Ratsub

A recursive graphics language

Adam Ponting

v1.0 October 2019

Tutorial and language specification

Contents

1. Ratsub tutorial

Ratsub is for drawing pictures using recursion and subdivision. A Ratsub program describes an initial shape, how to subdivide it into other shapes, and how long to keep subdividing. The shapes are made of points, which are joined with straight lines. When a shape reaches the maximum recursion level, specified in the shell command, the edge or interior of the shape, or both, are drawn.

Getting started

The Ratsub program on the right (yes, that's the whole program!) produces a black square with a white border.

```
def sq 4
```

Save it as `test.sdv` in the subdiv folder, then running the shell command `ratsub test 0` should produce a PNG image in the same folder. In the terminal you should see something like this:

```
bash$ cd subdiv
bash$ ratsub test 0
test 0
Making test-0.png...
Done.
```

IMage here

**** Then maybe do chess board?

Open a text editor, like SublimeText, VS Code, TextEdit, NotePad, TextWrangler etc—not a word processor, which may save text as rich text, MS Word or some other format. We want plain text files.

Type in this program, pressing `ENTER` after every line, including the last:

The indentation isn't necessary—it just makes the programs easier to read.

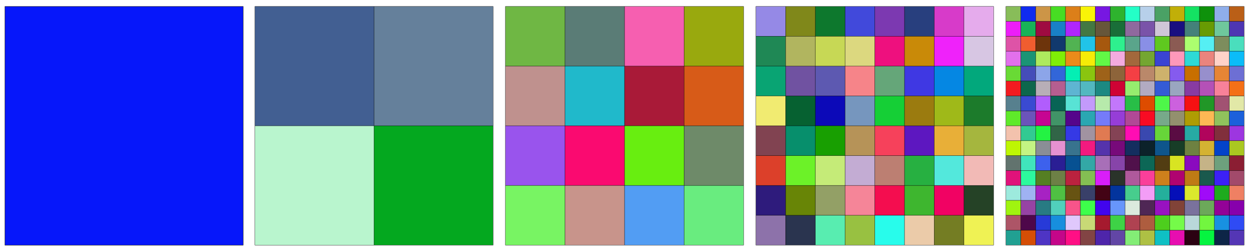
Save it as `test.sdv`, then on the command line enter:

```
def sq 4
  rand
  p4..7 p0.. 1/2 p1..0
  sq :415c
  sq :c526
  sq :04c7
  sq :7c63
```

```
$ ratsub test 0 4
```

Five `.png` images, named `test-0.png` to `test-4.png`, should appear in your current folder. If nothing happened, the error message should say why not, or at least where the compiler thinks an error is.

The images should look something like this:



Levels

As a Ratsub program runs, shapes are always called with an implicit level parameter, initially 0 for the first shape in the program, and a number of point parameters, which are implicit for the first shape.

A Ratsub program must be called with a level parameter indicating the maximum number of recursion levels to perform. The first shape in the program is level 0 the first time. The shapes it calls (other shapes, or itself, or both) are level 1. The shapes they call are level 2, etc.

When a shape is called at the maximum recursion level, it is drawn with its current colour setting. Nothing is drawn until the maximum level is reached—except using the `draw` command, which draws a shape immediately.

The shape calls are done *depth-first*, i.e. the first shape called goes all the way to max recursion depth before the second shape is called. And the order shapes are called affects what is on top, what is underneath. That can change the appearance of some programs entirely.

===Use examples! Especially those that go outside/overlap, e.g. the random IFS. Or even basic overlapping shapes. === Show example.

After 10 or 20 levels, the shapes are usually too small too see, and can take a long while to draw.[1]

***Do a diagram of here, with a program with two different shapes, showing the levels as rows, and what gets drawn when it stops at each level.

Also the ways of getting around that, with draw, wait, >3 (soon) etc...

