAT**.

Solution

The method `Shape.logo(...)` allows the creation of a `Shape` with LOGO commands. The "turtle" starts at $(0,0)$ looking "up".

DSL

```
def logoShape = shape(  
  type: "logo",  
  commands: "FD .5 RT 90 FD 1")  
scene.add(logoShape)
```

GROOVY

```
def logoShape = Shape.logo(  
  "FD .5 RT 90 FD 1")  
scene.add(logoShape)
```



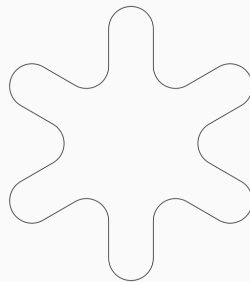
Angles are defined in degrees, where positive values correspond to counterclockwise orientation. JMathAnim includes 3 custom LOGO commands: **CLO** to close the path, and **LARC** / **RARC** `<angle>` `<radius>` that draw left or right arcs.

DSL

```
def logoShape = shape(  
  type: "logo",  
  commands: "REPEAT 6 [FD .5 " +  
    "RARC 180 .3 FD .5 " +  
    "LARC 120 .3] CLO")  
  
//Add to the scene  
scene.add(logoShape)  
  
//Ensure everything is on screen  
camera.adjustToAllObjects()
```

GROOVY

```
def logoShape = Shape.logo(  
  "REPEAT 6 [FD .5 " +  
    "RARC 180 .3 FD .5 " +  
    "LARC 120 .3] CLO")  
  
//Add to the scene  
scene.add(logoShape)  
  
//Ensure everything is on screen  
camera.adjustToAllObjects()
```



Problem: Draw a function graph

You want to plot the graph of a mathematical function.

Solution

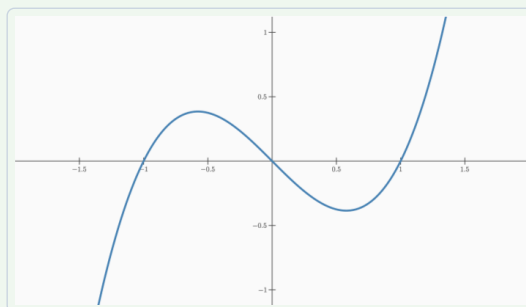
Use `funcGraph(...)`, passing the function as a Groovy closure.

DSL

```
axes(range: [-2, 2], xStep: .5,  
yStep: .5)  
def fg = funcGraph(  
  func: { x -> x * x * x - x },  
  xRange: [-1.5, 1.5],  
  style: [drawColor: "steelblue",  
    thickness: 15]  
)  
scene.add(fg)
```

GROOVY

```
scene.add(  
  Axes.make(-2, 2)  
    .generatePrimaryXTicks(-2, 2, .5)  
    .generatePrimaryYTicks(-2, 2, .5)  
)  
def fg = FunctionGraph.make(  
  { x -> x * x * x - x }, -1.5, 1.5)  
  
fg.drawColor("steelblue")  
  .thickness(15)  
scene.add(fg)
```



If you omit the range, the graph becomes **dynamic**: it spans whatever the camera is showing and redraws itself when you pan or zoom.

```
def fg = funcGraph(func: { x -> Math.sin(x) }) //Follows the camera
```

A `FunctionGraph` is a `Shape` underneath, so it accepts styling, labels, `play.showCreation`, and every other transformation. For curves that are not the graph of a function, see the `parametricCurve(...)` recipe.

Tip

With an explicit range, `numPoints`: controls how many sample points are used. Raise it for wildly oscillating functions, and lower it when the graph takes part in a long animation.

Problem: Create a parametric curve

You want to plot a curve given by parametric equations, such as a circle, a spiral or a polar rose.

Solution

`parametricCurve(...)` takes the coordinate functions as Groovy closures of the parameter, plus the parameter range `tRange`.

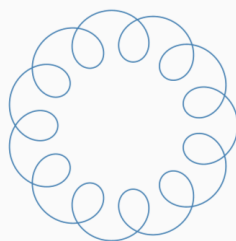
DSL

```
def c = parametricCurve(
  x: { .7*Math.cos(it)
      + .2*Math.cos(12*it) },
  y: { .7*Math.sin(it)
      + .2*Math.sin(12*it) },
  tRange: [0, 2 * PI],
  numPoints: 500,
  style: [drawColor: "steelblue"]
)
scene.add(c)
```

GROOVY

```
def c = ParametricCurve.make(
  { .7*Math.cos(it)
    + .2*Math.cos(12*it) },
  { .7*Math.sin(it)
    + .2*Math.sin(12*it) },
  { 0 }, //z(t): numPoints only
  0, 2 * PI, 500)

c.drawColor("steelblue")
scene.add(c)
```



Giving `r` and `theta` instead of `x` and `y` switches the curve to polar mode. The plain Groovy equivalent is `ParametricCurve.makePolar`.

DSL

```
//Three-petal rose
def rose = parametricCurve(
  r: { Math.cos(3*it) },
  theta: { it }, tRange: [0, PI])
```

GROOVY

```
//Three-petal rose
def rose = ParametricCurve.makePolar(
  { Math.cos(3*it) }, { it }, 0, PI)

//Archimedean spiral
```

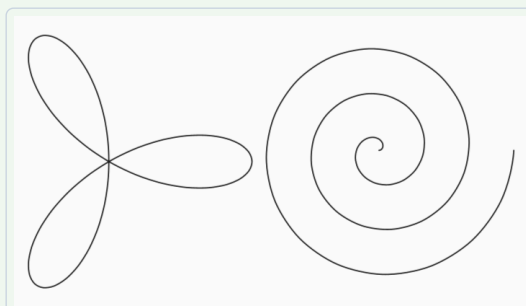
```
//Archimedean spiral
def spiral = parametricCurve(
  r: { .05*it }, theta: { it },
  tRange: [0, 6 * PI],
  stack: [to: rose,
    destinyAnchor: "right",
    gaps: .1])

scene.add(rose, spiral)
camera.adjustToAllObjects()
```

```
def spiral = ParametricCurve.makePolar(
  { .05*it }, { it }, 0, 6 * PI)

spiral.stack()
  .withGaps(.1, .1)
  .withDestinyAnchor(AnchorType.RIGHT)
  .toObject(rose)

scene.add(rose, spiral)
camera.adjustToAllObjects()
```



As with function graphs, `numPoints` sets how many sample points are used, and `tPoints` forces given parameter values to become nodes of the path, which is useful when the curve has a corner at a known value of the parameter.

Note

Instead of giving the coordinates one by one with `x:` and `y:` (or `r:` and `theta:`), a single `function:` closure can return them at once, as a `[x, y]` list, a `[x, y, z]` one, or any `Coordinates` object such as a `Vec`:

```
def c = parametricCurve(
  function: { t -> [Math.cos(t), Math.sin(2*t)] },
  tRange: [0, 2 * PI])

def d = parametricCurve(
  function: { t -> Vec.to(Math.cos(t), Math.sin(2*t)) },
  tRange: [0, 2 * PI])
```

This is the handy form when both coordinates share intermediate computations, since the closure body can name them once and use them twice.

Note

For the graph of an ordinary function $y = f(x)$, use `funcGraph(...)` instead. It is covered in the “Draw a function graph” recipe earlier in this chapter.

Problem: Shade the area under a graph

You want to fill the region between a function graph and the x axis, and then let the shaded interval grow while the animation runs.

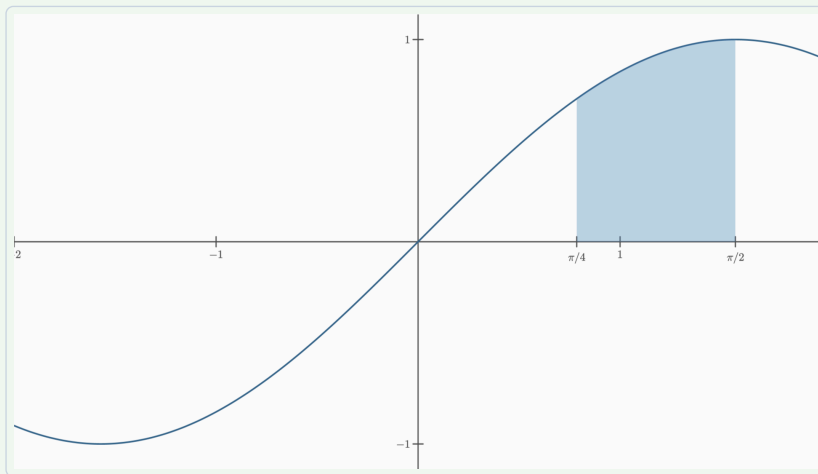
Solution

A `FunctionGraph` hands you that region as an ordinary `Shape`, through `getAreaShape(a, b)`. Being a plain shape, it takes the usual styling, the usual transforms and the usual animations.

```
def f = { double x -> 1 + Math.sin(x) }

def fg = funcGraph(func: f, xRange: [-3, 3],
  style: [drawColor: "steelblue", thickness: 8],
  addToScene: true)

def area = fg.getAreaShape(-1, 2)
apply(obj: area,
  style: [fillColor: "steelblue", fillAlpha: .35, drawAlpha: 0])
scene.add(area)
```



To make the right limit advance, keep a `scalar` for it and rebuild the path of the region on every frame. Only the path is copied, so the styling given above survives:

```
def b = scalar(-1) //the moving limit
def area = fg.getAreaShape(-1, -1) //empty region, styled once
apply(obj: area,
  style: [fillColor: "steelblue", fillAlpha: .35, drawAlpha: 0])
scene.add(area)

always(obj: area, dependsOn: b) { obj ->
  obj.path.copyStateFrom(fg.getAreaShape(-1, b.value).path)
}

animScalar(obj: b, from: -1, to: 2, runtime: 2)
```

Discussion

`scalar` + `always` + `animScalar` is the general pattern behind most “something moves and everything else follows” scenes: the scalar is the only thing the animation touches, the closure states how the drawing depends on it, and `dependsOn:` tells the update system to refresh the region **after** the scalar has changed, never one frame later.

Tip

Copying the whole object with `area.copyStateFrom(...)` would also copy the style of the freshly built region, undoing the fill. Copying just `obj.path` keeps geometry and appearance separate.

Problem: Draw a Riemann sum and refine it

You want the rectangles of a Riemann sum under a curve, and to double the number of rectangles a few times to suggest the limit.

Solution

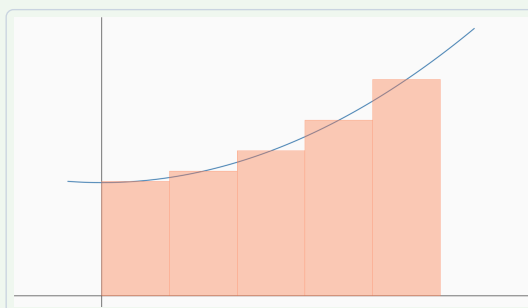
Wrap the construction in a closure taking the number of rectangles: what you get back is a `MathObjectGroup` that can be added, styled or animated as one object, and calling the closure again with another n gives the next stage.

```
axes()
def f = { double x -> 1 + x * x / 2 }

//Midpoint rule: n rectangles between a and b
def riemann(Closure f, double left, double right, int n) {
  def intervals = (linspace(left, right, n) as List).collate(2, 1, false)
  intervals.collect {
    def a = it[0]
    def b = it[1]
    shape(type: "rectangle",
          from: [a, 0], to: [b, f(.5 * (a + b))],
          style: [color: "coral", fillAlpha: .4,
                  thickness: 2])
  } as MathObjectGroup
}

def fg = funcGraph(func: f, xRange: [-.3, 3.3],
                  style: [drawColor: "steelblue", thickness: 8],
                  addToScene: true)

def bars = riemann(f, 0, 3, 6)
scene.add(bars)
camera.adjustToAllObjects()
```



The refinement is then a cross-fade between two consecutive stages. `appear` brings the new group into the scene and `disappear` takes the old one out, so no `scene.add` or `scene.remove` is needed:

```
[8, 16, 32].each { n ->
  def next = riemann(f, 0, 3, n)
  def out = disappear(obj: bars, type: "fadeout", runtime: .8, run: false)
  def into = appear(obj: next, type: "fadein", runtime: .8, run: false)
  animGroup(anim: [out, into])
  bars = next
}
```

Discussion

The two animations are built with `run: false` and handed to `animGroup` precisely so that they play **at the same time**; run one after the other and the curve is left bare for a second in between.

The method `linspace(left, right, n)` generates `n` numbers from `left` to `right`. For example `linspace(0, 3, 6)` generates `[0.0, 0.6, 1.2, 1.8, 2.4, 3.0]`, we convert it to the list with the `as List` converter and call to `collate(2, 1, false)` which means "build a list of pairs (2) with consecutive elements, each pair reuses (1) element from the previous list, don't generate the last one with only one element (false)". In practice we are generating the list of 6 intervals from 0 to 3: `[[0.0, 0.6], [0.6, 1.2], [1.2, 1.8], [1.8, 2.4], [2.4, 3.0]]`.

Tip

Change one line to get the other rules: `f(a)` for left endpoints, `f(b)` for right endpoints, and a `shape(type: "polygon", points: ...)` with the two heights for the trapezoidal rule.

Problem: Draw the tangent line at a point of a graph

You want a point sliding along a function graph carrying its tangent line with it, and a secant line that closes in on that tangent.

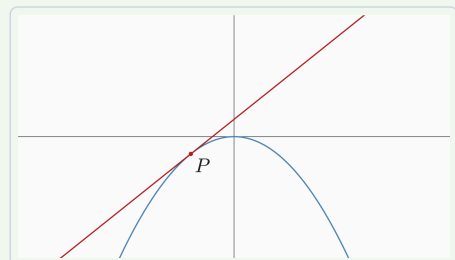
Solution

`pointOnGraph(...)` creates a point tied to the curve: it always sits at $(x, f(x))$, and it publishes the direction of the curve there through `getSlopePointRight()`. A constructible line through both is the tangent, and it keeps up on its own because a `ctLine` tracks the coordinates it was built from.

```
axes()
def f = { double x -> -x * x }

def fg = funcGraph(func: f, xRange: [-2.2, 2.2],
  style: [drawColor: "steelblue", thickness:
12],
  addToScene: true)
2
def P = pointOnGraph(x: -0.4, graph: fg,
  style: [color: "firebrick", dotStyle:
"circle", thickness: 40],
  label: r"$P$", addToScene: true)

def tangent = ctLine(a: P, b:
P.getSlopePointRight(),
  style: [drawColor: "firebrick", thickness:
```



```
12],
  addToScene: true)
```

Sliding P is a matter of writing its abscissa and letting the point put itself back on the curve. Note the `when: "before"`: the closure has to run **before** the point recomputes its own ordinate, or the dot would lag one frame behind the curve.

```
def x = scalar(-1.4)
always(obj: P, dependsOn: x, when: "before") { obj -> obj.moveTo(x.value, 0) }

animScalar(obj: x, from: -1.4, to: 1.8, runtime: 4, lambda: "smooth")
```

The secant is the same idea with a second point at distance h , and the classical picture is just `animScalar` driving h to zero:

```
def h = scalar(1.6)
def Q = pointOnGraph(x: P.v.x + 1.6, graph: fg,
  style: [color: "seagreen", dotStyle: "circle", thickness: 40],
  label: r"$Q$", addToScene: true)

always(obj: Q, dependsOn: h, when: "before") { obj -> obj.moveTo(P.v.x + h.value, 0) }

def secant = ctLine(a: P, b: Q,
  style: [drawColor: "seagreen", thickness: 4],
  addToScene: true)

animScalar(obj: h, from: 1.6, to: .01, runtime: 3, lambda: "smooth")
```

Tip

The `y` passed to `moveTo` is ignored: a `pointOnGraph` overwrites it with $f(x)$ on every update. Writing `0` there is the usual way of saying "only the abscissa matters".

Note

Do not drive h all the way down to `0`: the two points would coincide and the line through them would be undefined. A small value is visually indistinguishable and always well defined.

Problem: Show the sine and cosine of an angle on the unit circle

You want the classical unit-circle picture: a point turning around the circle and the two segments that project it on the axes.

Solution

One scalar holds the angle, one closure puts the point where it belongs, and the rest of the drawing is built out of constructible segments, which follow their endpoints without any further code.

```
axes(range: [-1.5, 1.5], xStep: .5, yStep: .5)

def t = scalar(0) //the angle

shape(type: "circle", style: [drawColor: "gray", thickness: 3],
```

```

addToScene: true)

def O = point(at: [0, 0], style: [drawAlpha: 0], addToScene: true)
def P = point(style: [color: "firebrick", dotStyle: "circle", thickness: 8],
  addToScene: true)
def H = point(style: [drawAlpha: 0], addToScene: true) //foot on the x axis

always(obj: P, dependsOn: t) { obj -> obj.moveTo(Math.cos(t.value),
  Math.sin(t.value)) }
always(obj: H, dependsOn: P) { obj -> obj.moveTo(P.v.x, 0) }

ctSegment(a: O, b: P, style: [drawColor: "gray", dashStyle: "dashed"],
  addToScene: true)
ctSegment(a: O, b: H, style: [drawColor: "steelblue", thickness: 8],
  label: r"$\cos t$", addToScene: true)
ctSegment(a: H, b: P, style: [drawColor: "coral", thickness: 8],
  label: r"$\sin t$", addToScene: true)

animScalar(obj: t, from: 0, to: 2 * PI, runtime: 5)

```

Note

H is only there to be the corner of the two segments, so it is drawn with `drawAlpha: 0`. It still has to reach the scene: an object that is not in it is not part of the dependency graph, and its updater would never run. `scene.addDependency(H)` registers it without drawing it, if you prefer that to an invisible object.

Tip

Attaching the labels to the segments rather than to the scene means they travel with them: a `label` given to a shape becomes an internal object of that shape and is drawn, moved and animated with it.

Problem: Write a LaTeX formula

You want to render a mathematical expression using LaTeX.

Solution

Use the `latex(...)` command, or the `LatexMathObject.make` method, to create LaTeX objects.

DSL

```

def t = latex(
  text: r'$e^{i\pi}+1=0$')
scene.add(t)

```

GROOVY

```

def t = LatexMathObject
  .make(r'$e^{i\pi}+1=0$')
scene.add(t)

```

$$e^{i\pi} + 1 = 0$$

LaTeX makes extensive use of the backslash character to invoke its commands, as well as `$` to enter/exit mathematical mode. Both characters need to be escaped properly in Groovy. You can add escape characters or use the `r` modifier before a string:

```
// INCORRECT - Groovy will interpret $ and \ as escape sequences
latex(text: "$e^{i\pi}+1=0$") // ERROR!

// CORRECT - Escape the special characters
latex(text: "\\$e^{i\\pi}+1=0\\$") // Works!

// BETTER - Use raw strings (prefix with 'r')
latex(text: r"$e^{i\pi}+1=0$") // Easiest!
latex(text: r'$e^{i\pi}+1=0$') // Single or double quotes

//And triple quotes to define multiline strings
latex(text: r'''
$
e^{i\pi}+1=0
$
''')
```

Tip

In the integrated editor, the LaTeX editor in the **Tools** menu writes LaTeX expressions and generates the code needed in JMathAnim (Figure 2). Apart from a fast preview, individual glyphs can be selected to work with parts of a math expression.

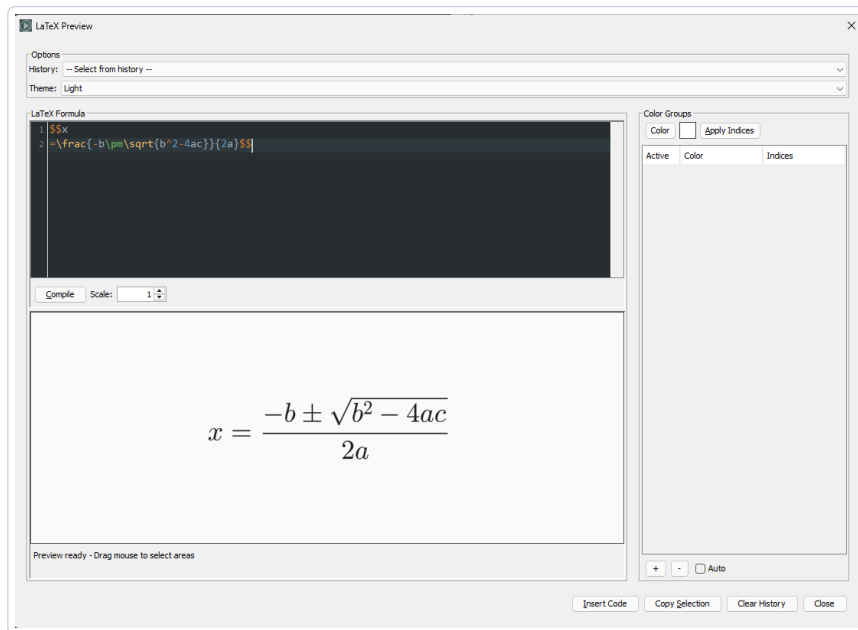


Figure 2: LaTeX editor

Problem: Create arrows

You want to create an arrow between 2 points.

