**Thirteen: a Library for Constructive Planar Geometry in Processing**

Thirteen is a Processing library for the construction of geometric figures. It is based on geometric primitives and their intersections.

Users generate geometric diagrams by describing them. Geometric constructions have been used by architects, draftsmen, font designers, and mathematicians to do reasoning about proportion, composition, and arrangement. You could think of it as a system for constructing grids.

Engineers will notice that the library does not provide a single operation not already in Processing: granted. The point is not functionality, but interface.

Most of the history of geometry predates Descartes. Geometry that does not rely upon the use coordinate systems, precise measurement, or even algebra was the best tool available for much of human history. Modern geometry is more convenient and powerful technology for many tasks, but any technology is but a tool, and every tool has its own character.

Art prefers character to convenience. Using Thirteen is like using watercolors instead of Photoshop. By giving up the undo command, what is gained?

**Definitions**

A **geometric object** may be either a point, a line, or a circle.

A **figure** is a collection of geometric objects.

A **page** is a plane upon which a figure can be drawn. As all digital display and printing devices are Cartesian, pages are used to translate figures for display.

Geometric objects are named by integers, automatically, at the time of their creation.

Object 0 is the null object. Parallel lines intersect at the null point, for instance.

Figures are networks of geometric objects that can be applied to a figure. Procedures may be defined that generate marks from other marks.

**All figures contain, at their creation, given h, and w:**

0: the null object

1: a point (generally taken to be the origin, though that’s irrelevant)

2: a horizontal line through A

3: a circle of radius w, whose center is A

4: a circle of radius h, whose center is A

These are the “given” objects.

**The Basic Operations on a Figure**

1: draw a line through two points

2: draw a circle with center p1 through point p2

3: draw an ellipse with centers p1 and p2, with extreme point p3

4: get the “first” intersection of two primitives (lines, circles, or ellipses)

The function takes the two primitives and a point, A.

If there are no points of intersection, return 0, the null object

If there is one point of intersection, return that.

If there are two points of intersection, return the one closest to A.

If they are equidistant from A, return the top one.

If they are level, return the leftmost one.

5: get the second intersection

If there are not two intersections, return null,

otherwise, return the point that is not the first intersection

**The Standard Page Setup**

Using the standard operations, these objects are generated from the given objects in the constructor.

5: The leftmost intersection of 2 and 3

6: The rightmost intersection of 2 and 3

7: The circle centered at 5 going through 6

8: The circle centered at 6 going through 5

9: The lower intersection of 7 and 8

10: The line through 1 and 9 (perpendicular to 2, the left margin)

11: The intersection of 3 and 10

12: The intersection of 4 and 10 (the bottom-left of the page)

13: The circle centered at 11 that goes through 1

14: The circle centered at 6 that goes through 1

15: The intersection of 13 and 14 farthest from 1 (1 is the other one)

16: The line through 6 and 15 (perp to 2, the right margin);

17: The rightmost intersection of 2 and 4

18: The circle with center 17 that goes through 1

19: The circle with center 12 that goes through 1

20: The intersection of 18 and 19 farthest from 1

21: The line through 12 and 20 (the lower margin)

22: The intersection of 21 and 16 (the lower-right corner)

Fun facts about the standard page!

The points 1, 6, 22, and 12 are the corners of the page.

The lines 2, 10, 16, and 21 are the margin lines.

The points 1, 6, 15, 11 give a square of width w.

The intersection points of circles 3 and 14 give the vertical centerline.

The intersection points of circles 4 and 19 give the horizontal centerline.

The angle 1,9,6 is 30 degrees.

These properties hold for all values of w and h.

**Convenience Functions**

The following operations are built from the fundamental functions. They add primitives other than the returned objects to the figure.