The sketch

The first step in writing a Ratsub program is to *make a sketch* of your idea. Draw the initial shape and the shapes it will call or be divided into. Then number the points, preferably in a way that can be efficiently coded.[2]

*** Show 9-agon or something with midpoints, how that can be done with

```
pgon 9 50 0
p9..17 p0.. .5 p1..0
```

and how dividing a line into points can be done with

```
p4..12 p0 1..9/10 p1
```

[1]Have example somewhere of a shape where the shapes dont get smaller.. but change shape.. or something. Well, in the pythagorean tree, the volume stays the same. Hmm but that's the same as usual subdivision scenario... I guess the "twin trwindragon' is a good example - looks like the 2 shapes only change shape (although, yes, each shape gets smaller....)

[2]Put that better! and Show how this is done. using a pbox, a pgon, and manually-set initial points (perhaps do that first - with a triangle?)

More programs

*** Before explaining more, put programs here, each introducing some new feature, just showing how its used. AWK book does that. Just link ahead to where they're explained. Use simplest program to show how theyre used usefully.

***To to: so, make a list of the order things should be introduced! If possible.....

2. The structure of Ratsub programs

A Ratsub program is made of an optional preamble, followed by one or more shape definitions.

Initial points are defined with x,y coordinates, and other points are affine combinations of them, e.g.

p4 p0 .5 p1

The 0.5 means p4 will be the average or midpoint of the two, halfway from p0 to p1 — mathematically, $\frac{\texttt{p0 + p1}}{2}$. Changing the .5 to various other numbers will get you points on the line between p0 and p1.

p4 p0 0 p1 means p4 is at p0.

p4 p0 1 p1 means p4 is at p1.

p4 p0 2 p1 means go from p0 to p1, then go the same distance again to reach p4. i.e. not p0+p1 but p0+2(p1-p0), i.e. p1+(p1-p0)

You need to use `left` and `right` (see p. 15) to makes a new point off the line between p0 and p1.

Preamble

Defining initial points

Initial points became the parameters of the first shape, the first time through. A square needs 4 initial points, a hexagon 6 etc.

A single initial point is set like so:

```
p0 100 100
```

They must be numbered in order, starting from 0.

***** Give an example of a wacky star0like shape defined by initial points

Defaults: If p0 is omitted, it's set to 0,0. If no initial points are set at all, they default to the 4 points of `pbox`, i.e. `pbox 0 0 100 100`.

pbox

The four points forming a (non-tilted) rectangle can be set with

`pbox` *a b c d*

where *a,b* is the lower left vertex and *c,d* the upper right.

Short form	is short for
`pbox` *c d*	`pbox 0 0` *c d*
`pbox`	`pbox 0 0 100 100`

pgon

`pgon` *n r θ*

This sets the points of a regular polygon as initial points. e.g.

```
    pgon 5 50 20
```

makes p0 to p4 the 5 vertices of a regular pentagon, centred at (0,0) with radius 50, with p0 at 20° up from the x-axis, and the other points following anti-clockwise.

***__Really just better to show examples than try to explain, without them!!!!__

Shape definitions

Defining new points

New points can defined by saying how far the point is along the line from one existing point to another, e.g.

```
    p2 p0 1/2 p1
```

defines a new point `p2`, halfway from `p0` to `p1`. This also can be written

```
    p2 p0 0.5 p1       or
    p2 p0 .5 p1
```

par

```
p3 par p0 p1 p2
```

New points can also be defined using `par`, which draws the 4th point of a parallelogram, given the first 3[1], i.e.

********diagram here

New point numbering

The first new point must be numbered to come immediately after the shape's parameter points (or the initial points, for the first shape). e.g. for

```
def square 4
```

the parameter points—the points the shape is called with—will be named p0, p1, p2 and p3, and the first new point must be p4.

[1]In other words, `p3=p1-p0+p2=p2-p0+p1`.

Defining multiple new points

Short form	Equivalent to
`p4..6 p0.. .5 p1..`	`p4 p0 .5 p1`
	`p5 p1 .5 p2`
	`p6 p2 .5 p3`
`p4..6 p0.. .5 p1..0`	`p4 p0 .5 p1`
	`p5 p1 .5 p2`
	`p6 p2 .5 p0`

Shape calls

shape points options

A shape may have no shape calls. This means *draw it right away*, without waiting for the level to reach the max.

Short forms

Short form	Short for
shape `p1..5`	*shape* `p1,2,3,4,5`
shape `:14253`	*shape* `p1,4,2,5,3`

N.B. The :14253-type short form can only be used if every point in the list has a single-digit number! `p8,9,10` cannot be abbreviated to `:8910`.

