COLLECTED NOTES ON GEOMETRIC MODELS

William Bricken
March 1988

A (PARTIAL) THEORY OF 2D-POINTS

Objects:
    a, b ,c POINT

Functions:
    Scale, Transpose, Distance

Relations:
    IsEqualTo,
    IsLeftOf, IsRightOf,
    IsAbove, IsBelow,
    IsHorizontalWith, IsVerticalWith

Definitions using the Theory of Real Numbers

POINT:
    (x,y) *a* pair of REAL numbers

Scale[*a* POINT by *a* REAL factor]:
```
newpoint.x = factor * point.x
newpoint.y = factor * point.y
```

Translate[*a* POINT by *a* POINT deviation]:
```
newpoint.x = deviation.x + point.x
newpoint.y = deviation.y + point.y
```

Distance[from POINT1 to POINT2]:
```
newreal = Sqrt[ (point2.x - point1.x)^2
              + (point2.y - point1.y)^2 ]
```

POINT1 IsEqualTo POINT2:
```
(point1.x = point2.x) and (point1.y = point2.y)
```

POINT1 IsLeftOf POINT2:
```
(point1.x < point2.x)
```

POINT1 IsRightOf POINT2:
```
(point1.x > point2.x)
```

POINT1 IsAbove POINT2:
```
(point1.y > point2.y)
```

```
POINT1 IsBelow POINT2:
      (point1.y < point2.y)

POINT1 IsHorizontalWith POINT2:
      point1.y = point2.y

POINT1 IsVerticalWith POINT2:
      point1.x = point2.x
```

Definitions using the Theory of Pairs of Reals

```
POINT:
      (x,y) a PAIR of REALS

Scale[a POINT by a REAL factor]:
      newpoint = factor * point

Translate[a POINT by a POINT deviation]:
      newpoint = deviation + point

Distance[from POINT1 to POINT2]:
      newreal = Sqrt[ (point2.x - point1.x)^2 + (point2.y - point1.y)^2 ]

POINT1 IsEqualTo POINT2:
      point1 = point2

POINT1 IsLeftOf POINT2:
      point2 - point1 = (+,y)

POINT1 IsRightOf POINT2:
      point2 - point1 = (-,y)

POINT1 IsAbove POINT2:
      point2 - point1 = (x,+)

POINT1 IsBelow POINT2:
      point2 - point1 = (x,-)

POINT1 IsHorizontalWith POINT2:
      point2 - point1 = (x,0)

POINT1 IsVerticalWith POINT2:
      point2 - point1 = (0,y)
```

Operator overloading permits both reals and vectors to use + and *. Note the symbol overloading in permitting + and - to be objects with a Class meaning as well as operators with a Method meaning. The use of free variables in the descriptive pattern of a point permits reference to arbitrary values, without having to commit to any specific value. Could be implemented with lazy evaluation, ie: if the pattern is free, then don't make the function call.

A natural POINT language and theory makes talking about and coding with points easy. The technical details of making points work by using REALS (for instance) are hidden. That is not to say that we want to talk about or code in points. There is an abstraction hierarchy, we can add and remove layers as we please.

By building a Vector Class, we can speak in a more convenient language, and we never have to worry about x and y details. We can even forget about dimensionality, since we assume that 3D vectors will be handled appropriately. This has nothing to do with implementation. Vectors can be implemented by Lists, by Arrays, by Reals, or even by Bits, by SystolicArrays, or by FingersPointingInCyberspace.

The conceptual win is that the implementation details are totally segregated from the mathematical language, and the mathematical description language funnels and constrains our imagination into a conceptual model. One of the major differences between engineer and artisan is that the engineer uses models constrained by external reality, while artisans use models constrained by internal reality.

If we accept engineering as using real world models, sophisticated CAD must also know about real world models. When we get beyond POINTS and LINES, we will encounter WEIGHTS and MEASURES.