LineThroughPointAPerpendicularToLineB

ClosestPointOnLineAToPointB

LineParallelToAThroughPointB

MidpointBetweenPointsAB

DiagonalUpFromLineAThroughPointB

TriangleSimilarToABCusingDEasAB

SubdivisionRig(PointA, PointB, numberOfDivisions): array

SubdivisionN(array, n);

**Examples of Using Thirteen**

Given the initial figure, construct a line perpendicular to 2 through 1

a = Figure.addCircle(4, 8);

b = Figure.addCircle(5, 7)

c = Figure.firstIntersection(a, b, 1); // upper intersection of ab

d = Figure.secondIntersection(a, b, 1); // lower

e = Figure.addLine(c,d); // the perpendicular through 1

Given the initial figure, construct a line parallel to 2 and distance h below it

E as above

F = secondIntersection, 6, e

G = circle through 1 at g

H = circle though 1 at 8

I = secondIntersection GH

J = line through F and I // J is || to 1

The area bound by 1, e, and J is the “default page”

Given the initial figure, construct a golden rectangle.

E = as above!

f = Figure.secondIntersection(e, 3); // the lower-left corner of a square of width w

g = midpoint(1,f)

h = add circle (g, 5)

I = secondIntersection(h, e, 1)

J = line parallel to 2 through i

K = line parallel to e through 5

L = firstIntersection of j and k

/// 1, 5, L, I is a golden rectangle.

Page Layout:

Given the standard page, construct the Villard Diagram for proportional layout of text.

A = firstIntersection(3, 14, 9);

B = secondIntersection(3, 14, 9);

C = addLine(a,b); // vertical midline

D = addLine(1, 22); // diagonals

E = addLine(6, 12);

F = firstIntersection(1, c);

G = addLine(f, 22);

H = firstIntersection(e, g);

I = addLine

The points 1, 6, 22, and 12 are the corners of the page.

The lines 2, 10, 16, and 21 are the margin lines.

The points 1, 6, 15, 11 give a square of width w.

The intersection points of circles 3 and 14 give the vertical centerline.

The intersection points of circles 4 and 19 give the horizontal centerline.

The angle 1,9,6 is 30 degrees.

**Cheater Functions**

These operations violate the purity of the initial project for the sake of being awesome.

CheaterPoint(x, y)

CheaterCircle(a, r);

CheaterEllipse(a, b, r);

ParametricPoint(object, t);

// for lines, returns a point between (0<t<1) the two points that define the line.

// for circles, returns a point on the circle, t=0==the rightmost point, going counterclockwise, returning there at t=1

**Extracting A Drawing from Intersection**

GetPositionX(ob);

GetPositionY(ob);

Get

**Patterns.**

typedef struct Pattern {

int name, inputCount, outputCount;

geObj objs[100];

}

Defining patterns.

An operation is a figure with anchor objects. If a figure becomes able to take a name and recognize how to assemble itself using a provided list of objects, it is essentially a function.

Figures are trees; patterns are also trees. The “input” fields link object-nodes. Therefore, a construction “function” is simply a list of nodes, that make a tree, with some kind of internal name/marker/notation for matching given objects to needed inputs.

So, object names need to become more complex.

In manually-constructed figures, names simply serve as tags, so that objects can refer to each other.

In figure definitions, names are local. Objects in a definition, referring to other objects in the definition, use one convention. References to parameters are handled differently. Once built, all the elements provided by the definition have names within the figure into which they are applied.

Pattern definitions need to get attached to something. They should be loadable and save-able—then we’ll have fonts! In no time at all, clearly.

These routines may be used to generate and manipulate patterns

Pattern:Define(): int;

Pattern:AddArgument(int pattern, int object): int objName;

Pattern:AddOperation(int pattern, int obj1, int obj2, int op): int result;

Pattern:End(int pattern, int result); // tells which in-pattern object to return

Pattern:Object(int pat, int name): obj;

Pattern:Set(int);

Pattern:Argument(int pattern, int patternObjectName, int obj);

Pattern:Apply(int pattern): obj;

Shouldn’t you just use procedures? Yes.

Dealing with constants

Circles are always built out of two points. Circles of half that radius can be made by first subdividing the line. Angles are similarly subdivided. At no point do angles, slopes, or radii need to be given as numbers.

Operations that generate these objects can be built from loops. Grids, also. There is a role for constants.

Drawing

Expectations are set by PS and PDF.

To draw a letter, you must be able to define regions to fill.

As soon as you start filling regions, you’ll want to use a pattern.

As soon as you have a pattern, you’re clipping.

To clip, you need to define regions. To define regions, you could use ginormous polygons, or triangles—but of course clipping triangles and wedges to triangles and wedges is a fool’s errand.

Or, you could just use PostScript/PDF

Let’s do that.