Solution

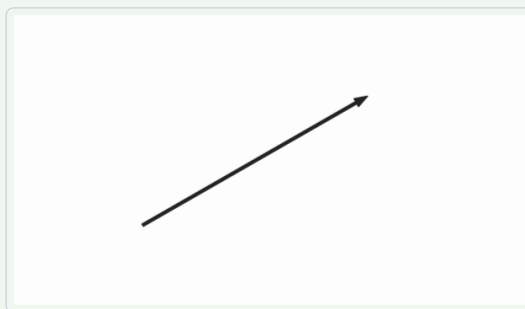
Use the `arrow(...)` command, or the `Arrow` class.

DSL

```
def ar = arrow(
  from: [-1, -.5],
  to: [.75, .5],
  head: "arrow1"
)
scene.add(ar)
```

GROOVY

```
def a = Vec.to(-1, -.5)
def b = Vec.to(.75, .5)
def ar = Arrow.make(a, b,
  ArrowType.ARROW1)
scene.add(ar)
```



Apart from the usual styling commands, you can set the arrow thickness and the curvature.

Problem: Create delimiters attached to objects

You want to add braces, brackets, or other delimiters next to a shape.

Solution

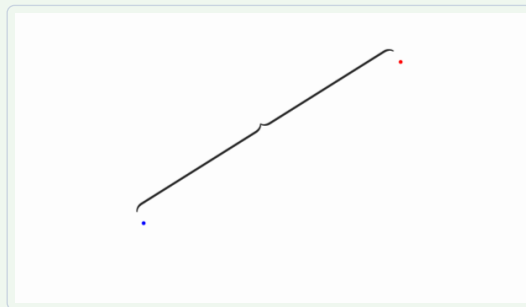
Use the `delimiter(...)` command, or the `Delimiter` class.

DSL

```
def a = point(at: [-1, -.5],
  style: [drawColor: 'blue'])
def b = point(at: [1, .75],
  style: [drawColor: 'red'])
def del = delimiter(
  from: a,           //start
  to: b,             //end
  type: "brace",     //Type
  gap: .1           //Gap point-
  delimiter
)
scene.add(a, b, del)
```

GROOVY

```
def a = Point.at(-1,
  -.5).drawColor("blue")
def b =
  Point.at(1, .75).drawColor("red")
def del = Delimiter.make(
  a,           //start
  b,           //end
  DelimiterType.BRACE, //Type
  .1           //Gap point-
  delimiter
)
scene.add(a, b, del)
```



The `Delimiter` class allows both classic delimiters like `BRACE`, `BRACKET`, or `PARENTHESIS`, and measuring delimiters like `LENGTH_BRACKET` and `LENGTH_ARROW`. A delimiter can also be stacked to an object:

DSL

```
def sq = shape(
  type: "centeredsquare",
  transform: [scale: [1.5, 1]],
  style: [drawColor: 'darkgreen']
)
def del1 = delimiter(
  stackedTo: sq, //Stack to this
  object
  anchor: "lower", //stacked lower
  type: "length_arrow",
  gap: .1         //Gap object-
  delimiter
)
def del2 = delimiter(
  stackedTo: sq,
  anchor: "right",
  type: "length_arrow",
  gap: .1
)
scene.add(sq, del1, del2)
```

GROOVY

```
def sq = Shape.square()
  .scale(1.5, 1)
  .center()
  .drawColor("darkgreen")

def del1 = Delimiter.makeStacked(
  sq,           //Stack
  to this object
  AnchorType.LOWER, //
  stacked lower
  DelimiterType.LENGTH_ARROW, //Type
  .1           //Gap
  object-delimiter
)
def del2 = Delimiter.makeStacked(
  sq, AnchorType.RIGHT,
  DelimiterType.LENGTH_ARROW, .1)

scene.add(sq, del1, del2)
```



Problem: Create lines and segments

You want to draw infinite lines, rays, or line segments.

Solution

There are three different objects. A `Shape.segment` is an ordinary shape bounded by its two endpoints, while `Line` and `Ray` redraw themselves on every frame so that they always span the whole visible view, no matter how you zoom or pan.

DSL

```
def seg = shape(
  type: "segment",
  from: [0, 0], to: [1, .5],
  style: [drawColor: 'orangered3'])

def lin = line(
  a: [0, 0], b: [1, 1],
  style: [
    drawColor: 'darkblue',
    thickness: 20])

//A direction can replace the
//second point instead of b
def ray = ray(
  a: [0, 0], dir: [1, -1],
  style: [
    drawColor: 'darkolivegreen3',
    thickness: 20,
    dashstyle: "dashdotted"])

scene.add(seg, lin, ray)
```

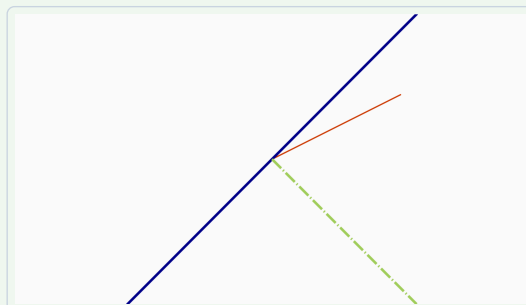
GROOVY

```
def seg = Shape.segment(
  Vec.to(0, 0), Vec.to(1, .5))
  .drawColor("orangered3")

def lin = Line.make(
  Vec.to(0, 0), Vec.to(1, 1))
  .drawColor("darkblue")
  .thickness(20)

//A direction can replace the
//second point
def ray = Ray.makePointDir(
  Vec.to(1, 0), Vec.to(1, -1))
  .drawColor("darkolivegreen3")
  .thickness(20)
  .dashStyle(DashStyle.DASHDOTTED)

scene.add(seg, lin, ray)
```



`Line` and `Ray` also offer ready-made axes and bisectors, useful as construction guides:

```
def x = Line.XAxis()
def y = Line.YAxis()
def d = Line.XYBisector() //The line y=x
def r = Ray.XAxisPositive() //From the origin, to the right
```

Problem: Create axes with ticks and legends

You want to display a Cartesian axis system with labeled ticks.

Solution

The `axes(...)` command builds a pair of axes with their ticks and numeric legends. Unlike most objects, it is added to the scene right away, so pass `addToScene: false` if you want to hold it back. In plain Groovy the ticks are generated in bulk:

DSL

```
axes(
    //Range covered by primary ticks
    xRange: [-3, 3], yRange: [-2, 2],
    //Distance between primary ticks
    xStep: 1, yStep: .5,
    //Smaller, unlabeled ticks
    secondaryX: [step: .25],
    style: [drawColor: "darkblue"]
)
```

GROOVY

```
def ax = Axes.make()
ax.generatePrimaryXTicks(-3, 3, 1)
ax.generatePrimaryYTicks(-2, 2, .5)
//Secondary ticks: last parameter is
//the max view width to keep them
//visible
ax.generateSecondaryXTicks(-3, 3, .25,
4)
ax.drawColor("darkblue")

scene.add(ax)
```



Individual ticks can carry their own LaTeX legend, which is the usual way of marking notable values. In plain Groovy the legends are added one by one:

DSL

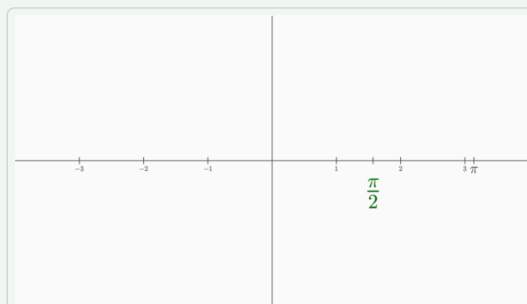
```
def ax = axes(
    xRange: [-PI, PI], xStep: PI/2,
    ticksX: [
        [at: PI/2, label: r"$\frac{\pi}{2}$"],
        [at: PI, label: r"$\pi$"]
    ]
)
```

GROOVY

```
def ax = Axes.make()
ax.generatePrimaryXTicks(-PI, PI, PI/2)

//A tick with a custom LaTeX legend
ax.addXTicksLegend(r"$\frac{\pi}{2}$",
    PI/2, TickType.PRIMARY)
ax.addXTicksLegend(r"$\pi$",
    PI, TickType.PRIMARY)

scene.add(ax)
```



Tip

The `axes` object, when defined in a DSL block, is automatically added to the scene. You can prevent this with the parameter `addToScene: false`.

Problem: Add a label or tip to a shape

You want to attach a LaTeX label or a marker to a specific point of a shape's contour.

Solution

The quickest way is the inline `label:` parameter, available in every DSL object. The label becomes an **internal object** of the shape: it moves, draws, and animates together with it.

DSL

```
//A label at the middle of the path
(t=0.5 by default)
def sq = shape(type: "square", label:
r"$x$")

//Several labels, anchored at the
vertices of a triangle
def tri = shape(type: "regularpolygon",
sides: 3,
label: [
[text: r"$A$", vertex: 0],
[text: r"$B$", vertex: 1],
[text: r"$C$", vertex: 2]],
stack: [to: sq,
destinyanchor: "right",
gaps: .4]
)
scene.add(sq, tri)
camera.adjustToAllObjects()
```

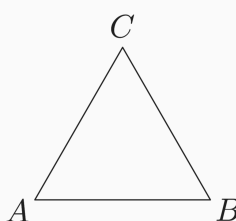
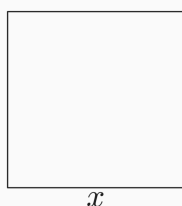
GROOVY

```
//A label at the middle of the path
(t=0.5 by default)
def sq = Shape.square()
.addLabel(r'$x$')

//Several labels, anchored at the
vertices of a triangle
def tri = Shape.regularPolygon(3)
.stack()
.withGaps(.4, .4)
.withDestinyAnchor(AnchorType.RIGHT)
.toObject(sq)

tri.addLabel(r"$A$").setVertexIndex(0)
tri.addLabel(r"$B$").setVertexIndex(1)
tri.addLabel(r"$C$").setVertexIndex(2)

scene.add(sq, tri)
camera.adjustToAllObjects()
```



When the label needs a life of its own (appear later, be highlighted or removed on its own), build it with the standalone `label(...)` command. The parameter `t` is the position along the path, from 0 to 1:

```
def hexagon = shape(type: "regularpolygon", sides: 6)

def sideLabel = label(path: hexagon, t: 1.5/6, text: r"$a+b$",
    style: [color: "darkslateblue"])
def arrowTip = label(path: hexagon, t: 0.5/6, type: "arrow2")
def equalMark = label(path: hexagon, t: 3.5/6, type: "equal2")

scene.add(hexagon, sideLabel, arrowTip, equalMark)
```

Pass exactly one of `text` (a LaTeX label), `type` (an arrowhead `arrow1`.. `arrow3`, `square`, `bullet`, or equal marks `equal1`.. `equal4`), or `object` (any MathObject used as the tip). The equivalent in plain Groovy is the `LabelTip` class, whose last parameter chooses the side of the curve:

DSL

```
def tip = label(path: hexagon,
    t: 1.5/6, text: r"$a+b$")
scene.add(tip)
```

GROOVY

```
def tip = LabelTip.make(
    hexagon, 1.5/6, r"$a+b$", false)
scene.add(tip)
```

Problem: Create copies of an object

For example, you have a Shape and want to create several copies without having to repeat all the commands you used to create the first one.

Solution

There are several ways. The most direct is to use the `.copy()` method, which returns a copy of the object.

DSL

```
def sq1 = shape(
    type: "centeredsquare",
    transform: [rotate: PI/4,
        scale: .25],
    style: [
        fillColor: 'yellowgreen',
        drawColor: 'brown',
        thickness: 20]
)
def sq2 = sq1.copy()
apply(
    obj: sq2,
    transform: [shift: [1,0]],
    style: [drawColor: 'blue'],
)

scene.add(sq1, sq2)
```

GROOVY

```
def sq1 = Shape.square()
    .center()
    .rotate(PI/4)
    .scale(.25)
    .fillColor("yellowgreen")
    .drawColor("brown")
    .thickness(20)

def sq2 = sq1.copy()
    .shift(1,0)
    .drawColor("blue")

scene.add(sq1, sq2)
```



Tip

The Groovy interpreter overloads the `+` operator to create copies easily. The expression `object + vector` will return a copy of the object shifted by the given vector. With the `sq1` object created previously:

```
def sq1 = ...
def sq2 = sq1 + Vec.to(.5, 0)
//Shorter syntax with 2-lists
def sq3 = sq1 + [0, .5]
//Sums and differences
def sq4 = sq1 - [0, .5]
//You can also scale the object!
def sq5 = 1.75*sq1 + [-0.5, 0]
scene.add(sq1, sq2, sq3, sq4, sq5)
```



Problem: Create a group of objects

You want to handle several objects as a single one.

Solution

A `MathObjectGroup` is a logical group; it behaves as one object: you style it, transform it, animate it, and add it to the scene in a single call, while its members remain individually reachable. However, it is not a `MathObject` itself. Adding a `MathObjectGroup` to the scene is equivalent to adding all its members. As a result, an object can be a member of two different groups.

DSL

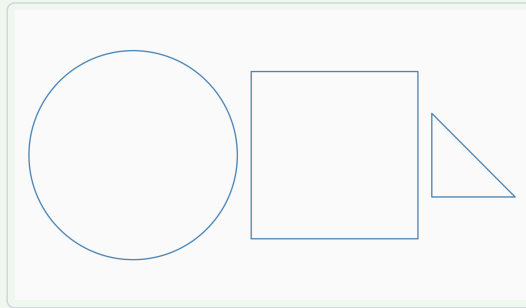
```
def a = shape(type: "circle",
  transform: [scale: .75])
def b = shape(type: "square",
  transform: [scale: 1.2])
def c = shape(type: "triangle",
  transform: [scale: .6])
def g = group(
  obj: [a, b, c],
  layout: "right",
  gap: .1,
  style: [drawColor: "steelblue"]
)
scene.add(g)
camera.adjustToAllObjects()
```

GROOVY

```
def a = Shape.circle().scale(.75)
def b = Shape.square().scale(1.2)
def c = Shape.triangle().scale(.6)

def g = MathObjectGroup.make(a, b, c)
  .setLayout(new SimpleLayout(
    LayoutType.RIGHT, .1, .1))
  .drawColor("steelblue")

scene.add(g)
camera.adjustToAllObjects()
```



The `layout` parameter arranges the members relative to each other. Simple directions are named by a `LayoutType` value (`right`, `left`, `upper`, `lower`, `diag1`..`diag4`, plus aligned variants such as `rupper` or `llower`), and more elaborate arrangements come from a layout spec map:

DSL

```
def g = group(obj: [a, b, c, d, e, f],
  layout: [type: "circular",
    center: [0, 0], radius: 2])

def g2 = group(obj: shapes,
  layout: [type: "box", size: 3,
    gaps: [.1, .1], corner: [0, 0]])
```

GROOVY

```
def g = MathObjectGroup.make(a, b, c, d,
  e, f)
g.setLayout(CircularLayout.make(
  Vec.to(0, 0), 2))

def g2 = MathObjectGroup.make(shapes)
g2.setLayout(BoxLayout.make(
  Vec.to(0, 0), 3, .1, .1))
```

A group can also be built out of copies of a single object, which saves writing a loop:

DSL

```
def gr = group(
  copies: shape(type: "square",
    transform: [scale: .2]),
  number: 5,
  layout: "right", gap: .05)
```

GROOVY

```
def gr = MathObjectGroup.makeCopies(
  Shape.square().scale(.2), 5)
gr.setLayout(new SimpleLayout(
  LayoutType.RIGHT, .05, .05))
```

Members are reachable by index with `gr.get(0)` or the Groovy overloading `gr[0]`, and `gr.size()` gives the number of elements. Styling or transforming the group reaches every member, so `gr.scale(.5)` scales the whole group as a block.

Tip

Groups are the natural input of `animSetLayout(...)`, which animates the transition from the current arrangement to a new one.

Problem: Create a group of objects with varying properties

You want to create a group of shapes with different colors, sizes, or other properties.

Solution

Use the Groovy `collect` method. This method applies a closure to all elements of a list. This list can be converted to a `MathObjectGroup`.

DSL

```
def cs=colorScale(
  name: "viridis",
```

GROOVY

```
def cs = ColorScale.viridis(3, 8)
```

```

    from: 3,
    to: 8
  )
  def mg = (3..8).collect {
    shape(
      type: "regularpolygon",
      sides: it,
      style: [
        fillColor: cs(it),
        alpha: .3],
      )
  } as MathObjectGroup
  scene.add(mg)
  camera.adjustToAllObjects()

```

```

def mg = MathObjectGroup.make()
for (int i = 3; i <= 8; i++) {
  mg.add(
    Shape.regularPolygon(i)
      .fillColor(cs(i))
      .alpha(.3)
  )
}
scene.add(mg)
camera.adjustToAllObjects()

```



Discussion

The `collect` method follows the philosophy “take this list of elements and work on them”. Groovy and the JMathAnim extensions provide many ways to define such lists. For example, `3..8` represents a list with numbers 3 to 8 inclusive. The method `arange(0, .4, .1)` returns the list `[0, 0.1, 0.2, 0.3]`, that is, “go from 0 to 0.4 (not included) in steps of .1”. The method `linspace(0, 1, 5)` is something like “go from 0 to 1 (including both) in exactly 5 steps”, that is, `[0.0, 0.25, 0.5, 0.75, 1.0]`. Depending on the case you will want to use one or the other.

Tip

In Groovy you can define a closure (say, a function or a method to apply to a list, for example) by explicitly defining its parameter:

```
def f = { x -> x*x }
```

or you can omit it and use the default `it`¹ variable for a shorter syntax:

```
def f = { it*it }
```

Tip

In the DSL example it is not necessary to convert the `list` to `MathObjectGroup` as `scene.add()` admits lists of objects, but `MathObjectGroup` is a fancy way to handle groups of

¹Not to be confused with the ancient cosmic entity that often takes the form of a clown and feeds on fear.

objects. It admits basic batch operations (color, shift, etc.) and the `layout` method to position the objects in a wide variety of ways.

Problem: Apply the same method to a list of objects

For example, you have 3 objects and want to put a text on them, avoiding code repetition.

Solution

Use the `each` method with a dictionary. Dictionaries have the syntax `[key:value,...]` and allow you to create a basic "map from this to this".

```
def good = shape(type: "polygon",
  points: [[-0.55, -0.60], [0.55, -0.60], [0.0, 0.65]],
  style: [color: "darkgoldenrod"],
  addToScene: true
)
def bad = shape(type: "polygon",
  points: [[-0.42, 0.65], [0.42, 0.65], [0.24, -0.60], [-0.24, -0.60]],
  style: [color: "gray11"],
  stack: [to: good, destinyAnchor: "right_and_aligned_lower"],
  addToScene: true
)
def ugly = shape(type: "polygon",
  points: [[-0.60, -0.60], [0.55, -0.52], [0.38, 0.45], [-0.32, 0.30]],
  style: [color: "saddlebrown"],
  stack: [to: bad, destinyAnchor: "right_and_aligned_lower"],
  addToScene: true
)

//With this dictionary we can define a map between strings and objects
def names = ["The Good": good, "The Bad": bad, "The Ugly": ugly]

//Iterate over every value of the dictionary
names.each { name, figure ->
  latex(
    text: name,
    stack: [to: figure, destinyAnchor: "lower", gaps: .2],
    addToScene: true
  )
}
camera.adjustToAllObjects()
```



The Good The Bad The Ugly

Tip

The Groovy `each` method is used when you want to execute certain code without worrying about the returned value. If you want to store the created texts in a variable, use `collect`. This method returns a `list` but you can convert it into a `MathObjectGroup` easily by putting an `as MathObjectGroup` at the end.

```
def captions = names.collect { name, figure ->
    latex(
        text: name,
        stack: [to: figure, destinyAnchor: "lower", gaps: .2]
    )
} as MathObjectGroup
```

Styling Objects

Problem: Change colors and thickness of an object

You want to customize the appearance of an object (draw color, fill color, thickness, alpha).