USING THEORY OF PAIRS AS A GEOMETRICAL BASIS

The Theory of Pairs

Objects:

```
      a,b,c,...  PAIRS
            composed of  x,y,...  ATOMS
```

Functions:

```
      Constructor:  {x1,x2}                      to pair up
      First:        First[{x1,x2}] = x1
      Second:       Second[{x1,x2}] = x2
```

```
Relations:
      Atom[x]
      Pair[a]

Axioms:
      Generate pair:  Pair[a] == Atom[a.x1] and Atom[a.x2] and (a = {x1,x2})
      Disjoint:       not (Atom[x] and Pair[x])
      Unique:         {x1,x2} = {x3,x4}  ->  (x1 = x3) and (x2 = x4)

Computational Techniques:
      Substitution:   (x1 = x3)  ->  {x1,x2} = {x3,x2}
      Decomposition:  Pair[a]  ->  a = {First[a], Second[a]}
```

Interpretation for Planar Geometry

```
POINT:
      a PAIR of REALS

LINE:
      a PAIR of POINTS

Pointing:
      (x1,x2)
Lining:
      (p1,p2)
```

Operator overloading and internal structure determines the meaning of parentheses.

```
Relations:
      Real[x]
      Point[p]
      Line[d]

Line[d] == (Point[d.p1], Point[d.p2])
        == ((Real[d.p1.x1], Real[d.p1.x2]), (Real[d.p2.x1], Real[d.p2.x2]))

Point[p] == (Real[x1], Real[x2])

First[d] == Point[d.p1]

First[p] == Real[p.x1]

Second[d] == Point[d.p2]

Second[p] == Real[p.x2]
```

Using Points and Heading (a vector)

That a LINE can be described by a POINT and a VECTOR just means that we are changing our *interpretation* of the basic underlying mathematical structure. That is, nothing changes but the implementation!

LINE:
    a mixed PAIR of (POINT, VECTOR)

POINT:
    a PAIR of REALS

VECTOR:
    a mixed PAIR of (HEADING, REAL)

HEADING:
    a PAIR of REALS

Structure of a Line defined by a Vector:
    Line[d] == ((REAL, REAL), ((REAL, REAL), REAL))

By building up the internal structure of our definition, we can develop a syntax that is concise. Trading off internal structure (definition) and external structure (rules) defines the evolution of mathematics.

The main idea is that the place we chose to stop and turn the implementation over to the machine can always be reached by syntax converters (pre-processors, compilers). There is no need to hobble our model with machine dependencies, or even with language dependencies. We can chose to model lines with point-point pairs, or with point-vector structures, or with *whatever is easiest for our conceptualization*. Syntax conversion generically modifies our model to fit the implementation architecture of choice. All this is called isomorphism: the organization stays the same, while we twiddle with the structure to achieve portability, efficiency, and understanding in different contexts.

MODELING LINES WITH TURTLE VECTORS

STATE:
    a mixed PAIR of (LOCATION, HEADING)

LOCATION:
    a POINT, which is a PAIR of REALS

HEADING:
    a VECTOR, which is a PAIR of REALS

Structure:

```
      State[m] == ((REAL, REAL), (REAL, REAL))
      Line[d] == ((REAL, REAL), (REAL, REAL))
      Path[t] == (State[0.p], ..., State[current.p])
```

In this model, lines are not in the definition, they are in the rules. A LINE is the difference between your current location and your previous location. This works because the origin is shifted dynamically, which means

```
            State[m] = ((0,0), HEADING)
```

at all times except during the transformation called translation. The difference is recorded in the PATH.

Functions:

```
      Translate[m] == State[m1.p1] = State[m2.p1]
                        and
                      Path[t] = Path[t] + State[m1]
```

ADD methods for:

```
      CONVERSION from one representation to another
      IMPLEMENTATION pass off to machine
      GENERATORS, etc.
```

WORK AND IDEAS FROM OTHER FOLKS

Paraphrased from Meyer

```
class POINT  with
      x,y         REAL
      T,S,D       FUNCTION

S[REAL factor]  is            (*scale by factor*)
      x := factor * x
      y := factor * y

T[REAL dx, REAL dy]  is       (*translate by dx and dy*)
      x := x + dx
      y := y + dy

D[POINT other]  is            (*distance to other point*)
      Sqrt[ (x - other.x)^2 + (y - other.y)^2 ]
```