3. Colouring

Colours are represented as *RGB* (red green blue), with the 3 values between 0 (off) and 1 (fully on) inclusive. Colours can be set with R G B values or with a colour name.

Colour can be applied in Ratsub in two different places in a shape definition:

- shape colour - usually in the line below the initial `def` line.

```
def sq 4
  red
  tri :012
```

- shape call colour. This sets the called shape's colour.

```
def sq 4
  tri :012 red
```

Also, gradient shading for a shape can be applied *after* new points for it have been defined.

Unless `onlyedges` is set, every shape is drawn with some colour. A shape's colour is either

1. its shape colour, if it has one
2. the colour given to it by the shape call it was called with,
3. otherwise, the default colour mid-grey (0.5,0.5,0.5).

Colouring syntax examples:

Method	Shape colouring	Shape call colouring
basic	`red`	`tri :012 red`
initial point shading	`ip0 red`	
gradient shading	`p0 red p1 green`	
colour-add mode		`tri :012 + 1 2 -1`
colour operator		`tri :012 not`

Basic colouring

Basic shape colouring

r g b
c (same as *c c c*)

colourname

Default: Black.

Basic shape call colouring

If a shape call has a colour, e.g.

```
sq :0123 red
```

it will pass that colour, otherwise it will pass its shape's colour, if it has one. Otherwise it will pass the colour its shape was passed when it was called.[1]

***is "passing" the best verb for this? Can I invent something better?

Random colouring

`rand`
`crand`

`rand` makes a random colour, different each time through the shape/shape call. `crand` is the same random colour each time.

Random range colouring

`rand` r_{min} r_{max} g_{min} g_{max} b_{min} b_{max}
`crand` r_{min} r_{max} g_{min} g_{max} b_{min} b_{max}

These make a random colour with red in the range $r_{min} \ldots r_{max}$, etc.

Shape shading

Apart from giving each shape a fixed flat colour, there are two ways to colour shapes with a varying shade.

Initial point shading

`ip`*n r g b*
`ip`*n colourname*

Initial point shading colours shapes by how close they are to a set of initial points.

(***edit this down:) It's done by specifying, in a shape definition, a set of initial points (the fixed points defined in the preamble) and their colours. When the shape is called, it's coloured by these points proportional to how close it is to them.

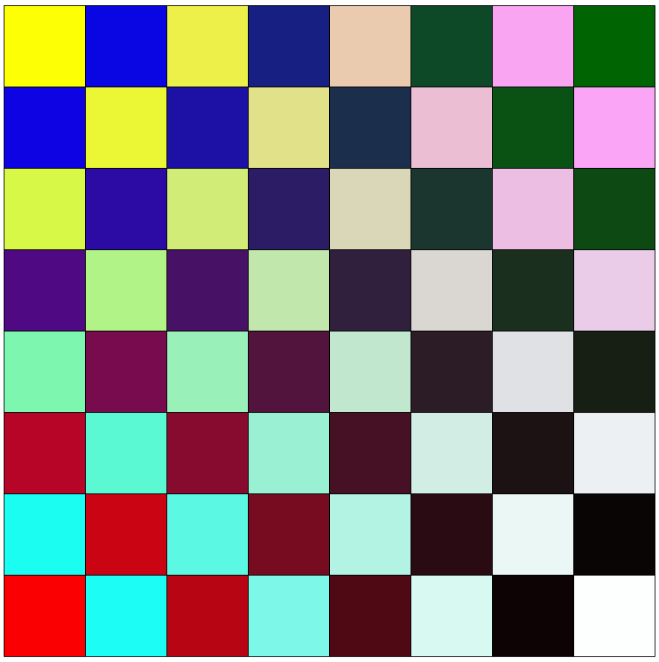
Extra initial points to control shading can be added to the preamble, and they will be ignored by the first shape, which only takes as many points as it's defined to need.

[1]I forget how this works.. I think if a program has any shape call colours at all, it goes into a different mode, where colours are inherited from passing shape. No? Otherwise all shapes use their own colour. I think. Ah, so maybe "=" was never defined because it does nothing?.... hmm Test and find out...