Solution

Use the methods `.drawColor`, `.fillColor`, `.drawAlpha`, `.fillAlpha`, or `.thickness`. The same properties are available in the `style` map of any DSL definition. The autocomplete feature of the editor lists the available options.

DSL

```
def c = shape(
  type: "regularinscribedpolygon",
  sides: 5,
  style: [
    drawColor: 'green',
    fillColor: 'orange',
    fillAlpha: .5,
    thickness: 40
  ]
)
```

GROOVY

```
def c = Shape.regularInscribedPolygon(5)
  .drawColor("green")
  .fillColor("orange")
  .fillAlpha(.5)
  .thickness(40)
scene.add(c)
```

Tip

There is a short form, `alpha`, that reaches the draw and the fill transparency in one call:

DSL

```
style: [alpha: .5]
```

GROOVY

```
obj.alpha(.5)
```

It sets `drawAlpha` and *multiplies* `fillAlpha`, so a shape whose interior is already half transparent gets dimmer still rather than jumping to the value written. Reach for `drawAlpha` and `fillAlpha` when the outline and the interior need values of their own.

Colors accept an SVG color name (`"orange"`), an RGB hex string (`"#FDCABC"`), RGBA (`"#A0C0D10F"`), or a `JMColor` object. A shared `JMColor` instance is **live**: changing it updates every object using it.

DSL

```
//Define a color
def col = color("#A0C0D10F")

//Both c and s have the same color
def c = shape(
  type: "circle",
  style: [fillColor: col]
)
def s = shape(
  type: "square",
  style: [fillColor: col]
```

GROOVY

```
//Define a color
def col = JMColor.parse("#A0C0D10F")

//Both c and s have the same color
def c = Shape.circle().fillColor(col)
def s = Shape.square().fillColor(col)
scene.add(c, s)

//Change the alpha to 0.5. Now c
//and s changed their color
col.setAlpha(.5)
```

```

)
scene.add(c, s)

//Change the alpha to 0.5. Now c
//and s changed their color
col.setAlpha(.5)

```

Problem: Save and reuse styles

You want to define a named style and apply it to multiple objects.

Solution

Named styles are stored in the `config` object and applied with `.style(name)`, or with the `style:` parameter of any DSL block when given a String instead of a map.

DSL

```

//A style built from scratch
createStyle(name: "myStyle",
    style: [thickness: 8,
        dashStyle: "dashed",
        fillColor: "steelblue"])

//A style copied from the current
//appearance of an object
createStyle(name: "fromTriangle",
    from: triangle)

//A style derived from another one,
//overriding some properties
createStyle(name: "myStyleThick",
    from: "myStyle",
    style: [thickness: 16])

```

GROOVY

```

//A style built from scratch
def mp = config.styles["default"].copy()
mp.setThickness(8d)
mp.setDashStyle(DashStyle.DASHED)
mp.fillColor("steelblue")
config.createStyleFrom(mp, "myStyle")

//A style copied from the current
//appearance of an object
config.createStyleFrom(triangle,
    "fromTriangle")

//A style derived from another one,
//overriding some properties
def mp2 =
config.styles["myStyle"].copy()
mp2.setThickness(16d)
config.createStyleFrom(mp2,
    "myStyleThick")

```

Applying it:

DSL

```

def c = shape(
    type: "circle",
    style: "myStyle")

```

GROOVY

```

def c = Shape.circle()
    .style("myStyle")

```

In plain Groovy the registration method is `config.createStyleFrom(object, : name)`

DSL

```

def triangle = shape(
    type: "regularpolygon",
    sides: 3,
    style: [thickness: 8,
        fillColor: "steelblue"]
)
createStyle(name: "myStyle",
    from: triangle)

```

GROOVY

```

def triangle = Shape.regularPolygon(3)
    .thickness(8)
    .fillColor("steelblue")

config.createStyleFrom(triangle,
    "myStyle")

```

Tip

You do not need a named style just to copy the appearance of one object onto another. `b.getMp().copyFrom(a.getMp())` copies every drawing parameter from `a` to `b` directly.

Problem: Apply a random color

You want to create objects with random fill or draw colors.

Solution

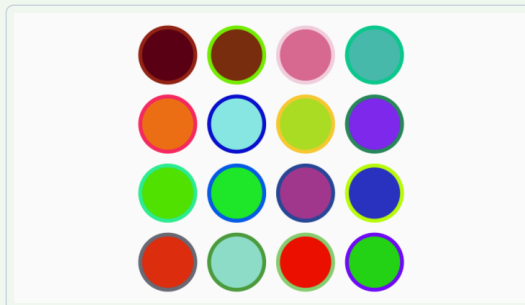
Use the special name `random` as a color value.

DSL

```
def mg = (0..<16).collect {  
  shape(  
    type: "circle",  
    transform: [scale: .2],  
    style: [  
      fillColor: 'random',  
      drawColor: 'random',  
      thickness: 30  
    ]  
  )  
} as MathObjectGroup  
layout(  
  apply: mg,  
  type: "box",  
  size: 4,  
  gaps: .1  
)  
scene.add(mg)  
camera.adjustToAllObjects()
```

GROOVY

```
def mg = MathObjectGroup.make()  
(0..<16).each {  
  mg.add(  
    Shape.circle()  
      .scale(.2)  
      .fillColor("random")  
      .drawColor("random")  
      .thickness(30)  
  )  
}  
mg.setLayout(BoxLayout.make(4, .1, .1))  
scene.add(mg)  
camera.adjustToAllObjects()
```



You can also use a `ColorScale` object to select random colors from it.

DSL

```
def cs = colorScale(  
  name: "viridis",  
  from: 0,  
  to: 1  
)  
  
def mg = (0..<16).collect {  
  shape(  
    type: "circle",  

```

GROOVY

```
def cs = ColorScale.viridis(0, 1)  
def mg = MathObjectGroup.make()  
(0..<16).each {  
  mg.add(  
    Shape.circle()  
      .scale(.2)  
      .fillColor(cs(Math.random()))  
  )  
}
```

```

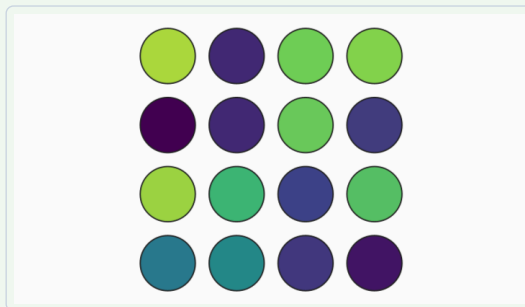
    transform: [scale: .2],
    style: [
        fillColor: cs(Math.random()),
    ]
)
} as MathObjectGroup
layout(
    apply: mg,
    type: "box",
    size: 4,
    gaps: .1
)
scene.add(mg)
camera.adjustToAllObjects()

```

```

mg.setLayout(BoxLayout.make(4, .1, .1))
scene.add(mg)
camera.adjustToAllObjects()

```



Discussion

A `ColorScale` object maps a number to a color. In this example we use the predefined `ColorScale.viridis` from 0 to 1. The method `Math.random()` is a standard Java method that returns a random number between 0 and 1, and `cs(t)` is a Groovy extension for the method `cs.getColorAt(t)`.

Problem: Apply a gradient to an object

You want to use a linear or radial gradient instead of a solid color.

Solution

A gradient is a `PaintStyle`, exactly like a plain color, so it can be passed to `fillColor` and `drawColor` wherever a color is accepted. Build a linear gradient from its two endpoints and add the color stops, given as values of the parameter between 0 and 1:

DSL

```

def lg = linearGradient(
    from: [0, 0], to: [1, 0],
    0.0: "orange",
    0.75: "blue",
    1: "violet"
)
def sq = shape(type: "centeredsquare",
    style: [fillColor: lg],
    transform: [scale: 1.5, rotate: PI/3])
scene.add(sq)

```

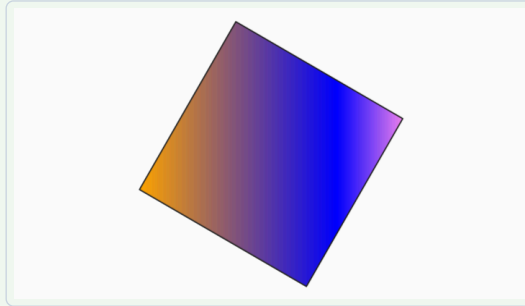
GROOVY

```

def lg = JMLinearGradient.make(
    Vec.to(0, 0), Vec.to(1, 0))
lg.add(0d, "orange")
    .add(0.75d, "blue")
    .add(1d, "violet")

def sq = Shape.square()
    .center()
    .scale(1.5).rotate(PI/3)
    .fillColor(lg)
scene.add(sq)

```



A radial gradient is given by its center and radius instead:

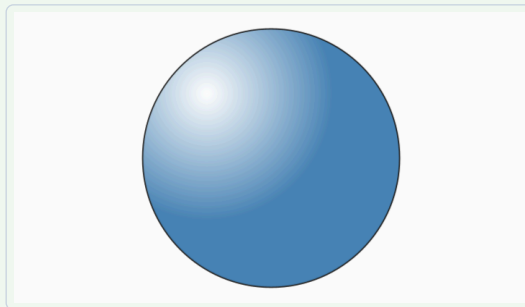
DSL

```
def rg = radialGradient(
  center: [.25, .75], radius: .5,
  0.0: "white",
  1.0: "steelblue"
)
def circle = shape(type: "circle",
  style: [fillColor: rg],
  addToScene: true)
```

GROOVY

```
def rg = JMRadialGradient.make(
  Vec.to(.25, .75), .5)
rg.add(0d, "white")
  .add(1d, "steelblue")

def circle = Shape.circle()
  .fillColor(rg)
scene.add(circle)
```



The `relative` flag decides how the coordinates are read. With `relative: true` (the default) they are relative to the bounding box of the shape, so the gradient follows the object when it moves or scales. With `relative: false` they are absolute math coordinates, which is what you want for a gradient shared by several objects or used as the background:

DSL

```
config.setBackgroundColor(
  linearGradient(
    from: scene.mathView.lower,
    to: scene.mathView.upper,
    relative: false,
    0.0: "midnightblue",
    1.0: "deepskyblue3")
)
latex(
  text: r"Good morning,\ \ sunshines!",
  addToScene: true,
  transform: [scale: 3],
  style: [fillColor: 'navajowhite2',
    drawColor: 'black',
    thickness: 3]
)
```

GROOVY

```
def bg = JMLinearGradient.make(
  scene.mathView.lower,
  scene.mathView.upper)
  .setRelativeToShape(false)
bg.add(0d, "midnightblue")
  .add(1d, "deepskyblue3")
config.setBackgroundColor(bg)

def t = LatexMathObject.make(
  r"Good morning,\ \ sunshines!")
  .scale(3)
  .fillColor("navajowhite2")
  .drawColor("black")
  .thickness(3)
scene.add(t)
```

Good morning,
sunshines!

Instead of listing the stops one by one, a predefined color scale can fill them in:

DSL

```
def sq = shape(
  type: "centeredsquare",
  transform: [scale: [2, 1], rotate:
15*DEGREES]
)
def lg = linearGradient(
  from: [.75, 0],
  to: [1, 1],
  colorScale: ["viridis", 0, 1]
)
//You can use apply to apply properties
to an object
//this is the same as sq.fillColor(lg)
apply(
  obj: sq,
  style: [fillColor: lg]
)
scene.add(sq)
```

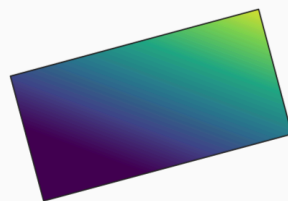
GROOVY

```
def sq =
  Shape.square().center().scale(2,
1).rotate(15 * DEGREES)

def lg =
  JMLinearGradient.make(Vec.to(.75, 0),
Vec.to(1, 1))
    .setRelativeToShape(true)
    .setColorScale(ColorScale.viridis(0,
1))

sq.fillColor(lg)

scene.add(sq)
```



Currently the implemented color scales are `viridis`, `magma`, `inferno`, `plasma`, `turbo`, `rainbow`, `grayscale`, and `bluered`. You can always see them in the autocomplete inside a DSL block. Of course you can define your own color scales. I recommend using `viridis` as it is a [color scale suitable for colorblindness](#).

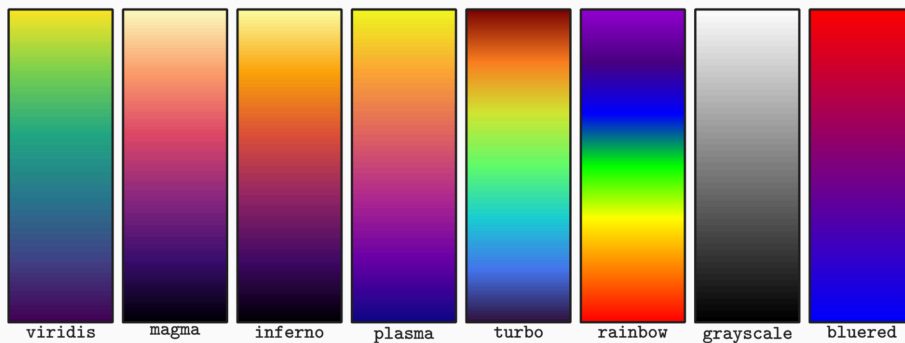
```
def colorSc = ["viridis", "magma", "inferno", "plasma",
  "turbo", "rainbow", "grayscale", "bluered"]

def groupSquares = colorSc.collect { createColorScaleBar(it) } as MathObjectGroup

layout(apply: groupSquares, type: "simple", layout: "uright", size: 3, gaps: 0.1)

scene.add(groupSquares)
camera.adjustToAllObjects()
```

```
def createColorScaleBar(String name) {
  shape(
    type: "square",
    transform: [scale: [1, 3]],
    style: [
      fillColor:
        linearGradient(
          from: [0, 0],
          to: [0, 1],
          colorScale: [name, 0, 1]
        )
    ],
    label: [text: "\\tt $name", side: 0]
  )
}
```



Tip

`cycle:` (`no_cycle`, `reflect`, or `repeat`) controls what happens beyond the endpoints of the gradient. `reflect` and `repeat` turn a two-stop gradient into a striped pattern.

Problem: Change the dash style of a shape

You want to draw a shape with dashed, dotted, or dash-dotted lines.

Solution

Use the `.dashStyle` method, or the `dashStyle` key of a style map. The available values are `SOLID` (the default), `DASHED`, `DOTTED`, and `DASHDOTTED`.

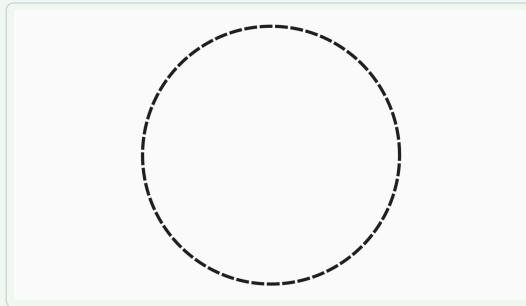
DSL

```
def c = shape(
  type: "circle",
  style: [dashStyle: "dashed",
    thickness: 25],
  addToScene: true
)
```

GROOVY

```
def c = Shape.circle()
  .dashStyle(DashStyle.DASHED)
  .thickness(25)

scene.add(c)
```



As with every keyword value in the DSL, the name is case-insensitive and can be abbreviated as long as the prefix is unambiguous, so `dashStyle: "dashdot"` selects `DASHDOTTED`.

Dashed strokes are frequently paired with a named style, so that construction lines share one definition across the whole animation:

DSL

```
createStyle(name: "guide",
  style: [drawColor: "gray",
    thickness: 2,
    dashStyle: "dotted"])

def helper = shape(type: "segment",
  from: [0, 0], to: [2, 1],
  style: "guide")
```

GROOVY

```
def mp = config.styles["default"].copy()
mp.drawColor("gray")
mp.setThickness(2d)
mp.setDashStyle(DashStyle.DOTTED)
config.createStyleFrom(mp, "guide")

def helper = Shape.segment(
  Point.at(0, 0), Point.at(2, 1))
  .style("guide")
```

Problem: Highlight one term of a formula

You want to colour, box or flash a single term of a LaTeX expression, without counting glyphs by hand and without the count breaking when you edit the formula.

Solution

Mark the term in the source with the `\jmtag{name}{...}` macro. The name travels with the glyphs it wraps, so everything afterwards refers to **the term**, not to a range of indices that the next edit would invalidate.

```
def fml = latex(
  text: r"$(a+b)^2=\jmtag{sq}{a^2}+\jmtag{cross}{2ab}+\jmtag{sq}{b^2}$",
  transform: [scale: 2],
  colorToTags: [["steelblue", "sq"], ["coral", "cross"]],
  addToScene: true)
```

$$(a + b)^2 = a^2 + 2ab + b^2$$

The same names give you the glyphs themselves, as a LaTeX object holding exactly that term:

```
def term = fml.getShapesOfTag("cross")

//A box around it
scene << shape(type: "rectangle",
  rect: term.boundingBox.addGap(.1, .1),
  style: [drawColor: "coral", thickness: 3])

//A contour flashing over the term
highlight(obj: term, type: "contour", box: true, gap: .1,
  style: [color: "gold", thickness: 12], runtime: 1.5)

//Or animate the term alone: the rest of the formula does not move
animStyle(obj: term, style: [color: "firebrick"], runtime: .8)
```

Discussion

`getShapesOfTag` does not copy anything: the returned object holds the very same glyph shapes the formula is drawing. Styling or animating it therefore styles the formula, which is what makes it usable as an animation target. The reverse also holds: do not add it to the scene, or those glyphs would be drawn twice.

Tip

Two regions may carry the same name, as `sq` does above, and they are then selected together. Regions also nest, so a tag inside another one belongs to both.

Note

Tags need the internal LaTeX engine, the one used by default. A formula compiled with an external `latex` back-end knows nothing about the macro, and `getShapesOfTag` comes back empty.

Tip

When the term is a whole numbered argument of a template formula, or a subscript, or every digit, a `latexStyle(...)` rule matching that condition saves writing the tags. See the `latexStyle` DSL for the full list of conditions.

Problem: Dim everything except one object

You want to draw attention to one object by fading the rest of the scene, and then to bring everything back exactly as it was.

Solution

Fade the others by animating only the alphas, and keep a copy of their style so that the way back does not have to guess what they looked like.

```
def spotlight = { MathObject... keep ->
  def rest = scene.mathObjects.findAll { o -> !keep.any { k -> k.is(o) } }
  def saved = rest.collect { it.mp.copy() } //style of each one
```

```

animStyle(obj: rest, style: [drawAlpha: .12, fillAlpha: .08], runtime: .5)

//what comes back is the way to undo it
return {
    animGroup(anim: [rest, saved].transpose().collect { o, mp ->
        animStyle(obj: o, from: mp, run: false, runtime: .5)
    })
}
}

```

From then on, focusing on something and restoring the scene are one line each:

```

def back = spotlight(fml)
highlight(obj: fml)
back()

```

Discussion

Giving `animStyle` an alpha with no colour means “keep the colour, change how transparent it is”, so a single call dims objects of any colour. The restoring animation instead uses `from:`, which animates towards a whole saved style, alphas, colours, thickness and all.

Note

`scene.mathObjects` lists what the scene actually draws. Adding a `MathObjectGroup` adds its members, not the group, so keeping a group out of the fade means passing its elements: `spotlight(*myGroup.objects)`. A compound object, on the other hand, enters the scene as one object and can be passed as it is.

Tip

Layers give a coarser version of the same effect with no bookkeeping at all: put the background material on its own layer and dim it with `apply(obj: scene.getObjectsFromLayers(0), style: [drawAlpha: .12])`.

Tip

You can add a blur effect in the objects you want to dim with `renderEffects: [blur: 1]` in the `animStyle` DSL block.

Positioning Objects

Problem: Center an object on the screen

You want an object to be centered on the screen.

Solution

Use the `.center()` method or `center: true` in a DSL block.

DSL

```
def sq = shape(
  type: "square",
  transform: [center: true]
)
```

GROOVY

```
//Square from (0,0) to (1,1),
//center at (1/2,1/2)
def sq = Shape.square()

//Now the square is centered
sq.center()
```

Problem: Put at a given position

You want to put an object at a given point of the math view.

Solution

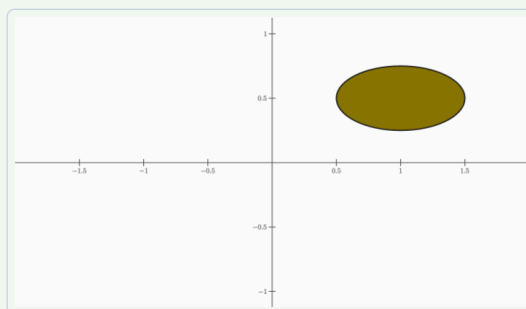
Use the `.moveTo` command.

DSL

```
axes(range: [-2, 2], xStep: .5,
yStep: .5)
def circle = shape(
  type: "circle",
  transform: [scale: [.5, .25], moveTo:
[1, .5]],
  style: [fillColor: 'gold4'],
  addToScene: true
)
```

GROOVY

```
scene.add(
  Axes.make(-2, 2)
    .generatePrimaryXTicks(-2, 2, .5)
    .generatePrimaryYTicks(-2, 2, .5)
)
Shape circle = Shape.circle()
  .scale(.5, .25)
  .fillColor("gold4")
  .moveTo(1, .5)
scene.add(circle)
```



The `moveTo` command shifts the object so that its center matches the given coordinates. In general, when we talk about the center of an object in JMathAnim we mean the center of its bounding box, not its geometric center.

Note

This command is equivalent to `circle.center = [1, .5]`.

Problem: Move or transform an object without animation

You want to reposition, rotate, or resize an object instantly.

Solution

Use the `.shift`, `.moveTo`, `.rotate`, or `.scale` commands. Any `MathObject` in JMathAnim (and in general any object implementing the `AffineTransformable` interface) admits these methods.

DSL

```
//Blue circle scaled uniformly .2
def c1 = shape(type: "circle",
  style: [drawColor: 'blue'],
  transform: [scale: .2])

//Red circle scaled .2 horizontally
//and .5 vertically
def c2 = shape(type: "circle",
  style: [drawColor: 'red'],
  transform: [scale: [.2, .5]])

//Green regular pentagon scaled .3
//and shifted with vector (-1,0)
def pent1 = shape(
  type: "regularinscribedpolygon",
  sides: 5,
  style: [drawColor: 'green'],
  transform: [scale: .3,
    shift: [-1, 0]])

//Violet pentagon scaled .5,
//center moved to (.75,.25)
def pent2 = shape(
  type: "regularinscribedpolygon",
  sides: 5,
  style: [drawColor: 'violet'],
  transform: [scale: .5,
    moveTo: [.75, .25]])

//Orange-filled square scaled .25,
//rotated  $\pi/4$  around (-.5,-.5)
def sq1 = shape(type: "square",
  style: [fillColor: 'orange'],
  transform: [scale: .25])
//Rotation around a point, in Groovy
sq1.rotate(Vec.to(-.5, -.5),  $\pi/4$ )

//Gold-filled square scaled .25,
//rotated  $-\pi/3$  around its center
def sq2 = shape(type: "square",
  style: [fillColor: 'gold'],
  transform: [scale: .25,
```

GROOVY

```
//Blue circle scaled uniformly .2
def c1 = Shape.circle()
  .drawColor("blue")
  .scale(.2)

//Red circle scaled .2 horizontally
//and .5 vertically
def c2 = Shape.circle()
  .drawColor("red")
  .scale(.2, .5)

//Green regular pentagon scaled .3
//and shifted with vector (-1,0)
def pent1 =
  Shape.regularInscribedPolygon(5)
  .drawColor("green")
  .scale(.3).shift(-1, 0)

//Violet pentagon scaled .5,
//center moved to (.75,.25)
def pent2 =
  Shape.regularInscribedPolygon(5)
  .drawColor("violet")
  .scale(.5).moveTo(.75, .25)

//Orange-filled square scaled .25,
//rotated  $\pi/4$  around (-.5,-.5)
def sq1 = Shape.square()
  .fillColor("orange")
  .scale(.25)
  .rotate(Vec.to(-.5, -.5),  $\pi/4$ )

//Gold-filled square scaled .25,
//rotated  $-\pi/3$  around its center
def sq2 = Shape.square()
  .fillColor("gold")
  .scale(.25).rotate(- $\pi/3$ )

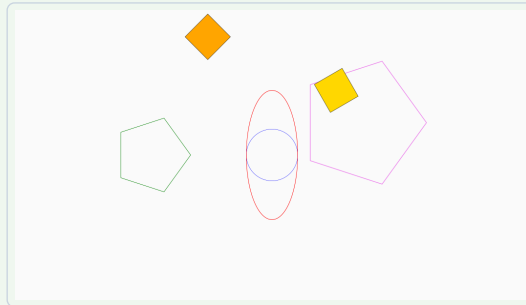
scene.add(c1, c2, pent1, pent2, sq1,
sq2)
```

```

    rotate: -PI/3])

scene.add(c1, c2, pent1, pent2, sq1,
sq2)

```



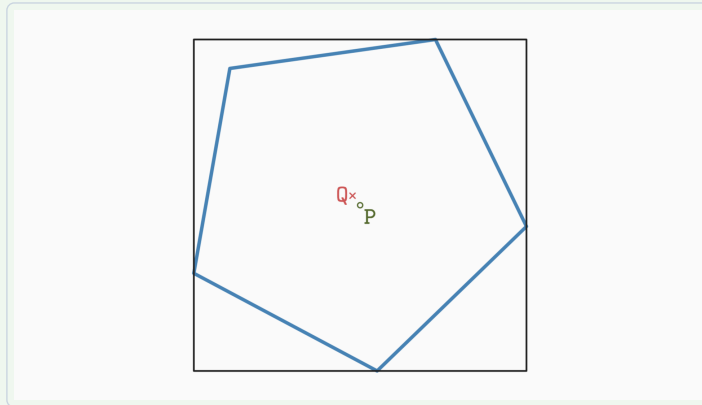
Note

In these transforms, **center** always means the center of the bounding box of the object, not the geometrical center of the object itself. In the following example, the crossed dot is the geometrical center of the pentagon and the circular dot the center of its bounding box, which is the one used in transforms.

```

def pent = shape(
    type: "regularpolygon",
    sides: 5,
    transform: [rotate: -28 * DEGREES, center: true, scale: 1.2],
    style: [thickness: 20, drawColor: 'steelblue'],
)
def bbox = shape(
    type: "rectangle",
    rect: pent.boundingBox
)
def P = point( //Center of bounding box
    at: pent.center,
    style: [dotStyle: "ring", color: 'darkolivegreen'],
    label: [text: r"\tt P", scale: .75, gap: .5,
        anchor: "diag4",
        color: 'darkolivegreen']
)
def Q = point( //Geometric center
    at: pent.centroid,
    style: [dotStyle: "cross", color: 'indianred3'],
    label: [text: r"\tt Q", scale: .75,
        gap: .25, anchor: "left",
        color: 'indianred3']
)
scene.add(pent, bbox, P, Q)

```



Problem: Position an object relative to another

For example, you have 2 objects and want to position one to the right of the other.

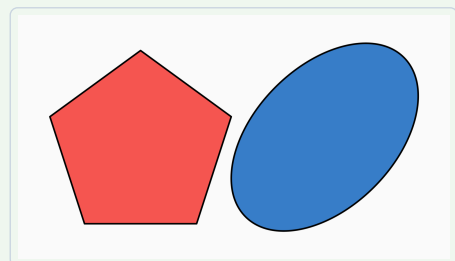
Solution

Method 1 Setting the boundary points of the objects

```
//Red pentagon
def pentagon = shape(
    type: "regularpolygon",
    sides: 5,
    style: "solidred",
    transform: [scale: .4],
)
//Blue ellipse, rotated -45 degrees
def ellipse = shape(
    type: "circle",
    style: "solidblue",
    transform: [scale: [.25, .4], rotate: -PI/4]
)

//Magic happens here!
ellipse.left = pentagon.right
//Ellipse is now stacked to the right of the
pentagon

scene.add(pentagon, ellipse)
camera.zoomToAllObjects()
```



Discussion

Any `Boxable` object (which includes `Mathobjects`, `Rects` or `Camera` objects) has the following getters and setters methods for the properties `center`, `right`, `left`, `upper`, `lower`, `lowerRight`, `lowerLeft`, `upperRight`, `upperLeft` which represents the respective reference point of its bounding box. They can be gotten or setted in the usual Java convention (`.getLeft()` or `.setLeft(Vec.to(.4, .3))` for example) or using Groovy shortcuts for getters/setters.

Setting one of these properties shifts the object to match the given location. For example `obj.left=Vec.to(2,3)` shifts `obj` so that its left point of the bounding box is located at (2, 3).

Note

Remember that Groovy has a nice way to provide simpler access to getters/setters. The code `def c=obj.right` is equivalent to `def c=obj.getRight()` and `obj.left=Vec.to(2, 3)` is equivalent to `obj.setLeft(Vec.to(2, 3))`

Note

The Groovy script interpreter has some useful overloads to create auxiliary objects easily. The following commands have all the same effect:

```
obj.left = Vec.to(1, 0) //A vector
obj.left = Point.at(1, 0) //A point
obj.left = [1, 0] //A 2-list
```

Tip

The editor has a built-in method autocompletion that you can invoke with `ctrl+shift+G` after the name of the variable. In some cases, JMathAnim will recognize what type of variable is and will offer you methods to complete. If it can't identify, it will show you all available options.

Method 2 Use `stack` command

Use the `stack` command: the `stack:` parameter of a DSL block, or the `stack()` method in plain Groovy. In Groovy the syntax is to invoke the utility class with `object.stack()`, add some stacking parameters, and finish and execute the stack with the `toObject` or `toScreen` method.

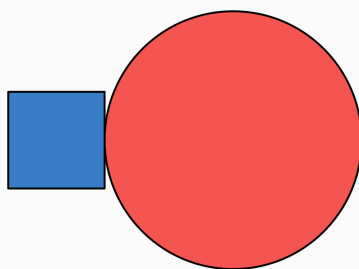
DSL

```
//Red circle
def c = shape(
    type: "circle",
    style: "solidred",
)
//Blue square, stacked
//to the LEFT of c
def s = shape(
    type: "square",
    style: "solidblue",
    transform: [scale: .75],
    stack: [to: c, destinyAnchor: "left"]
)
scene.add(c, s)
```

GROOVY

```
//Red circle
def c = Shape.circle()
    .style("solidred")
//Blue square
def s = Shape.square()
    .scale(.75).style("solidblue")

//Stack the object s to the LEFT of c
s.stack()
    .withDestinyAnchor(AnchorType.LEFT)
    .toObject(c)
scene.add(c, s)
```



The anchor of the origin object is automatically computed as the “reverse” of the destiny. In the previous example, stacking `s` to the `LEFT` of `c` means “move `s` so that `RIGHT` of `s` and `LEFT`

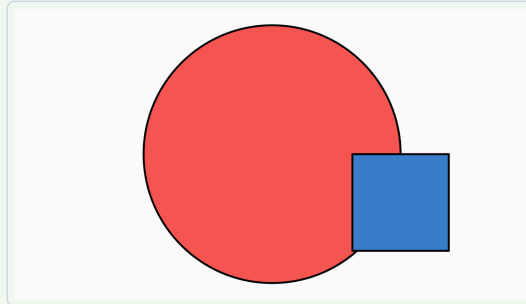
of `c` match". You can specify the origin anchor as well. For example, position `s` so that its `UPPER` point matches the `RIGHT` side of `c`. As `s` already exists, the DSL side stacks it with the `stack` extension, which takes the same named parameters directly on the object:

DSL

```
s.stack(to: c,
        destinyAnchor: "right",
        originAnchor: "upper")
scene.add(c, s)
```

GROOVY

```
s.stack()
    .withDestinyAnchor(AnchorType.RIGHT)
    .withOriginAnchor(AnchorType.UPPER)
    .toObject(c)
scene.add(c, s)
```



Tip

The same stacking can be written with the general-purpose `apply` command, which is worth it when you are already changing other things about the object in one go:

```
apply(
    obj: s, //it also admits a list of objects
    stack: [to: c, destinyAnchor: "right", originAnchor: "upper"],
    style: [fillColor: 'gold']
)
```

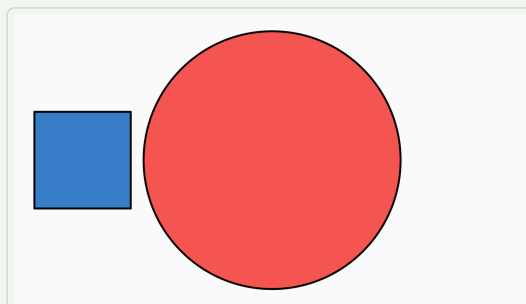
The `gaps` parameter (`.withGaps(hgap, vgap)` in Groovy) defines gaps between the objects in mathematical units. Depending on the anchor type, not all gaps are used: for horizontal stacking like `LEFT` or `RIGHT`, vertical gaps are not applied.

DSL

```
//Stack the object s to the LEFT of c
s.stack(to: c,
        destinyAnchor: "left",
        gaps: .1)
scene.add(c, s)
```

GROOVY

```
//Stack the object s to the LEFT of c
s.stack()
    .withGaps(.1, .1)
    .withDestinyAnchor(AnchorType.LEFT)
    .toObject(c)
scene.add(c, s)
```



The following commands allow defining gaps:

```

.withGaps(.1, .1) //Horizontal and vertical
.withGaps(.1) //Same as before
.withRelativeGaps(.1, .1) //Gaps relative to the size of the object
.withRelativeGaps(.1) //Same as before

```

Stacking does not need a separate step: give the `stack:` parameter when the object is created and it is born already in place.

DSL

```

def pentagon = shape(
  type: "regularpolygon",
  sides: 5,
  style: "solidorange",
  transform: [center: true],
)
def text = latex(
  text: "pentagon",
  transform: [scale: 1.5],
  stack: [gaps: .1,
    destinyAnchor: "lower", to:
pentagon]
)
scene.add(text, pentagon)

```

GROOVY

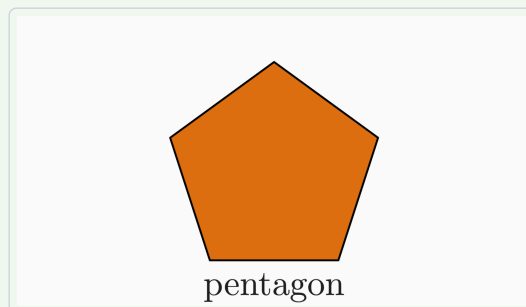
```

def pentagon = Shape.regularPolygon(5)
  .style("solidorange")
  .center()

def text =
  LatexMathObject.make("pentagon")
    .scale(1.5)
    .stack()
    .withGaps(.1)
    .withDestinyAnchor(AnchorType.LOWER)
    .toObject(pentagon)

scene.add(text, pentagon)

```



Tip

The integrated editor has built-in autocompletion for `AnchorType` values, as seen in Figure 3, with a sample image.

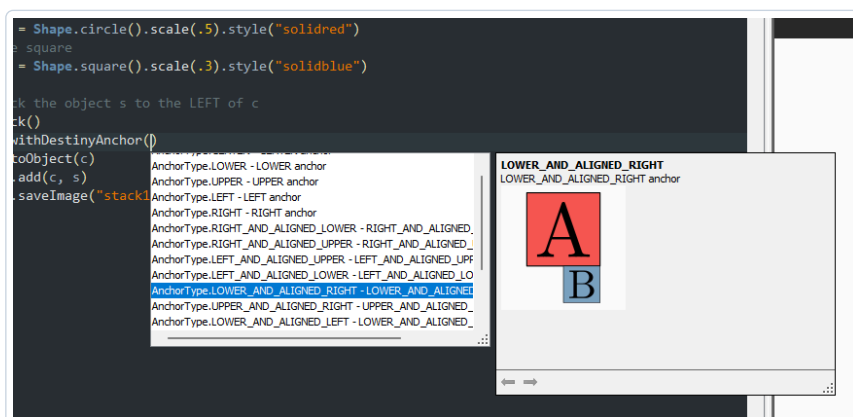


Figure 3: Autocompletion of `AnchorType` values

Problem: Move an object so that a corner matches a given location

For example, you want to move a square so that its upper left corner matches given coordinates.

Solution

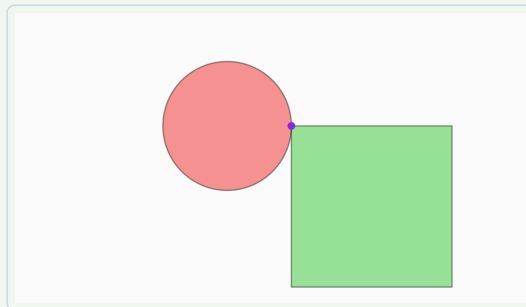
Use the `stack` command. This is a versatile command that admits several options to properly stack an object relative to another object, coordinates, or the screen.

DSL

```
def P = point(  
  at: [.15, .25],  
  style: [thickness: 60, drawColor:  
    'blueviolet']  
)  
def sq = shape(  
  type: "square",  
  style: [fillColor: 'limegreen',  
    alpha: .5],  
  transform: [scale: 1.25],  
  stack: [to: P, originAnchor: "diag2"]  
)  
def ci = shape(  
  type: "circle",  
  transform: [scale: .5],  
  stack: [to: P, originAnchor: "right"],  
  style: [fillColor: 'firebrick2',  
    alpha: .5]  
)  
scene.add(sq, ci, P)
```

GROOVY

```
def P = Point.at(.15, .25)  
  .thickness(60)  
  .drawColor('blueviolet')  
  
def sq = Shape.square()  
  .fillColor('limegreen')  
  .scale(1.25)  
  .alpha(.5)  
  .stack()  
  .withOriginAnchor(AnchorType.DIAG2)  
  .toObject(P)  
  
def ci = Shape.circle()  
  .fillColor('firebrick2')  
  .scale(.5)  
  .alpha(.5)  
  .stack()  
  .withOriginAnchor(AnchorType.RIGHT)  
  .toObject(P)  
  
scene.add(sq, ci, P)
```



Problem: Stack an object to a side or corner of the screen

You want to put text on one corner of the screen.

Solution

Use the `stack` command with the `toScreen` method.

DSL

```
def t1 = latex(  
  text: r"Upper Left",  
  stack: [screen: "upper_left",  
    gaps: .1],  
)
```

GROOVY

```
def t1 = LatexMathObject.make(r'Upper  
Left')  
  .scale(2)  
  .stack()
```

```

    transform: [scale: 2]
  )

  def t2 = latex(
    text: r"Lower Right",
    stack: [screen: "lower_right",
            gaps: .1],
    transform: [scale: 2]
  )

  scene.add(t1, t2)

```

```

    .withGaps(.1, .1)
    .toScreen(ScreenAnchor.UPPER_LEFT)

  def t2 = LatexMathObject.make(r'Lower
  Right')
    .scale(2)
    .stack()
    .withGaps(.1, .1)
    .toScreen(ScreenAnchor.LOWER_RIGHT)

  scene.add(t1, t2)

```

Upper Left

Lower Right

Problem: Align a group of objects

You want to align several objects along a common edge.

Solution

Aligning moves an object until one of its edges matches the corresponding edge of a reference object, without touching the other coordinate. The method is `.align(reference, AlignType)`, and the available types are `LEFT`, `RIGHT`, `UPPER`, `LOWER`, `HCENTER`, and `VCENTER`.

DSL

```

def a = shape(
  type: "square",
  style: "solidred",
  transform: [moveTo: [.67, -.32]]
)
def b = shape(
  type: "circle",
  style: "solidblue",
  transform: [
    scale: .25, //Order of transforms
    moveTo: [-.5, 1], //matters!
    align: [a, "upper"] //last one,
  ],
  align
)
def c = line(
  a: [0, 0],
  b: [1, 0],
  transform: [align: [a, "upper"]]
)
scene.add(a, b, c)

```

GROOVY

```

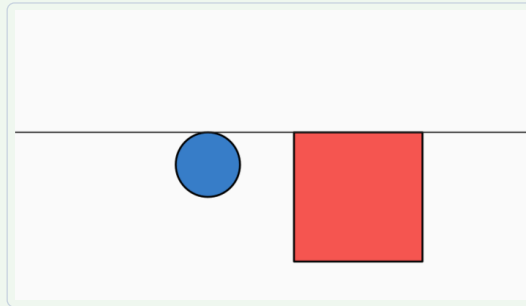
def a = Shape.square()
  .style("solidred")
  .moveTo(.67, -.32)

def b = Shape.circle()
  .style("solidblue")
  //Order of transforms matters!
  .scale(.25)
  .moveTo(-.5, 1)
  //last one, align
  .align(a, AlignType.UPPER)

def c = Line.make(
  Point.at(0, 0), Point.at(1, 0))
  .align(a, AlignType.UPPER)

scene.add(a, b, c)

```



Note

Alignment always refers to the bounding box of the object, not to its geometrical outline. For a rotated shape, the box is the one that encloses it in the current position.