Multiple shading colours can be defined on one line.
*** but use something other than chessboard! just did that previously.

```
def s1 4
  ip0 red ip1 blue
  ip2 dark green ip3 black
  p4..7 p0.. .5 p1..0
  s2 :415c
  s1 :c526
  s1 :04c7
  s2 :7c63
def s2 4
  ip0 cyan ip1 yellow
  ip2 violet ip3 white
  p4..7 p0.. .5 p1..0
  s2 :415c
  s1 :c526
  s1 :04c7
  s2 :7c63
```

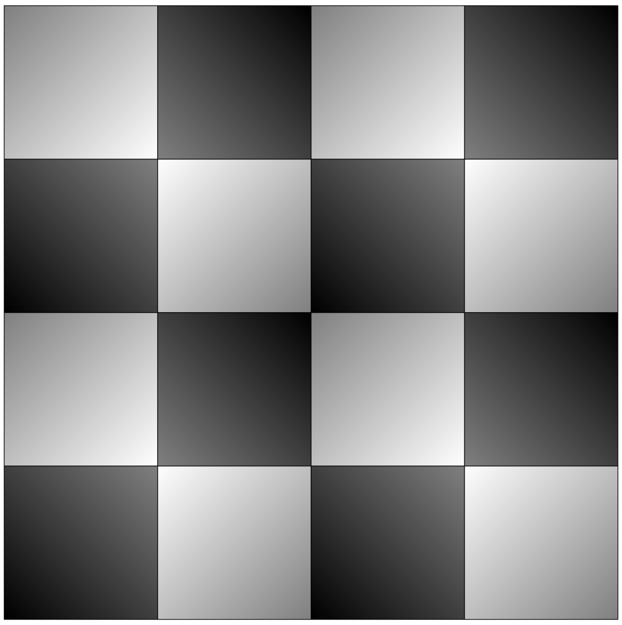
Level 3



Gradient shading

```
def sq 4
  p4 c
  p5..8 p0.. .5 p1..0
  sq2 :5164
  sq2 :4627
  sq2 :0548
  sq2 :8473
def sq2 4
  p4 c
  p5..8 p0.. .5 p1..0
  sqwhite :4516
  sqblack :4627
  sqblack :4805
  sqwhite :4738
def sqblack 4
  p2 black p0 grey
def sqwhite 4
  p0 white p2 grey
```

Level 2



You can also define new points to use with the gradient shading.
***Give example of that here.

Shape call colouring

Colour-add mode

shapename points + r g b
shapename points - r g b
shapename points + c
shapename points - c

This colours a shape depending on its call path down through the levels—on the add-colours of its ancestor shapes. It's used at the end of shape calls, e.g.

```
sq :0123 + 1 2 3
```

makes the red of the new `sq` shape 1/10th brighter, the green 2/10ths brighter, and the blue 3/10ths brighter. If the present shape's colour is (R,G,B), this calls `sq` with the colour (R+r/10,G+g/10,B+g/10). Because the colour adjustments are usually small fractions, the units are 10ths.

`sq :0123 + 1` means the same as `sq :0123 + 1 1 1`.

The initial shape takes its shape colour, but after that any shape colours are ignored in favour of the colour passed to it by the shape that called it.

Random

shapename points `random`

Colour operators

These are used in shape calls to transform the shape's colour

Operator	
`not/!`	R G B ⇒ 1-R 1-G 1-B. Black (0 0 1) becomes white (1 1 1), blue (0 0 1) becomes yellow (1 1 0) etc.
`rot`	Rotate colours to the right. R G B ⇒ B R G.
`rot2`	Rotate right twice/rotate left. R G B ⇒ G B R.

4. Random numbers

Random numbers can be used with colours, and with distances, for setting new points. *Random ranges* allow fine control over the desired random numbers.

***Currently rand/crand can only be used in shape defs. Hmm should be allowed to use at least rand in initial point defs. (crand will be the same, as initial points are only set once.)