Note

Aligning objects with special bounding boxes like `line` or `ray` may lead to unexpected behavior. In the previous example, a horizontal line worked well, but if you replace the second point with something like `b:[1,1]` the `align` method will move the line so that the *visible* bounding box at the time is upper aligned with `a`, resulting in strange camera-dependent behavior.

Problem: Creating a mesh of objects

For example, you want to create a mesh of points distributed in a 4x5 Cartesian grid.

Solution

You can achieve this in several ways:

Method 1 In a straightforward way, with 2 nested loops

```
for (i in 0..<5) {
  for (j in 0..<4) {
    scene.add(Point.at(i,j))
  }
}
camera.adjustToAllObjects()
```

Method 2 Similar, but in a more Groovy way, using the overloaded operator `*`

```
def dot = point()
def grid = ((0..<5)*(0..<4)).collect { dot + 0.5*it } as MathObjectGroup
grid.center()
scene.add(grid)
```

The code `(0..<5)*(0..<4)` deserves an explanation: in Groovy, `0..<5` returns the list `[0, 1, 2, 3, 4]`, and similarly `0..<4` returns `[0, 1, 2, 3]`. The JMathAnim Groovy interpreter has an extension defined for the binary operator `*` for lists, which computes in this case the Cartesian product: `[[0, 0], [0, 1], ..., [4, 3]]`. Note that each of the 20 elements of this list is a pair with the coordinates of the Cartesian grid. The `collect` method can then iterate over this list to create the points. The `as MathObjectGroup` is optional, but converts the just-created

list into a `MathObjectGroup` object (this is another extension) which is better for handling groups of objects. The expression `0.5*it` is itself another extension. Inside this loop, the variable `it` takes values `[0,0]`, `[1,0]`, etc. An expression like `0.5*[2,1]` means “multiply every element of this list by 0.5”. Finally, the expression `dot+0.5*it` (another extension) means “a copy of dot shifted `0.5*it`”.

Method 3 Use `MathObjectGroup` to handle the location

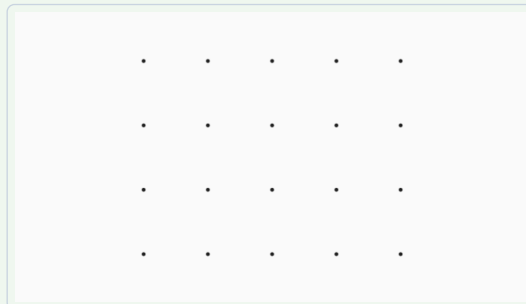
In these contexts, `MathObjectGroup` is always useful. Just create a group composed of 20 copies of the same object and apply a box layout to it.

DSL

```
def mg = group(  
  copies: point(),  
  number: 20,  
  layout: [type: "box",  
    gaps: .5,  
    size: 5],  
  transform: [center: true],  
  addToScene: true  
)
```

GROOVY

```
def mg =  
  MathObjectGroup.makeCopies(Point.origin(),  
    20)  
  .setLayout(BoxLayout.make(5, .5, .5))  
  .center()  
  scene.add(mg)
```



Discussion

The `layout` method is a powerful way to allocate several objects in predefined layouts. You can look at the available layouts in the javadocs, or find them by invoking the autocomplete in the editor for the `type` parameter in the DSL box. There is an optional parameter for this layout called `direction` to define the order of the elements inside the box. The default is `right_up`, which means “fill first row from left to right and next row goes up”.

Tip

When the elements of the mesh have to be reachable by a name of their own afterwards, a `compound` built from a keyed map is the better tool. The chapter on compound objects builds a chessboard that way.

Problem: Distributing objects in a circle

You want to distribute a group of objects forming a circle.

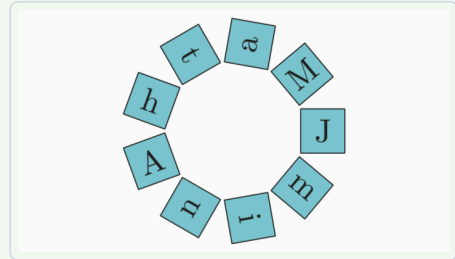
Solution

Use the `layout` command with the `type: "circular"` parameter.

```
def t = "JMathAnim" as List

def myGroup = t.collect {
  def box = shape(
    type: "square",
    transform: [scale: .5],
    style: [fillColor: 'cadetblue3']
  )
  def text = latex(
    text: it,
    transform: [scale: 2],
    stack: [to: box]
  )
  group(
    obj: [box, text]
  )
} as MathObjectGroup

myGroup.layout(type: "circular", rotation:
"smart")
scene.add(myGroup)
camera.adjustToAllObjects()
```



Tip

The `rotation` parameter determines the rotation of the elements. The `smart` value tries to rotate the element without putting it "upside down".

Tip

The `radius` and `center` parameters can be specified for this layout. If omitted, it uses `center: [0,0]` and `radius: 1`.

Problem: Distributing objects over a path

You want to put the objects of a group over a path.

Solution

Method 1 Use the `layout` command with the `type: "path"` parameter.

```
def bricks = group(
  copies: shape(type: "square"),
  number: 40,
  transform: [scale: [.2, .1]]
)

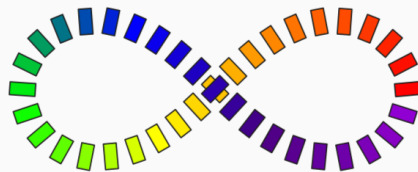
def cs = colorScale(
  name: "rainbow",
  from: 0, to: 39
)
bricks.eachWithIndex { obj, i ->
  obj.fillColor(cs(i))
}
```

```

def lemniscate = parametricCurve(
  function: { t ->
    def s = Math.sin(t)
    def c = Math.cos(t)
    def x = 1.5 * c / (1 + s * s)
    def y = s * x
    [x, y]
  },
  tRange: [0, 2 * PI]
)
bricks.layout(type: "path", path: lemniscate, rotation: "rotate")
scene.add(bricks)

camera.adjustToAllObjects()

```

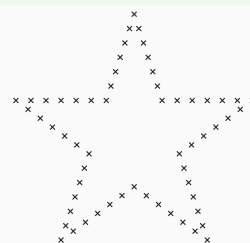


Method 2 Use Groovy loops

```

def star = shape(
  type: "logo",
  commands: "rt 18 repeat 5 [rt -72 fd 1 rt
144 fd 1 CL0]",
  transform: [center: true]
)
def points = linspace(0, 1, 60 + 1)
  .init().collect {
    point(at: star(it),
      style: [dotStyle: "cross", thickness: 40])
  } as MathObjectGroup
points.center()
scene.add(points)
camera.adjustToAllObjects()

```



Discussion

Noticed the `60+1`? The `linspace(0, 1, 60+1)` method generates a `List` with 61 numbers equally distributed from 0 to 1 (both included). In a closed path, that may be a problem, because location 0 and 1 are the same, so to avoid a duplicate point, we generate 61 numbers and remove the last one, obtaining 60 locations from 0 to 1 (the last one not included). This is accomplished with the `init()` method, which removes the last element of a `List`. Then, with the `collect` method we generate a list of `point` objects located at position `star(it)`. Finally, we convert the list into a `MathObjectGroup`, we center it on screen and adjust the camera so everything gets in the photo.

Tip

We use 60 because it is a multiple of 5. This way we ensure that a point lies exactly on the corners of the star. Mathematics, son!!

Tip

If you want to remove the first element, use `tail()` instead of `init()`.

Problem: Layout of elements of unequal size

You want to apply a layout to a list of elements of unequal size.

Solution

Use the `homogenize` parameter of `apply(...)` or `layout(...)`, which calls `homogenizeBoundingBoxes` underneath. It adjusts the internal gaps of the bounding boxes of every element of a `MathObjectGroup` so that they all end up with the same dimensions.

```
def cs = colorScale(name: "viridis", from: 1, to: 3)

//Create 16 boxes with increasing size
def normalGroup = linspace(1, 3, 16).collect {
  shape(
    type: "square",
    style: [fillColor: cs(it)],
    transform: [scale: .1 * it]
  )
} as MathObjectGroup

def homogenizedGroup = normalGroup.copy()

apply{//Homogenize bounding boxes
  obj: homogenizedGroup,
  homogenize: [anchor: "center", gaps: 0]
}

layout{//Apply the same box layout to both groups
  apply: [homogenizedGroup, normalGroup],
  type: "box",
  size: 4,
  gaps: [0.1, 0.1],
}

apply{//stack normalGroup to the left of homogenizedGroup
  obj: normalGroup,
  stack: [to: homogenizedGroup, destinyAnchor: "left", gaps: 0.5]
}

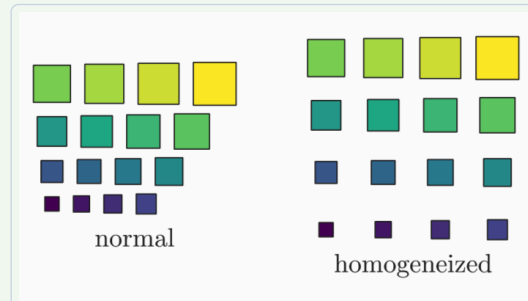
["normal": normalGroup,
 "homogenized": homogenizedGroup].collect { name, gr ->
  latex(
    text: name,
    stack: [to: gr, destinyAnchor: "lower", gaps: .1],
    addToScene: true
  )
}
```

```

    )
}
scene.add(homogeneizedGroup, normalGroup)

camera.adjustToAllObjects()

```



Discussion

Most `layout` methods work “one step at a time”, stacking one object to the previous one in a defined way. In the case of the `box` layout, for example, with the `right_up` parameter, it stacks an object to the right of the previous one until filling a row. In this case, the next object will be stacked above the first object of the previous row. This method is easy, but only takes into account the sizes of the current and previous objects, not the “global” sizes.

Tip

`homogenizeBoundingBoxes` uses the `.setGaps` method on each object to set its internal gaps.

```

def t = latex(
  text: "A",
  transform: [scale: 12]
)
def b1 = shape(//Initial bounding box
  type: "rectangle",
  rect: t.boundingBox,
  style: [drawColor: 'darkgoldenrod3']
)
//The order is upper, right, lower, left (like a
clock...)
t.setGaps(.1, .2, .3, .8)
def b2 = shape(//Bounding box after setGaps
  type: "rectangle",
  rect: t.boundingBox,
  style: [drawColor: 'darkolivegreen']
)
scene.add(t, b1, b2)

```



Problem: Align LaTeX objects to their baselines

You have some LaTeX objects and want to put them vertically aligned.

Solution

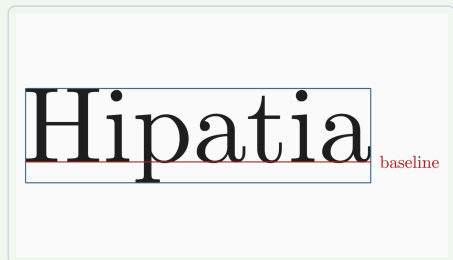
Use the `align` command with the `baseline` parameter. A LaTeX object (well, any glyph object) has a “height” and a “depth”, and the baseline is the “horizon” between them. All the glyphs of a word, for example, are vertically aligned by their baselines. This makes no difference for a glyph like “a”, but it does for descenders like “g”, “p” or “y”. You can draw the baseline of a LaTeX expression with the following code:

```
def t = latex(text: "Hipatia",
  transform: [height: 2],
  addToScene: true
)

scene << shape(type: "rectangle", rect:
t.boundingBox,
  style: [drawColor: 'steelblue4'])

def baseline =
t.baselineSegment.drawColor("firebrick")

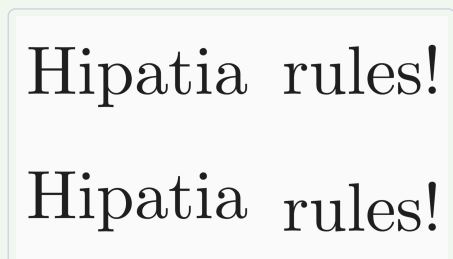
scene << baseline
latex(text: "baseline",
  style: [color: 'firebrick'],
  stack: [to: baseline,
  destinyAnchor: "right",
  gaps: .1],
  addToScene: true
)
```



To align LaTeX expressions properly, use the `baseline` anchor and its variants `baseline_left` and `baseline_right`. You can compare the result obtained with `right_and_aligned_lower` or any other anchor you want:

```
def t1 = latex(text: "Hipatia",
  transform: [scale: 3])
def t2 = latex(text: "rules!",
  transform: [scale: 3])
t2.stack(to: t1,
  destinyAnchor: "baseline_right",
  gaps: [.3, 0])

def t3 = t1 + [0, -1] //Copy of t1 shifted (0,-1)
def t4 = t2 + [0, -1] //Copy of t2 shifted (0,-1)
t4.stack(to: t3,
  destinyAnchor: "right_and_aligned_lower",
  gaps: [.3, 0])
scene.add(t1, t2, t3, t4)
camera.adjustToAllObjects()
```



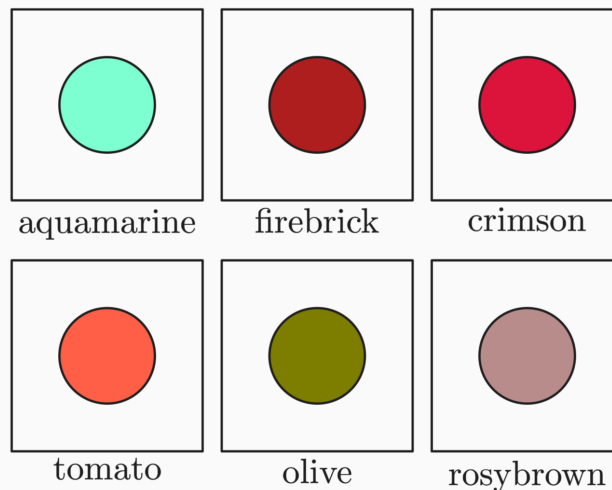
Problem: Create several boxes with titles in a box layout

For example, you want to show several cases inside a box, and want to position the boxes in a row-col layout, but you want to be sure the varying size of the titles won't affect the proper layout.

Solution

Use the `layout` command with the `homogenize` parameter.

```
def labeledBoxes = ['aqua', 'firebrick', 'crimson',  
  'tomato', 'olive', 'rosybrown'].collect {  
  [  
    shape( //the box  
      type: "centeredsquare",  
      label: [text: "$it", side: 0]  
    ),  
    shape( //the content  
      type: "circle",  
      style: [fillColor: it],  
      transform: [scale: .25]  
    )  
  ] as MathObjectGroup //Each box is a group  
} as MathObjectGroup //A group of groups  
  
def labels = labeledBoxes.collect {  
  it[0].label  
} as MathObjectGroup  
  
apply(obj: labels, homogenize: [anchor: "baseline"])  
  
labeledBoxes.layout(  
  type: "box", size: 3, gaps: .1,  
  direction: "right_down",  
  homogenize: [anchor: "upper"]  
)  
scene.add(labeledBoxes)  
camera.adjustToAllObjects()
```



Discussion

For every element of the list `['aqua', 'firebrick', ...]` we generate a `list` with 2 elements: the first is the `shape` object representing the box with its label; the second, the content, a filled circle. At the end of the list the `as MathObjectGroup` command converts it to a `MathObjectGroup`. Once the `collect` method has finished, it returns a `list` with each element a `MathObjectGroup`. Again, with `as MathObjectGroup` we convert it, obtaining that

`labeledBoxes` is a `MathObjectGroup` with its elements being `MathObjectGroup`s as well. This kind of structure is common for complex layouts and is easy to deal with once created. Note this piece of code:

```
def labels = labeledBoxes.collect {  
  it[0].label  
} as MathObjectGroup  
apply(obj: labels, homogenize: [anchor: "baseline"])
```

With this method we collect all `label` objects from the box (item 0 of the 2-list) and put them in a `MathObjectGroup`. The `apply` method homogenizes the bounding boxes of the labels, keeping their baselines vertically aligned. If you omit this piece of code, you will see that some labels sit lower than others, because of taller glyphs such as “f”.

Tip

Note the `homogenize: [anchor: "upper"]` in the `layout` method. This means “make all boxes the same size but align them by the upper side”. This is important mostly due to the presence of labels, which make the size of each element rather unpredictable.

Problem: Positioning randomly

You want to position a collection of objects in random positions on the screen.

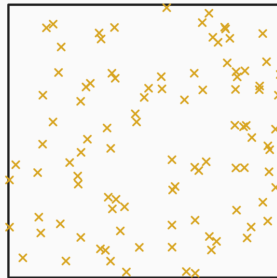
Solution

The `layout` command provides four ways of distributing randomly: `uniform` for typical random points, `normal` for a normal (Gaussian) distribution, `poisson` for a distribution where points tend to be apart from one another, and `thomas`, which does the opposite, that is, points tend to form clusters. All of them take a `Rect` as an argument to locate the objects and (except for `normal`) the objects will be located inside (well, their centers will be located inside).

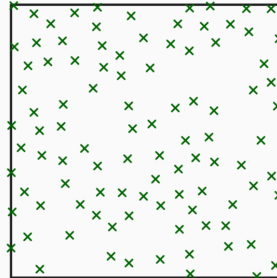
```
def distributions = [  
  'uniform': 'goldenrod',  
  'poisson': 'darkgreen',  
  'thomas': 'darkorchid4'  
]  
def randomBoxes = distributions.collect {  
  distributionName, colorName ->  
  def box = shape( //the box  
    type: "centeredsquare",  
    label: [text: "\\tt $distributionName", side: 0])  
  
  [box,  
    group(  
      copies: point(style: [dotStyle: "cross", drawColor: colorName]),  
      number: 100,  
      layout: [type: "random",  
        distribution: distributionName,  
        rect: box.boundingBox]  
    )  
  ] as MathObjectGroup //Each box is a group  
} as MathObjectGroup //A group of groups
```

```
randomBoxes.layout(
  type: "simple", layout: "uright", gaps: .1)

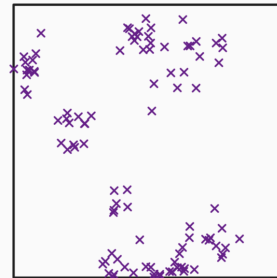
scene.add(randomBoxes)
camera.adjustToAllObjects()
```



uniform



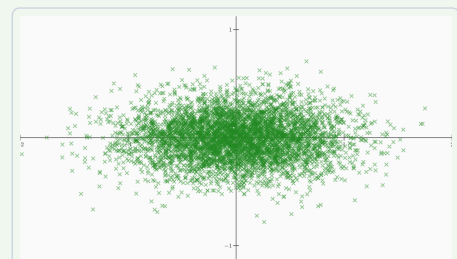
poisson



thomas

For `distribution: "normal"` it is better to use the `deviation` parameter, which is the standard deviation of the Gaussian distribution. You can use `rect` too, and its center will be taken as the center of the distribution.

```
axes(
  xRange: [-5, 5],
  yRange: [-5, 5],
  xStep: 1,
  yStep: 1
)
group(
  copies: point(
    style: [dotStyle: "cross",
    drawColor: 'forestgreen', alpha: .5]),
  number: 5000,
  layout: [type: "random",
    distribution: "normal",
    deviation: [.5, .2]],
  addToScene: true
)
```



Problem: Position an object over other objects

You want to add an object to the scene and ensure it is always drawn over other objects.

Solution

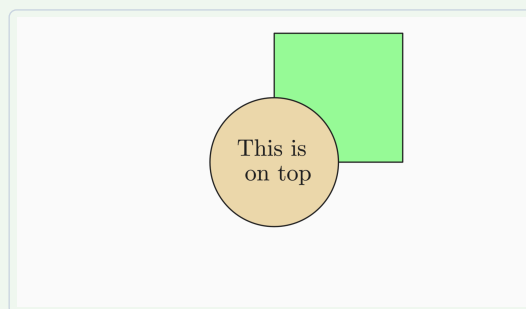
Use the `layer` command. An object with a higher layer will be drawn over objects with lower layers. Objects with the same layer number are drawn with the rule "last added to scene is located over first". Layers can be negative numbers too. The default layer number is 0.