*** Random numbers are very incompletely implemented. Write down exactly what they should do (based on what they already partly do!) and make sure they do it.

rand

```
p6 p0 rand p1
```

— this places p6 randomly somewhere on the line between p0 and p1 (including perhaps on those points). This random number is different every time the shape is called.

crand

```
p6 p0 crand p1
```

— this places p6 randomly somewhere on the line between p0 and p1 (including perhaps on those points). This random number is constant - the same every time the shape is called.

srand

`srand` *n*

— this seeds the random number generator. With the same *n*, the random numbers (and thus image) will be the same each time the program's run. For random numbers different each time without having to use `srand`, use `random` and `crandom` instead of `rand`.

Random range

new points

colours

5. More commands

`draw`

There are two ways of drawing a shape immediately—regardless of current level:

- Write `draw` instead of a shape name in a shape call. If a draw call has a colour, e.g.

```
        draw :0123 red
```

it will use that colour, otherwise it will use its shape's colour, if it has one. Otherwise it will use the colour its shape was passed when it was called.

- A shape without any shape calls will be drawn immediately.

`Draw` is most useful when the child shapes only cover part of the parent, (new metaphor?!) .. or lie outside it. ***EXAMPLES

`grid`

`grid` θ

`grid` transforms the usual square grid into a grid with the y-axis θ degrees from the x-axis. It only affects the initial points. e.g.

**Show diagram of this and example

Turning `left` and `right`

newpointdef `left` *a*
newpointdef `right` *a*

Turning is the only way to make a new point that's not on the line between its 2 defining points.[1]

```
p2 p0 .5 p1 left .25
```

This travels halfway from p0 to p1, then turns left and travels half that distance—the .25 means 1/4 the distance *between p0 and p1*.

***do a diagram of this!!

[1](***Use Koch curve as first example)

`flip`

shapecall `flip`

Used to call a flipped-over version of a shape, especially a line, for example in line fractals like the Koch curve, and IFS.

Flipping a shape only makes a difference if `left` or `right` are used in the shape's new point definitions.

***examples

Wait shapes and `wait`

Used at the end of shape calls:

`wait`	makes a wait shape, i.e. wait 1 level then call.
`wait` *n*	makes *n* linked wait shapes, i.e. wait *n* levels then call.

temp

Give example of pics needing 2, 3 wait shapes. Also multiple wait shapes, calls to different numbers in different places.

Conditionals

`>n        =n        >-n        =-n`

With > or =, you can do different shape calls[1] depending on the current level.

```
def sq 4
  tri :012
  tri :032
  >2
  tri :123
  tri :103
```

This does the first two shape calls if the current level<=2, and the second two if it's >2. i.e. :

If level>2 then do C and D, else do A and B

A negative branch number $-n$ means *n levels before the maximum level*.[2]

[1]I was gonna say, you can perform different actions.. but what are the options? shape calls or....? hmm maybe set shading? draw? wait? uh....

[2]Maybe better to write `levels` instead of always 'the maximum level'. Then instead of 'the current level', 'the level'. Also, 'before' is time, 'above/below' is space.. what is best way of talking about it?

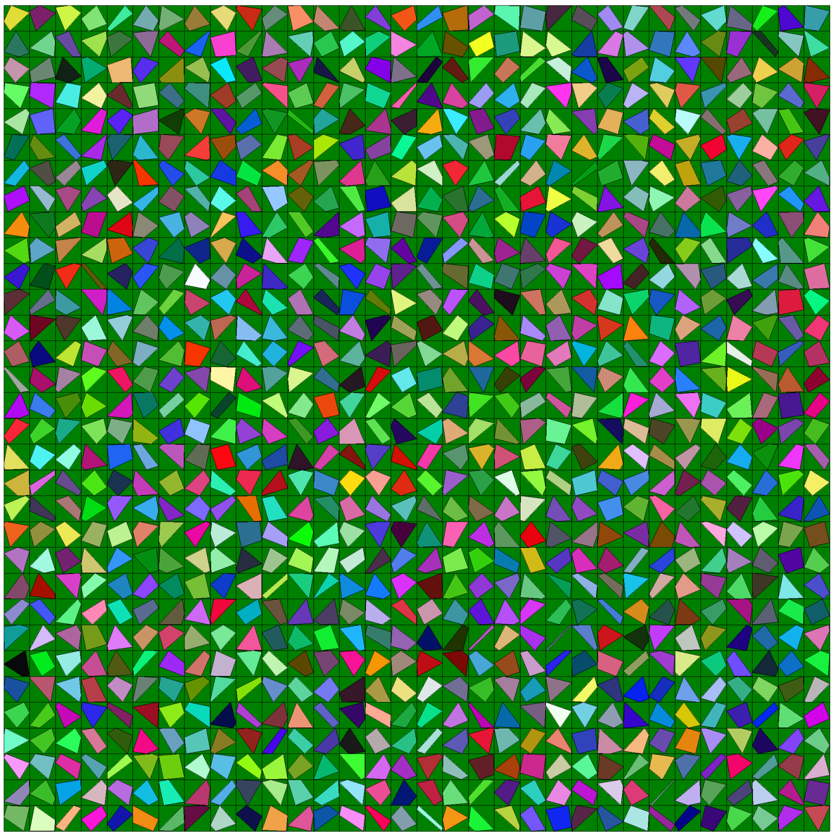
```
def sq 4
  tri :012
  tri :032
  >-2
  tri :123
  tri :103
```