DSL

```
def c = shape(  
    type: "circle",  
    style: [layer: 1, fillColor:  
'wheat2'],  
    transform: [scale: .5])  
def sq = shape(type: "square",  
    style: [fillColor:  
'palegreen']) //Default layer 0  
  
def t = latex(text: r'This is\\ on top',  
    style: [layer: 2])  
  
//c will always be drawn over sq  
scene.add(c, sq, t)
```

GROOVY

```
def c = Shape.circle()  
    .fillColor('wheat2')  
    .scale(.5)  
    .layer(1)  
def sq = Shape.square()  
    .fillColor('palegreen')  
def t = LatexMathObject.make(r'This is\\  
on top').layer(2)  
scene.add(c, sq, t)
```



Problem: Add a little watermark in a corner

You want to add a text in a corner of the screen over everything.

Solution

Use the `layer` and `fixedCamera` parameters. You can use `layer` with a high enough number (or if you want to be absolutely sure, `Integer.MAX_VALUE`) and `fixedCamera` ensures the object will be located in the same place if the camera moves.

DSL

```
def watermark = latex(  
    text: r"\tt #jmathanim",  
    style: [alpha: .7, layer: 9999],  
    addToScene: true,  
    stack: [screen: "lower_right",  
    gaps: .1],  
    fixedCamera: true  
)
```

GROOVY

```
def watermark =  
    LatexMathObject.make(r'\tt #jmathanim')  
        .layer(9999)  
        .alpha(.7)  
        .stack()  
        .withGaps(.1, .1)  
        .toScreen(ScreenAnchor.LOWER_RIGHT)  
        .setCamera(fixedCamera)  
scene.add(watermark)
```

Labelling Objects

Problem: Label a point

For example, you have 3 points at different coordinates and want to label them A, B, C.

Solution

Use the `label` command.

DSL

```
def A = point(
  at: [0, 0],
  label: r"$A$"
)
def B = point(
  at: [1, 0],
  label: [
    text: r"$B$",
    scale: .75,
    anchor: "left"
  ]
)
def C = point(
  at: [-.5, .75],
  label: [
    text: r"$C$",
    color: 'steelblue'
  ]
)
scene.add(A, B, C)
```

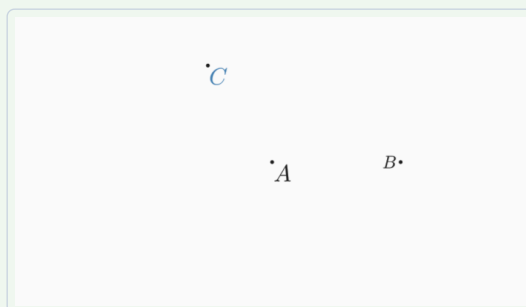
GROOVY

```
def A = Point.at(0, 0)
  .addLabel(r"$A$")

def B = Point.at(1, 0)
  .addLabel(r"$B$")
  .anchor(AnchorType.LEFT)
  .scale(.75)

def C = Point.at(-.5, .75)
  .addLabel(r"$C$")
  .color("steelblue")

scene.add(A, B, C)
```



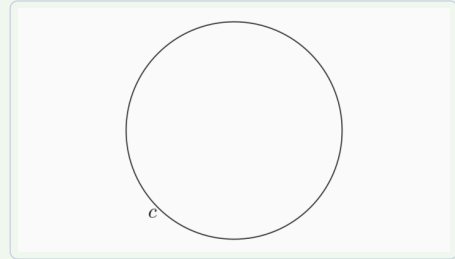
Problem: Labelling shapes

You want to apply a label to a `Shape` object.

Solution

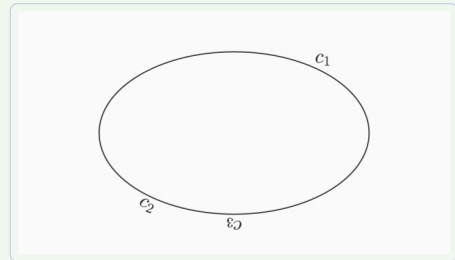
Use the `label` command. The syntax is the same as in the previous case, but it admits additional parameters.

```
def r = shape(
  type: "circle",
  label: [text: r"$c$",
    t: 0.625,
    rotation: "fixed"]
)
scene.add(r)
```



You can specify several labels by passing a list:

```
def c = shape(
  type: "circle",
  transform: [scale: [1.25, .75]],
  label: [
    [text: r"$c_1$", t: 0.125, rotation: "fixed"],
    [text: r"$c_2$", t: 0.625, rotation: "smart"],
    [text: r"$c_3$", t: 0.75, rotation: "rotate"],
  ]
)
scene.add(c)
```

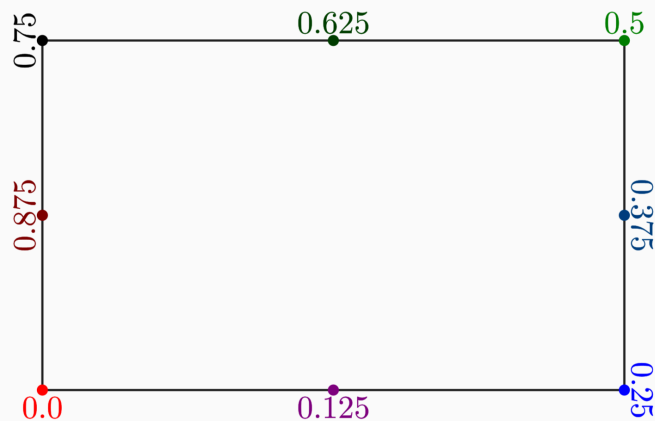


Tip

The `t` parameter is a number from 0 to 1 parametrized by the length of the curve, so that 0 represents the first point of the path, 1 the last, and 0.5 the middle point. Internally JMathAnim uses a Bézier parametrization that is different from the length parametrization, but may be useful in some cases, especially for straight paths. For example, if you want to locate a label at $2/5$ of the third side of a rectangle, the Bézier parametrization can be used to locate the point, as $[0, 1/4, 2/4, 3/4]$ are the Bézier locations of the square vertices, so $t=2/4+(2/5)*1/4$ gives the location we are looking for.

```
def cs = ["red", "blue", "green", "black", "red"] as ColorScale
def r = shape(
  type: "square",
  transform: [scale: [1.25, .75], center: true],
)
arange(0, 1, 1 / 8).collect {
  addLabelAt(r, it, cs)
}
scene.add(r)
camera.zoomToAllObjects()

def addLabelAt(MathObject obj, double t, ColorScale cs) {
  obj.label(text: "\\$t\\$", color: cs(t), scale: .375, normalized: false, t: t)
  point(at: obj.getVecAt(t),
    addToScene: true,
    style: [drawColor: cs(t),
      thickness: 50,
      layer: 1]
  )
}
```



Tip

JMathAnim implements the Groovy extension `MathObject.label(DSL block)` to easily define label parameters in DSL format. Both forms are equivalent:

DSL

```
obj.label(text: "\\$t\\$",
  color: cs(t), scale: .375,
  normalized: false, t: t)
```

GROOVY

```
obj.addLabel("\\$t\\$")
  .location(t)
  .normalized(false)
  .color(cs(t))
  .scale(.375)
```

Tip

The method `r.getVecAt(t)` returns the `Vec` position at the Bézier-parametrized `t`. If you want the length-parametrized position, use the method `r.getParametrizedVecAt(t)` or the shorter version `r(t)`, which is a Groovy extension of the `call` method.

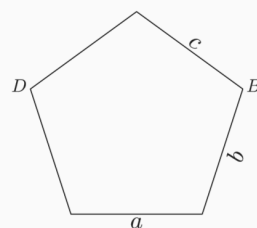
Problem: Add a label to a vertex or side

You want to add a label to a given vertex or side of a polygon or polyline.

Solution

Use the `side` or `vertex` parameter in the `label` list.

```
def r = shape(
  type: "regularpolygon",
  sides: 5,
  transform: [center: true],
  label: [
    [text: r"$a$", side: 0],
    [text: r"$b$", side: 1],
    [text: r"$c$", side: 2],
    [text: r"$B$", vertex: 2, scale: .75],
```



```

    [text: r"$D$", vertex: 4, scale: .75],
  ]
)
scene.add(r)
camera.zoomToAllObjects()

```

Problem: Modifying a label

You want to access the label of an object to modify it, for example, to change its text or scale it or animate it.

Solution

Use the `getLabel()` command.

DSL

```

def A = point(at: [0, 0], label: [text:
r"$A$"])
def B = A + [1, 0] //Copy with label "A"
def C = A + [0, 1] //Copy with label "A"
def labelB = B.label //reference object
to create the label
def labelC = C.label
labelB.setLatex(r"$B$")
labelC.setLatex(r"$C$")
apply(
  obj: [labelB, labelC],
  transform: [scale: 1.5],
  style: [color: 'chocolate4']
)
//Add objects to the scene (another way)
scene << [A, B, C]

```

GROOVY

```

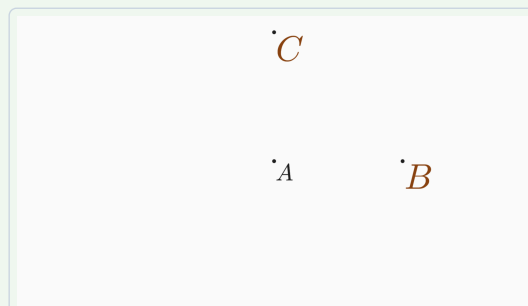
def A = Point.at(0, 0)
def labelObjA = A.addLabel(r"$A$")

//The label has to exist before the
//copies are taken
def B = A.copy().shift(1, 0)
def C = A.copy().shift(0, 1)

def labelB = B.label //reference object
to create the label
def labelC = C.label
labelB.setLatex(r"$B$")
labelC.setLatex(r"$C$")

[labelB, labelC].each
{ it.scale(1.5).color("chocolate4") }
scene.add(A, B, C)

```



Tip

Remember that in Groovy, the method `obj.getLabel()` can be written in a shorter way with `obj.label`.

Discussion

(For intermediate users) The label attached to a `MathObject` is a special container of `AbstractTippableObject`, called `RigidBody`, that basically takes a reference object

(the "A" LaTeX object for example) and places a copy of the object properly located at the label position. So, in order to make modifications to the label, you must access the reference object. The method `B.label` returns that reference object, not the container `AbstractTippableObject`, BUT the Groovy method `def labelObjA = A.addLabel(r"$A$")` does return the `AbstractTippableObject` instance. Anyway, if you need to access the `AbstractTippableObject` instance itself, you can use the command `def labelRefA = A.getInternalObject("label0")` as labels are registered as internal objects with special names.

Problem: Modifying multiple labels of an object

You want to access the labels of an object with multiple labels, like a `Shape` object for example.

Solution

Use the `getLabel` command with a parameter.

DSL

```
def sq = shape(
    type: "square",
    transform: [scale: [2, 1], center:
true],
    label: [
        [text: r"$a$",
            side: 0, rotation: "fixed"],
        [text: r"$b$",
            side: 1, rotation: "fixed"],
        [text: r"$c$",
            side: 2, rotation: "fixed"],
        [text: r"$d$",
            side: 3, rotation: "fixed"]]
)
//Oops, in a rectangle, c=a and b=d,
//they should be labelled the same!
//Let's fix that...
def l2 = sq.getLabel(2) //Label 2 ("c")
def l3 = sq.getLabel(3) //Label 3 ("d")
l2.setLatex(r"$a$")
l3.setLatex(r"$b$")

scene.add(sq)
```

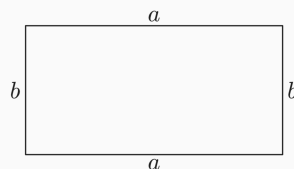
GROOVY

```
def sq = Shape.square()
    .scale(2, 1)
    .center()

['$a$', '$b$', '$c$',
'$d$'].eachWithIndex { text, i ->
    sq.addLabel(text)
        .setRotationType(RotationType.FIXED)
        .setSideIndex(i)
}

def l2 = sq.getLabel(2) //Label 2 ("c")
def l3 = sq.getLabel(3) //Label 3 ("d")
l2.setLatex('$a$')
l3.setLatex('$b$')

scene.add(sq)
```



Discussion

Every label created is registered as an *internal object* of the object it belongs to, under the key given in the spec (key: entry) or under the generated names `label0`, `label1`, `label2` ... in creation order. Being internal means the label is updated, drawn and animated together with its parent, and must never be added to the scene on its own.

`getLabel` is the way to reach a label again once it has been created. It comes in three flavors: `getLabel()` returns the first label of the object, in registration order. This is the convenient form for the common case of a single label. In Groovy `obj.getLabel()` can be shortened to `obj.label`. `getLabel(int n)` is shorthand for the generated keys: `getLabel(2)` is exactly `getLabel("label2")`, that is, the third label declared. `getLabel(String key)` returns the label registered under that key, which is the natural form when the labels were named explicitly with the `key: entry`.

All three are thin wrappers around `getInternalObject(key)`, with one important extra step: what they hand back is not the tippable itself, but the reference object the tippable was built from, typically a `LatexMathObject` (or whatever object was passed as `object:`). This distinction matters: the tippable keeps a transformed copy for drawing, rebuilt on every update from the reference; the reference is the canonical source. Changing the reference, as in `sq.getLabel(2).setLatex(r"$a$")`, is therefore permanent: the tippable declares the reference as a dependency, so the next update pass rebuilds the drawn copy with the new text. Transformations applied there, such as `getLabel("A").scale(1.5)`, compose inside the placement, so a scaled or rotated label still follows its path correctly.

When you need the tippable itself rather than its content, for instance to change the gap, the side index or the rotation type after creation, ask for the internal object directly with `getInternalObject("label0")`, or simply keep the value returned by `addLabel(...)`, which is the tippable.

Tip

Remember that `"$a$"` gives an error in Groovy because `$` is a reserved token in strings with double quotes, but perfectly legal in single-quoted strings `'$a$'`. If unsure, use the special raw form `r"$a$"` which accepts both `$` and `\` characters.

Problem: Adding congruence or parallel marks to a polygon

You have a polygon and want to mark certain sides as congruent or parallel.

Solution

Use the `label` command with the `type` parameter. This has some predefined label objects.

DSL

```
def parallelogram = shape(type:
    "square",
    transform: [
        affine: [[0, 0], [2, 0], [.5, 1]],
        center: true],
    label: [
        [type: "equal1", side: 0],
```

GROOVY

```
def parallelogram = Shape.square()
// affine: [A,B,C] maps the canonical
// frame (0,0),(1,0),(0,1) into A,B,C
def tr =
    AffineJTransform.createAffineTransformation(
        Vec.to(0, 0), Vec.to(1, 0),
```

```

        [type: "equal1", side: 2],
        [type: "equal2", side: 1],
        [type: "equal2", side: 3],
    ],
    addToScene: true
)
def t = latex(text: r"I am ♪\\a
parallelogram ♪",
    stack: [
        to: parallelogram,
        destinyAnchor: "lower",
        gaps: .1
    ],
    addToScene: true
)

```

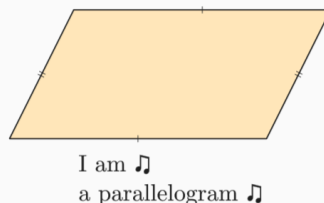
```

Vec.to(0, 1),
    Vec.to(0, 0), Vec.to(2, 0),
Vec.to(.5, 1), 1)
tr.applyTransform(parallelogram)
parallelogram.center()

// equal marks: 1 mark on sides 0 and 2,
// 2 marks on sides 1 and 3
[1, 1, 2, 2].eachWithIndex { marks, i ->
    def side = [0, 2, 1, 3][i]
    def tip =
TippableObject.equalLengthTip(parallelogram,
parallelogram.getDefaultLabelLocation(),
marks)
    tip.setSideIndex(side)
parallelogram.addInternalObject("label"
+ i, tip)
}
scene.add(parallelogram)

def t = LatexMathObject.make(r'I am ♪\\a
parallelogram ♪') // \\ is the LaTeX
line break
t.stack()
    .withGaps(.1, .1)
    .withDestinyAnchor(AnchorType.LOWER)
    .toObject(parallelogram)
scene.add(t)

```



Problem: Adding labels as independent objects

For example, you have a `Shape` object in the scene and want to animate the creation of labels. You cannot do this if labels are defined as internal objects.

Solution

Use the `label(..)` DSL block. The main difference between using something like `def l=label(...)` and defining the label inside a `shape(..)` block, for example, is that it is not created as an internal object, so you can add it or remove it from the scene and animate it separately.

```

def r = shape(
    type: "regularpolygon",
    sides: 5,

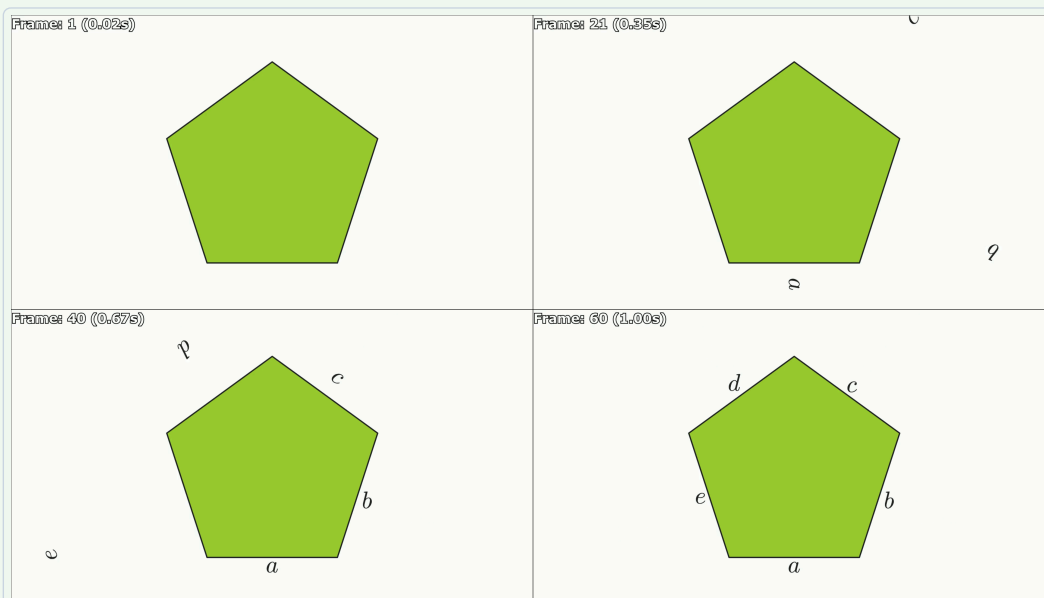
```

```

    transform: [center: true],
    style: [fillColor: 'yellowgreen'],
    addToScene: true
)
//Single quotes: "$a$" is a broken GString
def texts = ['$a$', '$b$', '$c$', '$d$', '$e$']
def labels = texts.withIndex().collect { txt, i ->
    label(text: txt, path: r, side: i, rotation: "fixed")
}

appear(
    runtime: 1,
    type: "movein",
    obj: labels,
    delay: .5,
    effects: [turns: 1]
)

```



Problem: Adding length labels to delimiters

You want to add a label to a delimiter that automatically shows the length.

Solution

Use the `delimiter` DSL block with the parameter `lengthLabel`.

DSL

```

def sq = shape(
    type: "centeredsquare",
    transform: [scale: [1.5, 1]],
    style: [drawColor: 'darkgreen']
)
def del1 = delimiter(
    stackedTo: sq, //Stack to this object
    anchor: "lower", //stacked lower

```

GROOVY

```

def sq = Shape.square()
    .scale(1.5, 1)
    .center()
    .drawColor("darkgreen")

def del1 = Delimiter.makeStacked(
    sq, //Stack to this object
    AnchorType.LOWER, //stacked lower
    DelimiterType.LENGTH_ARROW, //Type

```

```

    type: "length_arrow",
    gap: .1,
    lengthLabel: "0.00",
)
def del2 = delimiter(
    stackedTo: sq,
    anchor: "right",
    type: "length_arrow",
    gap: .1,
    lengthLabel: [
        format: "0.0",
        color: 'darkgreen',
        scale: .7]
)
scene.add(sq, del1, del2)

```

```

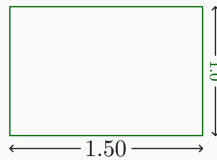
    .1 //Gap object-delimiter
)
del1.addLengthLabelTip("0.00")

def del2 = Delimiter.makeStacked(
    sq, AnchorType.RIGHT,
    DelimiterType.LENGTH_ARROW, .1)

del2.addLengthLabelTip("0.0")
del2.getLabelTip()
    .scale(.7)
    .color("darkgreen")

scene.add(sq, del1, del2)

```



Problem: Point at a detail with a callout

You want a note sitting in a free corner of the screen with a line running from it to the object it talks about, and the line to stay attached when either of the two moves.

Solution

A `connector` is a leader line whose two ends are objects rather than points: it leaves the note at its border, arrives at the border of the target, and recomputes itself whenever one of them moves.

```

def target = shape(type: "circle",
    transform: [scale: .35, shift: [1.4, -.6]],
    style: [fillColor: "steelblue", fillAlpha: .5, thickness: 4],
    addToScene: true)

def note = latex(text: r"the radius",
    transform: [scale: .8],
    style: [color: "gray20"],
    stack: [screen: "upper_left", gaps: .4],
    addToScene: true)

def leader = connector(from: note, to: target,
    curvature: .3,
    style: [drawColor: "gray50", thickness: 3],
    addToScene: true)

```

Nothing else is needed to keep the three together: moving the target drags the leader with it, and so does moving the note.

```
animShift(obj: target, dx: -1, dy: 1, runtime: 2)
```

Discussion

A **connector** and an **arrow** look alike and are not the same thing: an arrow joins two **points**, a connector joins two **objects** and keeps a gap against their outlines, so the head never lands on top of the thing it points at.

Tip

type: "stepped" replaces the straight body with horizontal and vertical runs, the look of a technical drawing. **curvature** is meaningless there, and is ignored with a warning.

Tip

For a note that has to stay glued to a moving object instead of to a corner of the screen, no connector is involved: `updater(type: "stackTo", obj: note, to: target, destinyAnchor: "upper", gaps: .1)` keeps it at a fixed distance from the target for the rest of the scene.

Setting Up Your Scenes

Problem: Setting up a proper workspace

You want to organize your scripts, resources, etc. before you get lost.

Solution

Use the predefined directory structure. By default JMathAnim uses the following project structure:

root_dir/	Your scripts live here or in a subdirectory
├ media	All videos and images generated go here
├ resources	In this directory you can find all (well, most of) your resources:
│ ├── config	Config files to be parsed. Colors, styles, scene settings, etc.
│ ├── images	All images you may use, both bitmap or svg format.
│ ├── geogebra	If you plan to import Geogebra files, that's your place!
│ └ plugins	For storing jar files with JMathAnim plugins

When you try to import an image, for example

```
image(
  file: "image.png",
  transform: [height: 2, center: true]
)
```

JMathAnim will try to load the file from the `images` directory, giving a non-stopping error if the file does not exist.

If you need to access a file in another location, just use the `'!'` modifier and give it an absolute path:

```
image(
  file: r"!c:\media\topsecret\image.png",
  transform: [height: 2, center: true]
)
```

Tip

JMathAnim has project management. It is basically a file `.jproj` that stores information about one specific session: current directory, opened files, etc.

Problem: Using resources from other directory

You want to write and execute scripts in one directory but want to reuse the `resources` directory in another place.

Solution

Use the `config` command. The `config` object lets you configure almost any aspect of the behavior of the scene executor, including directories.

DSL

```
config(  
    resourcesDir: "/home/tao/  
myResources"  
)
```

GROOVY

```
config.setResourcesDir(  
    "/home/tao/myResources")
```

Tip

It is recommended to use the raw form `r`, especially on Windows: the path separator there is the backslash `\`, which Groovy reads as the start of an escape sequence inside an ordinary string.

```
config(  
    resourcesDir: r"c:\resources" //Everything Ok!  
)  
  
config(  
    //\r is read as a carriage return:  
    //the path becomes "c:esources"  
    resourcesDir: "c:\resources"  
)
```

Problem: Use a secondary resources location

You want to use your own default resources directory but you also have some interesting things in another one and want JMathAnim to look on this one also.

Solution

Use the `secondaryResourcesDir` parameter. If this value is set and JMathAnim cannot find a file in the primary resources directory, it will try in this one. In this example (assuming Windows):

```
config(  
    secondaryResourcesDir: r"c:\resources"  
)  
def img = image(file: "image.png",  
    transform: [height: 2, center: true])
```

First JMathAnim will try to look in your `<root_dir>\resources\images` directory. If not found, it will try in the secondary resources dir `c:\resources\images`. If not found, it will emit a non-stopping error.

Tip

The `secondaryResourcesDir` is an absolute path. It doesn't use the `"!"` or `"#"` modifiers.

Problem: Reuse a common setup across several scenes

Several scenes of your video share the same axes, background and logo. You want to build them once in a helper method and call it from each scene.

Solution

A script is a plain Groovy script, so you can declare methods in it. Any method declared in the script body inherits the whole DSL, so it can call `axes(...)`, `text(...)` or `config` exactly like the main body does. Put the shared stage in one of these methods and let it return the pieces the scene will need:

```
// Common stage: dark background, axes and a corner logo
def commonStage(boolean animate = false) {
    config(load: "#dark.xml")//Any config option here...
    def ax = axes(
        xRange: [-3, 3],
        yRange: [-2, 2],
        xStep: .5, yStep: .5,
        addToScene: false)

    def logo = latex(text: r"\tt #jmathanim",
        style: [color: "gray"],
        stack: [screen: "lower_right", gaps: 0.1],
        addToScene: true)

    if (animate) {
        appear(obj: ax, type: "draw", runtime: 1)
    }
    //Returns a dictionary with all elements
    return [axes: ax, logo: logo]
}

def stage = commonStage(true)
```

To share it with the other scenes, move the method to its own file, say `common.groovy`, in the same directory as your scripts, and pull it in from every scene with the `import(...)` directive:

```
import("common.groovy")

def stage = commonStage(false)//Don't animate
appear(obj: stage.axes, type: "fadein", runtime: 1)
// ...the rest of this particular scene
```

Problem: Reusing personalized palettes

You have spent hours carefully choosing your color palette and want to reuse it in your animations.

Solution

Use named styles and store them. There are several ways to do that, just choose the one you prefer:

Method 1 Use xml config files

Just use config files with the style definitions you want in an XML file like this:

```
<JMathAnimConfig>
<style name="myBeautifulStyle">
  <drawColor>violet</drawColor>
  <fillColor>white</fillColor>
```

```
<fillAlpha>0.5</fillAlpha>
<stroke thickness="30"/>
</style>
</JMathAnimConfig>
```

put this in a file in the `resources/config/` folder of your root project, for example `myStyles.xml`, and with a code like this you will have the style available:

DSL

```
config(load:"myStyles.xml")
def c=shape(
    type: "circle",
    style: "myBeautifulStyle")
```

GROOVY

```
config.parseFile("myStyles.xml")
def c=Shape.circle()
    .style("myBeautifulStyle")
```

Tip

For XML users: In the source code of the project there is the file `/jmathanim-core/src/main/resources/config/JMathAnimConfig.dtd` defining the structure of the JMathAnimConfig XML file.

Tip

The XML definition of a `LatexStyle` can be generated through the LaTeX style editor in the GUI. Just create it and select "copy XML" to copy the definitions to the clipboard, so you can paste it into the `.xml` file you choose.

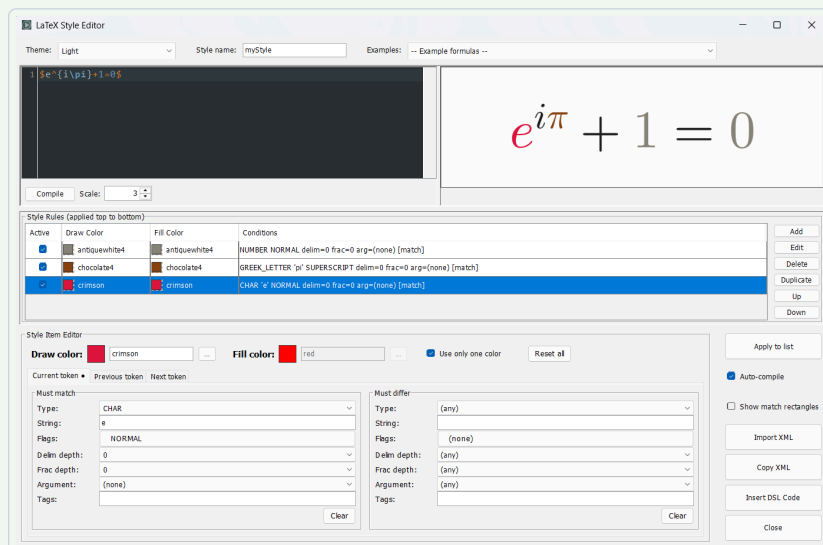


Figure 4: LaTeX style editor

Method 2 Use code definitions and import

Define your styles using DSL blocks or Groovy code like this:

DSL

```
createStyle(
    name: "myRedStyle",
    //Copy base values from this style
    from: "default",
    style: [
```

GROOVY

```
def mp = config.styles["default"].copy()

mp.drawColor("red")
mp.setThickness(50d)
mp.setDashStyle(DashStyle.DASHED)
```

```
drawColor: "red",
thickness: 50,
dashStyle: "dashed"
]
)
def c=shape(
type: "circle",
style: "myRedStyle")
```

```
config.createStyleFrom(mp, "myRedStyle")
def c=Shape.circle()
.style("myRedStyle")
```

Save it in a `.groovy` file in your project and you can use the `import(...)` command to load the definitions. This is a good approach if you have a file like `common.groovy` that you import in every animation and don't want to fight with the `xml` format.

Animating Objects

Note

In construction...

Output and Input

Problem: Generate images

You want to use JMathAnim to generate images.

Solution

Just create the objects you want to include in the image and use the `scene.saveImage(filename)` command.

```
def c = shape(type: "circle", addToScene: true)

//By default, save it in media/mycircle.png
scene.saveImage("mycircle.png")
```

You can specify `png`, `tif` or `gif` (static) formats. The default format if extension is not defined is `png`.

Tip

You can also pass the `svg` extension and it will be saved in this format. For extra options you can use

```
scene.saveSVGImage(String filename, boolean useMathView, boolean addBackground)
```

where the flag `useMathView` determines whether the bounding box of the SVG file created matches the current math view or only the exported objects. The `addBackground` flag, if set, exports a rectangle filled with the background color. The default for `scene.saveImage("file.svg")` is `useMathView = true` and `addBackground = false`.

Tip

If no frames were generated, or the scene has changed since the last one, a `scene.advanceFrame()` is automatically called.

Compound Objects

Most recipes so far build one object at a time. This chapter is about the other direction: assembling several objects into a new one that the rest of the scene treats as a single thing, with pieces you can still reach by name.

The tool for that is the `compound` command. It is worth learning properly, because a well-built compound is the difference between a diagram you can animate and restyle in one line and a pile of loose variables you have to keep in step by hand.

Problem: Decide between a group and a compound

You have several objects that belong together and you don't know whether to use `group` or `compound`.

Solution

The difference is where the pieces live.

A `group` is a *logical* container. Its members are citizens of the scene in their own right: adding the group to the scene adds them one by one, and the group itself draws nothing of its own. It exists so you can operate on all of them at once, and an object can belong to as many groups as you like. (The one exception: a group carrying labels of its own does enter the scene, since it is the one that draws them.)

A `compound` is the scene object. It is added, drawn, layered, removed and animated as one thing, and its pieces are internal to it: adding the compound does not add them, the compound paints them itself, and removing the compound removes them with it.

That last part is a convention rather than a lock. Nothing stops you from adding a piece to the scene on its own, from animating one directly (which adds it, as any animation does with whatever it animates), or from handing the same object to a second compound. What you get is a piece with two owners, and the usual outcome is one that outlives the removal of its compound or that two different things try to place. Reach into a compound to read and to style; let the compound itself be what you add, move and animate.

GROUP

```
//The scene ends up with 2 objects
def g = group(
  obj: [shape(type: "square"),
        shape(type: "circle")],
  layout: "right", gap: .1,
  addToScene: true
)
```

COMPOUND

```
//The scene ends up with 1 object
def c = compound(
  obj: [shape(type: "square"),
        shape(type: "circle")],
  layout: "right", gap: .1,
  addToScene: true
)
```

Use `group` to operate in bulk on objects that are their own citizens, and `compound` to build a new object out of parts: a balance out of its base, its beam and its two pans, a chessboard out of its squares.

Note

The `layer` of a compound places it among the objects of the scene, while the layers of its pieces only order them among themselves. To draw one piece over another, set the layer on the piece after creating the compound.

Problem: Build an object out of named pieces

You want to assemble a drawing from several shapes and be able to refer to each of them afterwards.

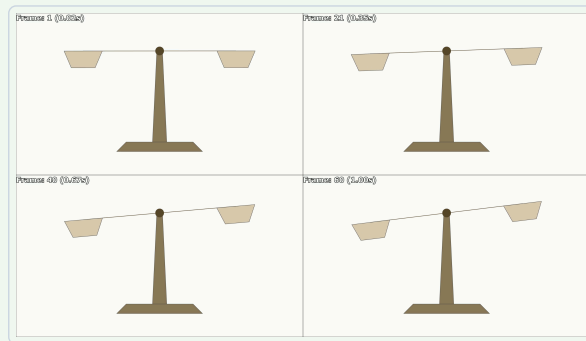
Solution

Pass a `Map` to `obj`: . The keys become the names of the pieces, and the writing order is also the drawing order.

```
def balance = compound(
  // Every piece is written under a name.
  // Writing order is drawing order
  obj: [base : shape(type: "polygon",
    points: [[-.9,-1.2],[.9,-1.2],[.7,-1],[-.7,-1]]),
    stand: shape(type: "polygon",
    points: [[-.15,-1],[.15,-1],[.06,.9],[-.06,.9]]),
    beam : shape(type: "segment",
    from: [-1.6, .9], to: [1.6, .9]),
    panL : shape(type: "polygon",
    points: [[-2,.9],[-1.2,.9],[-1.35,.55],[-1.85,.55]]),
    panR : shape(type: "polygon",
    points: [[1.2,.9],[2,.9],[1.85,.55],[1.35,.55]]),
    pivot: shape(type: "circle",
    transform: [scale: .09, shift: [0, .9]]]),

  // The common base every piece starts from
  style: [drawColor: "#3d3d3d", thickness: 3, fillColor: "#c8b998"],

  // ...and what each piece corrects, by its name
  parts: [base : [fillColor: "#8a7a5c"],
    stand : [fillColor: "#8a7a5c"],
    beam : [thickness: 6, drawColor: "#5a4a2c"],
    // a glob hits both pans
    "pan*": [thickness: 4, fillColor: "#d9cbb0"],
    pivot : [fillColor: "#5a4a2c", thickness: 1, layer: 1]]
)
scene.add(balance)
camera.scale(1.5)
// A name selection is animated as one thing: beam and both pans turn together
animRotate(obj: balance["pan*", "beam"], angle: 7 * DEGREES,
  center: balance.pivot.center, runtime: 1)
```



The same object without the DSL. `CompoundMathObject.make` takes the very same `Map`, `style:` is the compound's own drawing properties (which reach every piece), and `parts:` is `get(name)` followed by the ordinary style setters. The one thing that does not survive the translation is the glob: plain Groovy has no `"pan*"`, so the two pans are listed.

GROOVY

```
def balance = CompoundMathObject.make([
    base : Shape.polygon(Vec.to(-.9,-1.2), Vec.to(.9,-1.2),
                        Vec.to(.7,-1), Vec.to(-.7,-1)),
    stand: Shape.polygon(Vec.to(-.15,-1), Vec.to(.15,-1),
                        Vec.to(.06,.9), Vec.to(-.06,.9)),
    beam : Shape.segment(Vec.to(-1.6, .9), Vec.to(1.6, .9)),
    panL : Shape.polygon(Vec.to(-2,.9), Vec.to(-1.2,.9),
                        Vec.to(-1.35,.55), Vec.to(-1.85,.55)),
    panR : Shape.polygon(Vec.to(1.2,.9), Vec.to(2,.9),
                        Vec.to(1.85,.55), Vec.to(1.35,.55)),
    pivot: Shape.circle().scale(.09).shift(0, .9)
])

// style: the compound styles itself, and that reaches every piece
balance.drawColor("#3d3d3d").thickness(3).fillColor("#c8b998")

// parts: get(name) plus the same setters, one piece at a time
balance.get("base").fillColor("#8a7a5c")
balance.get("stand").fillColor("#8a7a5c")
balance.get("beam").thickness(6).drawColor("#5a4a2c")
["panL", "panR"].each { balance.get(it).thickness(4).fillColor("#d9cbb0") }
balance.get("pivot").fillColor("#5a4a2c").thickness(1).layer(1)

scene.add(balance)
camera.scale(1.5)

play.rotate(1, balance.get("pivot").center, 7 * DEGREES,
            balance.get("panL"), balance.get("panR"), balance.get("beam"))
```

Discussion

Two parameters are doing different jobs here, and the order in which they apply is what makes them compose.

`style:` is the common base: it is written to every piece. `parts:` then corrects individual pieces by name, and it is applied *after* `style:` on purpose, because the whole point of a correction is to win over the base. Writing them the other way round would leave the corrections buried.

A piece can also be given its style in its own `shape(...)` call, as usual. The reason to hoist it into `style:` and `parts:` is that both stay readable when the pieces are many: you see the

family resemblance in one place and the exceptions in another, instead of hunting through six constructor calls.

Tip

Pieces given without names, as a plain list, are still perfectly usable: they are then reached by index, `c[0]`, `c[1]`. Names pay off as soon as there are more than three or four of them.

Problem: Reach a piece of a compound after creating it

You want to restyle or move one of the pieces of a compound that already exists.

Solution

Use the subscript with the name of the piece.

```
balance["pivot"].drawColor("#b5544a") //one piece
balance.part("pivot")                  //the same, for scripts preferring a call
```

Several names at once give a `MathObjectGroup`, so the whole selection can be operated on in one go:

```
balance["panL", "panR"].thickness(5) //two pieces
balance["panL,panR"].thickness(5)    //the same, as one String
balance["pan*"].thickness(5)         //the same, as a glob
```

The result is a *logical* group: the pieces stay inside the compound, which is still the one drawing them. Use it for bulk operations, not to add them to the scene.

The same names work from `apply(...)`, which is the way to restyle by name without reaching for the pieces yourself:

```
apply(obj: balance, parts: [panR: [fillColor: "#b5544a"]])
```

Tip

A name may be a path: `myAxes["xAxis/ticks"]` descends into the parts of a part. And `"*"` selects every piece at once.

Problem: Place a piece against another piece

One piece of the compound has to sit next to a sibling, but while you are writing the `obj:` map the siblings do not exist yet.

Solution

Use the `init:` parameter, a closure run once on the finished object.

DSL

```
def house = compound(
  obj: [
    walls: shape(type: "square",
```

GROOVY

```
def walls = Shape.square()
    .fillColor("yellow3")
def roof = Shape.regularPolygon(3)
```

```

    style: [fillColor: 'yellow3']],
    roof: shape(type: "regularpolygon",
        sides: 3,
        style: [fillColor: 'firebrick'])
],
init: { c -> c["roof"]
    .stack(to: c["walls"],
        destinyAnchor: "upper") }
)
scene.add(house.center())

```

```

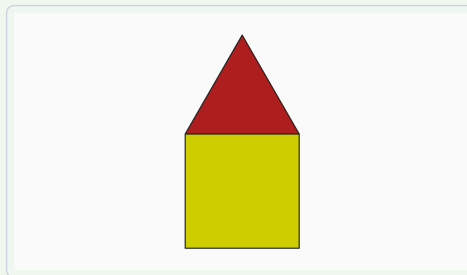
    .fillColor("firebrick")

def house = CompoundMathObject.make(
    [walls: walls, roof: roof])

//No init: outside a block the
//statements simply come after
house.get("roof").stack()
    .withDestinyAnchor(AnchorType.UPPER)
    .toObject(house.get("walls"))

scene.add(house.center())

```



Discussion

The values of the `obj:` map are evaluated *before* the compound exists, so a piece written there cannot refer to a sibling: at that moment there is no container and no names, only expressions being computed. Inside `init:` they are all there, under the names you gave them. That is also why `init:` has no counterpart in the plain API: there the object is built by a statement rather than inside one expression, so placing a piece against a sibling is just the next line. `init:` buys back, for the DSL, the moment that plain Groovy has for free.