Here the second branch is taken when the current level is more than 2 less than the maximum level. For example, if the program is run with "`ratsub myprog 8`"[43] then the second branch is taken when the current level >6.

Similarly, `=5` means ”take this second branch when the current level is 5”; `=-5` means ”take this second branch when the current level is 5 less than the maximum level”.

```
width .02
def sq 4
  p4 c
  p5..8 p0.. .5 p1..0
  sq :5164
  sq :4627
  sq :0548
  sq :8473
  =-2
  sq1 :0123
  draw :0123 green
def sq1 4
  p4..7 p0.. rand p1..0
  draw :4567 rand
```

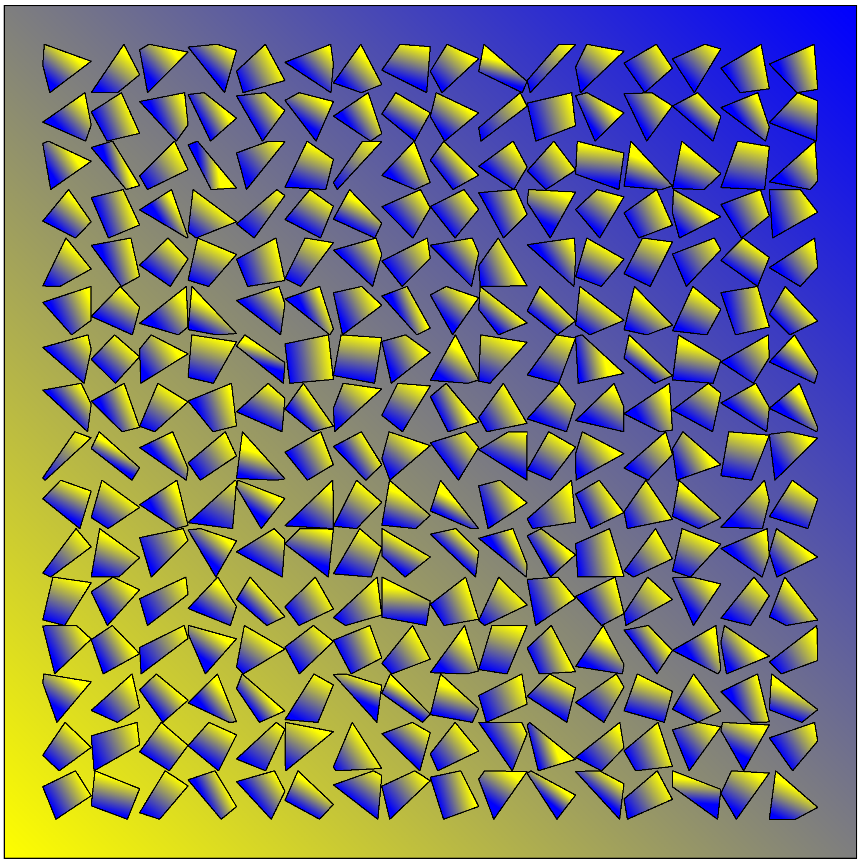
Level 7



[43]But really, explain in the start of book what is maxlevel (maybe that's a good word for it!) so don't need to explain here, most of the way through book.

```
margin 6
def frame 4
  p4 c
  p5..8 p4 1.1 p0..
  sq :0123
  backgr :5678
def backgr 4
  p0 yellow p2 blue
def sq 4
  p4 c
  p5..8 p0.. .5 p1..0
  sq :5164
  sq :4627
  sq :0548
  sq :8473
  =-2
  sq1 :0123
def sq1 4
  p4..7 p0.. rand p1..0
  shard :4567
def shard 4
  p4 p3 .5 p0
  p5 p1 .5 p2
  p4 blue p5 yellow
```

Level 7



***More sensible not to have margin, but to come INWARDS with the new points...

6. Drawing in 3D with pcube

pcube $r\ \theta_x\ \theta_y\ \theta_z\ z_{\text{viewpoint}}\ z_{\text{viewplane}}$

This adds 8 initial points, the vertices of a cube seen in perspective. The image is drawn as if the viewer's eye is at (0,0,$z_{\text{viewpoint}}$) and the viewplane[1] the plane $z = z_{\text{viewplane}}$.

N.B. The algorithm involves dividing by $z - z_{\text{viewpoint}}$, so will make an error if that's 0, i.e. if you get so close that your eye hits a vertex of the cube! [2]

[1]Is it called that? Guess so

[2]Better: don't say this, just make the error message say that.!!

7. Ratsub language reference

Levels

Each Ratsub program specifies a (theoretically) infinite number of pictures, depending on the level parameter which sets the maximum recursion level/call stack depth. The first shape in a program calls others, they call others etc. When the recursion reaches N levels deep, a shape instead draws itself using its parameter points and colour settings.