`init:` runs last, after the layout, the transform, the style and the parts, so nothing the block does afterwards can undo its work. That ordering is the reason it can place a piece against a sibling at all: by then both of them are where the layout and the transform left them.

Note

The full order in which a `compound` block applies its parameters is: `homogenize`, then `layout`, then the common properties (`style`, `transform`, `stack`), then `parts`, and `init` last. Whenever two parameters seem to fight, this is the order that decides.

Problem: Create a chessboard

You want to build an 8×8 chessboard with two alternating colors, and you want to be able to refer later to each square by its usual name (`board.a1`, `board["e4"]`) or (column, row) numbers (`board.at(1,2)`).

Solution

Build a `Map` whose keys are the square names and whose values are the squares themselves, then hand that map to the `compound` command together with a `box` layout.

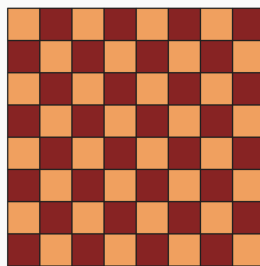
```

def cols = 'a'..'h'
def rows = 1..8
def squareColors = ["sandybrown", "brown4"] // light, dark

// Row-major: the order a box layout with direction "right_up" consumes
def squares = [:]
rows.each { r ->
    cols.eachWithIndex { col, c ->
        squares[col + r] = shape(type: "square",
                                style: [fillColor: squareColors[(c + r) % 2]])
    }
}

def board = compound(
    obj: squares,
    layout: [type: "box", size: 8, direction: "right_up"],
    transform: [height: 2, center: true],
    methods: [at: { c, r ->
        (c in 1..8 && r in 1..8) ? delegate[('a'..'h')[c - 1] + r] : null }]
)
scene.add(board)

```



Discussion

This is a good example of combining plain Groovy syntax with DSL blocks. The recipe has three separate stages, and keeping them separate is what makes the result easy to work with afterwards: first **name and style** every piece, then **arrange** them, and only then **place and size** the whole thing.

First, name the pieces you are going to use:

```

def cols = 'a'..'h'
def rows = 1..8
def squares = [:]

```

In Groovy, `'a'..'h'` and `1..8` are ranges, so they can be iterated directly. The literal `[:]` is an empty map, and a Groovy map is a `LinkedHashMap`: it **remembers the order in which keys were inserted**. That single fact is what allows the same object to play two roles at once, a dictionary (`"e4"` maps into a `Shape` square) and an ordered list (the sequence the layout will consume). Building the map with a straightforward loop instead of writing 64 `shape(...)` calls is the whole point of dropping to plain Groovy here: the DSL is used for **what a square is**, Groovy for **how many of them there are**.

Second, generate the squares. Before creating the `compound`, we build a map of the form `name: square`, taking a string like `"a1"` into the `Shape` object representing that square. We accomplish this with the following nested loop:

```
rows.each { r ->
  cols.eachWithIndex { col, c ->
    squares[col + r] = shape(type: "square",
                             style: [fillColor: squareColors[(c + r) % 2]])
  }
}
```

We iterate over all the elements of `rows` and, for each one, over all the elements of `cols`. This way the first row is completed first, then the second one, and so on. Note that we don't worry about the size of the "square" shapes: we will deal with it later. Note that in this loop, `col` is a string and `r` an integer, so for example if `col="d"` and `r=3`, then `col+r` gives "d3". Let's look at this odd-looking line, the one that produces the chessboard coloring:

```
style: [fillColor: squareColors[(c + r) % 2]]
```

`eachWithIndex` gives both the column letter `col` and its position `c` (0 for `a`, 7 for `h`), while `r` is the rank number itself (1 to 8). The parity of `c + r` alternates along a row and along a column at the same time, which is exactly the chessboard pattern.

The `%` operator, in case you don't know, is the *remainder* operator, and here it simply gives the remainder of the Euclidean division by 2. The result is that an expression like `n % 2` returns 0 if `n` is even and 1 if `n` is odd (an old programming trick, using mathematics that has been around for millennia).

Note that the parities work out to the real convention: `a1` gives $(0 + 1) \% 2 = 1$, the dark color, and `h1` gives $(7 + 1) \% 2 = 0$, the light one, so every player has a light square on their right-hand corner.

Third, create a proper JMathAnim container for all these squares. A `compound` object is ideal:

```
def board = compound(
  obj: squares,
  layout: [type: "box", size: 8, direction: "right_up"],
  transform: [height: 2, center: true],
  methods: [at: { c, r ->
    (c in 1..8 && r in 1..8) ? delegate(['a'..'h'])[c - 1] + r : null }]]
)
```

Don't be afraid of the `methods` parameter: it is optional, and we discuss it later.

The `layout` command positions the squares in the right place:

```
layout: [type: "box", size: 8, direction: "right_up"]
```

A `box` layout consumes the elements in order and places them one after another: `size: 8` closes a row after 8 elements, and `direction: "right_up"` means "fill each row from left to right, and put the next row above the previous one". No gaps are given, so the squares end up edge to edge.

This is the step that has to be kept in sync with the loop. The outer loop runs over ranks and the inner one over files, so the insertion order is `a1, b1, ..., h1, a2, ...`: rank 1 is inserted first and `right_up` stacks the later rows above it, so rank 1 lands at the bottom of the screen, where a chessboard seen from White's side has it.

Swapping the two loops, or asking for `direction: "right_down"`, would still produce a perfectly valid 8×8 board, but the names would no longer match the positions, which is the kind of bug that is invisible until the first piece is placed.

This line does the JMathAnim magic:

```
transform: [height: 2, center: true]
```

Transforms in a DSL block are applied in the order they are written, and here they are applied to the compound after the layout has run. So the board is resized as a single unit: `height: 2` scales it until its bounding box is 2 math units tall (the squares keep their relative sizes and stay glued together), and `center: true` moves that bounding box to the center of the screen. This is why the individual squares were created at their default size and never scaled or moved by hand: the layout takes care of the relative positions and the compound transform of the absolute ones.

The methods parameter

With the `compound` object, it is easy to refer to a square using its name:

```
//1.e4, the most popular opening move in chess
shape(
  type: "circle",
  style: [fillColor: 'ivory'],
  transform: [height: board.e4.height * .8],
  stack: [to: board.e4],
  addToScene: true
)
```

As you have noticed, though, the key is a string like `"a1"` or `"b5"`, which is human readable, but not "programming friendly" (for example, it is not easy to implement a loop over squares with this format). A more convenient indexing would be a pair `[col,row]` like `[1,2]`, equivalent to `"a2"` (columns and rows counted from 1 to 8).

Sure, you can define a method that converts a pair like `[3,4]` into the key `"c4"` and use it to access the squares, but the created object then depends on a separate method you have to define and "carry around". The `methods` parameter is here to save the day:

```
methods: [at: { c, r ->
  (c in 1..8 && r in 1..8) ? delegate[('a'..'h')[c - 1] + r] : null }],
```

This parameter takes a map of closures that become methods of the object itself. Here we define the `at` method, which takes 2 parameters `c`, `r`, turns them into the human-readable key `"a1"`, `"b3"`, etc. and returns the corresponding square. If numbers are not valid (not between 1 and 8) it returns `null`.

The key is the `delegate` keyword. Inside a closure declared this way, `delegate` is the object the method belongs to, that is, the board. So `delegate[...]` is the very same access as `board[...]` above, written from inside the object, where its name is not known yet. For example, `board.at(2,1)` first computes the key `"b1"` and then returns `board["b1"]`.

Note that the closure returns `null` instead of failing when the pair falls outside the board, which spares a lot of bounds checking in the code that uses it: a knight sitting on the edge is asked for squares that do not exist all the time.

So we can now access a square of the board in two ways:

```
def whiteKingSq = board.e1
def blackQueenSq = board.at(4, 8) //Col 4, row 8
```

The beauty of this is that the created object is self-contained. You can implement the methods you want and they will live inside the object (well, that's the whole point of Object Oriented Programming...).

Note

The parameters of the closure become the parameters of the method, with one Groovy quirk worth knowing: a closure written without an explicit parameter list still takes one implicit `it`. So `methods: [total: { delegate.size() }]` declares a method that accepts zero or one argument: `board.total()` works, and so does `board.total(99)`, silently ignoring the argument. Write `{ -> ... }` when the method should take none and reject anything else.

The quirk only bites when the body actually uses `it`. Called with no argument, `it` is `null`, so the failure comes from inside the body (a `NullPointerException`, say) instead of a clear complaint about the number of arguments.

Tip

The same three-stage pattern (build a keyed map, lay it out, transform the whole) scales to any structured diagram: a calendar month, a keyboard, a Sudoku grid. The only thing to get right is that the insertion order of the map matches the direction in which the layout consumes its elements.

Note

The equivalent Groovy code without DSL is something like this:

```
def cols = 'a'..'h'
def rows = 1..8
def squareColors = ["sandybrown", "brown4"] // light, dark

def board = CompoundMathObject.make()

// Row-major: the order a box layout with direction "right_up" consumes
rows.each { r ->
    cols.eachWithIndex { col, c ->
        def sq = Shape.square()
        sq.fillColor(squareColors[(c + r) % 2])
        board.addWithKey(col + r, sq)
    }
}

board.setLayout(BoxLayout.make(8)) // direction RIGHT_UP is the default
    .setHeight(2)

// methods: [at: {...}] installs the closure in the per-instance metaclass
board.metaClass.at = { c, r ->
    (c in 1..8 && r in 1..8) ? delegate[('a'..'h')[c - 1] + r] : null
}

scene.add(board.center())
```

Problem: Animate the geometry of a compound with a single value

You have a composite object with a moving part, a lid that opens or a needle that turns, and you want to animate it without touching the pieces one by one.

Solution

Declare the values the geometry depends on with `state:`, and a `pose:` closure that places the pieces from them.

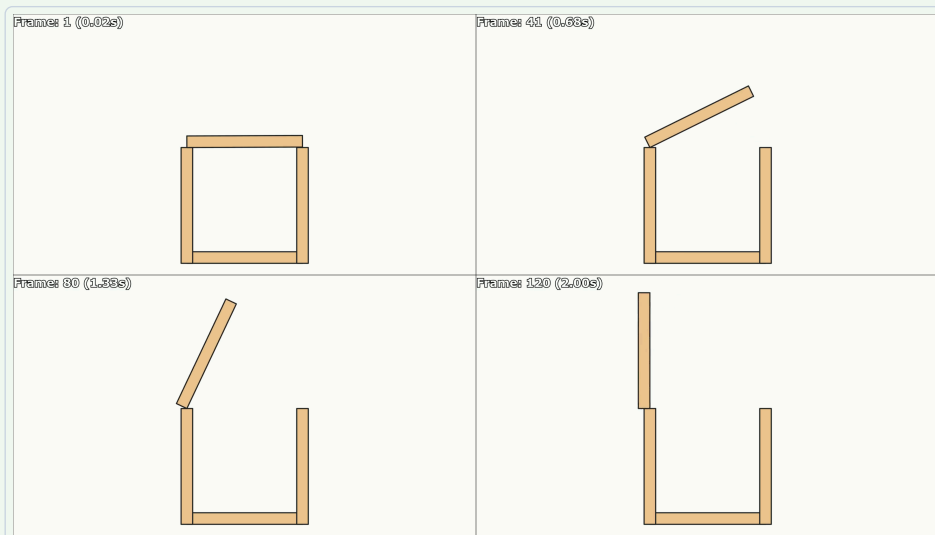
```
def botSide = shape(type: "square",
  transform: [center: true, scale: [1, .1], shift: [0, -.5]],
  style: [fillColor: 'burlywood2'])

def leftSide = botSide.copy()
  .transform(rotate: [[0, 0], -PI / 2], align: [botSide, "lower"])
def rightSide = leftSide + [1, 0]
def lid = botSide + [0, 1]

def box = compound(
  obj: [bottom: botSide, leftSide: leftSide,
    rightSide: rightSide, lid: lid],
  state: [aperture: 0],
  pose: { b -> b["lid"].rotate(b["leftSide"].upper,
    90 * DEGREES * b.state("aperture")) }
)

box.stack(screen: "lower", gaps: .1)
scene.add(box)

animState(obj: box, to: [aperture: 1], runtime: 2)
```



Discussion

`state:` declares named values with their initial value, and `pose:` says what the geometry looks like for a given state. The closure runs once at the start and then only when a value has changed, so the compound rebuilds itself exactly as often as it needs to.

Read `pose`: as a drawing rather than as a movement. Before it runs, every piece is put back to the position it was built in, the lid closed, so the angle you write is the angle the lid ends up at. Nothing accumulates: setting the value backwards, interrupting the animation halfway or animating with `lambda: "backAndForth"` all land exactly where the value says.

And note the subscript. `b.leftSide` would work too, because a compound answers `get(String)` and Groovy uses that as a catch-all property getter, but only as a fallback: a real property of the object is found first. That is why the piece is not called `left`, since `b.left` is the left anchor of any object's bounding box, a usable point, so nothing would fail and the arm would simply hinge in the wrong place. The subscript has no fallback. The builder warns when a piece name collides with a property.

`rebuild`: is the same closure written in differences, for when putting every piece back on each frame is too much. There the pieces stay where the previous run left them and `b.delta("aperture")` is what moves them, at the price of a closure that has to be an exact inverse of itself. Only one of the two may be written. In this example, defining the behavior of the lid with `rebuild` instead of `pose` reads:

```
rebuild: { b -> b["lid"]
            .rotate(
                b["leftSide"].upper, 90 * DEGREES * b.delta("aperture"))
        }
```

What that costs is worth knowing. Neither form costs anything on a frame where no value moved, so a compound sitting still is free either way. On the frames where one does move, `pose` copies the geometry of every piece back before placing them: for the four rectangles here that is nothing, but it grows with the amount of geometry, and a compound of formulas or function graphs pays it on every frame of the animation. `rebuild` costs only what the closure does.

There is a reason `pose` reads so naturally, and it is not a coincidence: it is the same deal JMathAnim's own animations make. `animRotate` and its siblings save the state of what they animate and restore it before applying the transform on every single frame. `pose` does exactly that with the pieces. Start with `pose`, and move to `rebuild` only when the compound is big enough for the restore to show.

Once declared, the state is the only handle you need: `apply(obj: box, state: [aperture: 1])` jumps to a position and `animState(...)` animates the way there.

Without the DSL

This is the one compound feature whose plain form is a class rather than a call: `state`: and `pose`: are shorthand for subclassing `StatefulCompound`. The subclass declares its values in the constructor and implements two methods, `createInstance()` (used when the object is copied) and `rebuild()`, which is the closure under another name.

```
class Box extends StatefulCompound<Box, Shape> {

    Box() {
        declareState("aperture", 0)    //0 closed, 1 open
        //The deal that pose: makes. Leave it out and rebuild()
        //works on differences, as rebuild: does
        setAbsoluteRebuild(true)
    }
}
```

```

    }

    protected Box createInstance() { new Box() }

    protected void rebuild() {
        //Absolute rebuild: the lid is closed again before this runs,
        //so this is where it ends up, not how far it turns
        get("lid").rotate(get("leftSide").upper,
            90 * JMathAnimScene.DEGREES * state("aperture"))
    }
}

def box = new Box()
box.addWithKey("bottom", botSide)
box.addWithKey("leftSide", leftSide)
box.addWithKey("rightSide", rightSide)
box.addWithKey("lid", lid)

```

Tip

DEGREES and **PI** are bindings of the script, so they are not in scope inside a class declared in it. Write `JMathAnimScene.DEGREES` and `Math.PI` there.

Constructible Objects

Every object in the previous chapters is moved by telling it to move. A constructible object is not: it is *described* instead of placed, and it recomputes itself whenever anything it was described in terms of changes. The circle through three points is not a circle you position, it is the circle those three points determine, and moving one of them is the only way to change it.

The whole family lives behind the `ct` commands, and the pattern is always the same: free points at the bottom, everything else described on top of them, and a single `shift` on a free point at the end that the entire construction follows.

Problem: Draw the incircle of a triangle

You want the circle inscribed in a triangle. There is no command for it, and although its center is easy enough, two angle bisectors meeting, you have no number to give as its radius.

Solution

The piece you are missing is a point of tangency. The incircle touches each side at the foot of the perpendicular dropped from the incenter, and that foot is exactly what `ctPoint(type: "closest", ...)` returns: the point of an object nearest to a given one. With a touch point in hand, `ctCircle(center:, through:)` closes the construction and no distance is ever computed.

DSL

```
def A = ctPoint(at: [-1.5, -1])
def B = ctPoint(at: [2, -1])
def C = ctPoint(at: [.5, 1.5])
def tri = ctPolygon(points: [A, B, C],
  label: [
    [vertex: 0, text: r"$A$"],
    [vertex: 1, text: r"$B$"],
    [vertex: 2, text: r"$C$"]
  ]
)

//Incenter: two angle bisectors,
//vertex is the middle parameter
def I = ctIntersectionPoint(
  a: ctAngleBisector(A: B, B: A, C: C),
  b: ctAngleBisector(A: A, B: B, C: C)
)

//Radius: the touch point is the foot of
//the perpendicular from I to a side
def T = ctPoint(type: "closest",
  on: ctSegment(A: A, B: B), point: I)
def incircle = ctCircle(
  center: I, through: T)

camera(adjustToObjects: [A, B, C],
  gaps: 0.4)
```

GROOVY

```
def A = CTPoint.at(-1.5, -1)
def B = CTPoint.at(2, -1)
def C = CTPoint.at(.5, 1.5)
def tri = CTPolygon.make(A, B, C)

//Vertex labels, attached to the polygon
["A", "B", "C"].eachWithIndex { txt, i -
>
  def tip = LabelTip.make(tri,
    tri.getDefaultLabelLocation(),
    '$' + txt + '$')
  tip.setVertexIndex(i)
  tri.addInternalObject("label" + i,
    tip)
}

//Incenter: two angle bisectors,
//vertex is the middle parameter
def I = CTIntersectionPoint.make(
  CTAngleBisector.make(B, A, C),
  CTAngleBisector.make(A, B, C))

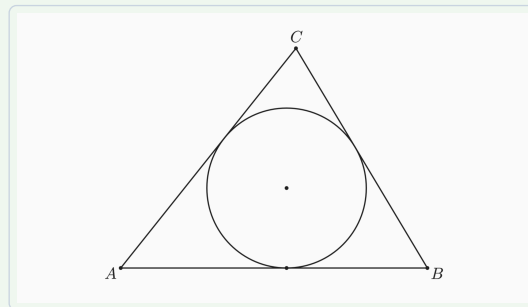
//Radius: the touch point is the foot of
//the perpendicular from I to a side
def T = CTClosestPoint.make(
  CTSegment.make(A, B), I)
```

```
scene.add(A, B, C, tri, I, T, incircle)
```

```
def incircle =
    CTCircle.makeCenterPoint(I, T)

camera.setGaps(.4, .4)
camera.adjustToObjects(A, B, C)

scene.add(A, B, C, tri, I, T, incircle)
```



Note that the angle bisector takes its vertex as the *middle* parameter, so the bisector from *A* is written `ctAngleBisector(A: B, B: A, C: C)`. Any two of the three do the job: the third one is a theorem, not a requirement, and drawing it is a good way to show that theorem.

The segment given to `on:` is auxiliary. It is never added to the scene, and it exists only so that the projection has something to land on. Writing the same step the long way needs one more object and says less:

```
def side = ctSegment(A: A, B: B)
def T = ctIntersectionPoint(a: ctLineOrthogonal(A: I, dir: side), b: side)
```

Discussion

Every piece of the figure is constructible, so nothing here is a number that goes stale. Move *A* with `animShift(obj: A, vector: [0, .8])` and the sides, the bisectors, the incenter, the touch point and the circle all follow, because each one is described in terms of the ones before it. That is the whole point of building the figure this way rather than computing the inradius and drawing a circle with it.

Tip

`type: "closest"` projects onto anything that owns points, not only segments. Passing the polygon itself, `ctPoint(type: "closest", on: tri, point: I)`, projects onto its boundary and gives a touch point without naming a side, although the three distances are equal here and which side wins is left to rounding.