This can be bypassed with the `draw` command in a shape's definition—the shape will be drawn immediately.

Program structure

Structure of a Ratsub program

1. optional settings
   - `margin` f
   - `grid` θ
   - `no edges`
   - `only edges`
   - `background` r g b
   - `width` f
2. initial points
   - point definitions
     - `p`n x y
   - `pbox` x y
   - `pbox` x_1 y_1 x_2 y_2
   - `pgon` n r θ
   - `pcube` r θ_x θ_y θ_z $z_{\text{viewpoint}}$ $z_{\text{viewplane}}$
3. One or more shape definitions

Structure of a SHAPE DEFINITION

1. `def` *shape n*
2. optional
   - colour setting
     - *colourname*
     - *r g b*
     - *c*
     - `rand`
   - shading points
     - `ip`*n colourname*
     - `ip`*n r g b*
3. 0 or more new point definitions
   - single
     - p*n* c
     - p*n* p*q* *a*/*b* pr [TURN]
     - p*n* p*q* *a* pr [TURN]
   - multiple
     - p*m*`..`*n* p*q*[`..`] *a* p*r*[`..`][*s*] [TURN]
     - p*m*`..`*n* p*q*[`..`] *a* … *b*/*c* p*r*[`..`][*s*] [TURN]
4. 0 or more SHAPE CALLS
5. optional
   - branching
     - >n
     - >-n
     - =n
     - =-n
   - then 0 or more SHAPE CALLS

TURN
- left *a*
- right *a*

A SHAPE CALL
-
  - *shapename*
  - `draw`
- points
  - p$n_0, n_1, \ldots, n_k$
  - :$n_0 n_1 \ldots n_k$
- optional
  - colour
    - *colourname*
    - *r g b*
    - `not`
    - `rot`
    - `rot2`
    - `=`
    - `+/-` *r g b*
    - `+/-` *c*
  -
    - `wait`
    - `wait` *n*

Program structure in detail

Optional settings

`grid` θ	Makes the angle between the x and y axes θ degrees, for the initial point definitions.
`margin` *num*	set width of the margin around the picture
`margin` *l t r b*	set width of the left, top, right, bottom margin
`margin` *a b*	*a* is left/right margin, *b* is top/bottom margin.
`no edges`	no shape edges are drawn
`only edges`	only edges are drawn
`background` *r g b*	colour of the background bounding box
`width` *num*	set line width for drawing shape edges.

Initial point definitions

These define the vertices of the initial shape, which are initially passed as parameters to the first shape definition. If `grid` is set then these points will be affected. Use " to repeat an x- or y-coordinate from the line above.

`p3 5.5 2.3`	defines point 3 at 5.5,2.3
`pbox` *a b c d*	defines four points, the corners of a rectangle numbered clockwise from lower left. The new points p0 . . . 3 are: *a,b a,d c,d c,b*.
`pbox` *c d*	short for `pbox 0 0` *c d*
`pbox`	short for `pbox 0 0 100 100`
`pgon` $N\ r\ \theta$	Defines the N points of a polygon, with one more point at the centre 0,0. The point have radius r and p0 is at θ degrees. The N points are numbered anticlockwise.[1]
`pcube` $r\ \theta_x\ \theta_y\ \theta_z\ z_{\text{viewpoint}}\ z_{\text{viewplane}}$	Defines the 8 points of a cube in 3D.

Shape definitions

These define shapes and how they'll be subdivided into other shapes. Shapes can also be subdivided into themselves, as in the example program above.

[1] Explain somewhere about setting colours. With *r g b* or *colourname* or `#fff` or `#ffffff` etc

Initial point definitions and the first shape

Ratsub manual: The first shape defined is drawn with the initial points from the preamble, using as many points as it needs, e.g. "def sq 4" defines a shape "sq" with 4 points, which will take p0, p1, p2 and p3 of the initial points the first time it's called, and the 4 points it's called with each time after that. So only the first shape defined can take initial points as parameters. After the first shape has been called once, the initial points can only be used by shapes for their shading. "p0" then means, the first parameter in the call to that shape, and not the first initial point.

The `def` line

`def` *shape N*
N is the number of points in the shape, the number parameters it takes from the shape that calls it. The first shape, the first time it's run when the program starts, takes the initial points (defined in the preamble) as its parameters.

Shape colour setting (optional)

colourname	set the fill colour.[1]
r g b	set the shape colour to red *r*, green *g*, blue *b*. (colour values are between 0 and 1)
c	set the shape colour to red *c*, green *c*, blue *c*.
`rand`	random colour
$+\ \Delta r\ \Delta g\ \Delta b$	
$-\ \Delta r\ \Delta g\ \Delta b$	use colour-add mode

Shading point definitions

`ip0` *r g b*	Assigns the colour *r g b* to the initial point 0, the first defined in the preamble (Not the first parameter of the shape!)
`ip0..3` *r g b*	Assign the same colour to initial points 0,1,2,3.

New point definitions

	Defines...
`p2 p0 1/3 p1`	a new point p2, 1/3 of the way from p0 to p1.
`p2 p0 0.4 p1`	a new point p2, 4/10 of the way from p0 to p1.
`p2 p0 rand|crand|random p1`	a new random point somewhere on the line between p0 and p1.
`p4 c`	defines a new point, the average of the shape's parameters (or the initial points, for the first shape).

New point definition short forms

```
p3 p1 " p2                uses the same fraction/number as the previous line.

p4..6 p0 1..3/4 p1        p4 p0 1/4 p1
                          p5 p0 2/4 p1
                          p6 p0 3/4 p1

p4..6 p0 1,2,4/5 p1       p4 p0 1/5 p1
                          p5 p0 2/5 p1
                          p6 p0 4/5 p1

p5..7 p1..  .95 p0        p5 p1 .95 p0
                          p6 p2 .95 p0
                          p7 p3 .95 p0

p5..7 p0 .95 p1..         p5 p0 .95 p1
                          p6 p0 .95 p2
                          p7 p0 .95 p3

p5..7 p0..  .95 p1..      p5 p0 .95 p1
                          p6 p1 .95 p2
                          p7 p2 .95 p3

p5..7 p0..  .95 p1..0     p5 p0 .95 p1
                          p6 p1 .95 p2
                          p7 p2 .95 p0
```

***DO PICS OF THESE!!!

This last form is frequently useful, e.g. for the midpoints of the sides of a pentagon :

```
pgon 5 50 0
def pent 5
 p5..9 p0.. 1/2 p1..0
```

as the last side is from p4 back to the first point p0.

Maybe draw these situations?