Paraphrasing Bundy

```
Triangle[a,b,c]  is
      not[a = b]] and not[a = c]] and not[b = c]]
        and
      not[collinear[a,b,c]]

Symmetry:
      line[a,b] = line[c,d] -> line[a,b] = line[d,c]

Familiar axioms:
      equality of angles and lines
      congruence
      parallelism

      Triangle[a,b,c] = Triangle[d,e,f] -> line[a,b] = line[c,d]

      Angle[a,c,b] = Angle[d,f,e]
        and
      Angle[c,a,b] = Angle[f,d,e]
        and
      Line[b,c] = Line[e,f]  ->  Triangle[a,b,c] = Triangle[d,e,f]
```

Some fragments of Boundary Representations for 3D

```
VERTEX:
      (v POINT;
       alternative:  x,y,z NUMERICAL)

FACE:
      (f PLANE;
       alternative: a,b,c NUMERICAL;
            (ax + by + cz + 1 = 0) )

EDGE:
      (e LINE;
            (x = (y - y0)/a = (z - z0)/b) )

POLYHEDRAL TOPOLOGY:
      (f FACE-CLASS;
       v VERTEX-CLASS;
       e EDGE-CLASS;
            (f Surround f,f,f,f)
            (f Surround v,v,v,v)
            (f Surround e,e,e,e)
            (v Surround f,f,f)
            (v Surround v,v,v)
            (v Surround e,e,e)
            (e Surround f,f)
            (e Surround v,v)
            (e Surround e,e,e,e)
```

Paraphrase Rankin

```
POINT:
      (x,y COORDINATES;
       alternative: r LENGTH;
                    theta ANGLE;
            (x = r*Cos[theta])
            (y = r*Sin[theta])
            (r = Sqrt[x^2 + y^2])
            (theta = Arctan[y/x]) )
```

Due to the computational complexity of trigonometric algorithms, software models prefer Cartesian coordinates. Graphics algorithms prefers to avoid angles (the seam between theta = 0 and theta = 2Pi).

```
ILINE:
      (theta ANGLE;
       p POINT;
            (m = Tan[theta])              Free to use new variables
            (p.y = m*p.x + c)             c is undefined,
                                            how to say "y-intercept"?
                                            Also, not valid for x = 0.
       alternative:  t PARAMETER;
                     b,e POINT;
            (x = b.x + t*e.x)
            (y = b.y + t*e.y)

LINE:
      (b,e POINT;
       lambda PARAMETER;
            (x = lambda*b.x + (1 - lambda)*b.y)
            (y = lambda*e.x + (1 - lambda)*e.y)
            (0 ≤ lambda ≤ 1) )
```

Miscellaneous notes

```
GKS PRIMITIVES
      POLYLINE:
      POLYMARKER:
      FILL-AREA:
      TEXT:
      CELL-ARRAY:

GEOMETRICALLY COMPLETE MODELS
      Spatial Enumeration
      Primitive Instancing
      CSG
      Boundary
      Sweeps
```

Paraphrased from Tyugu

```
POINT:
      (x,y,z  LENGTH)
LENGTH:
      NUMERIC

POINT:
      (r  LENGTH;
      phi, theta  ANGLE)
```

```
POINT:
      (x,y,z  LENGTH)  has
      (alternative:  r LENGTH;
                     phi, theta ANGLE;
       converters:  (r^2 = x^2 + y^2 + z^2)
                    (r * Sin[theta] = z)
                    (r * Cos[phi] = x)
                    (r * Sin[phi] = y)  )
```

This assumes an intelligent constraint engine. Note that the last converter is redundant, can be added anytime for efficiency.

Alternatives to a Class are just different structures that achieve the same organization.

Here LINE inherits from POINT:

```
LINE:
      (p,q POINT) has
      (alternative:  len LENGTH;
                     slope ANGLE;
      converters:  (Sin[slope] * len = q.z - p.z)
                   (len^2 = (p.x - q.x)^2 + (p.y - q.y)^2 + (p.z - q.z)^2) )
```

TYUGU'S SIMPLE GEOMETRIC OBJECTS

```
Basic concepts:
      SIDE,
      DIAGONAL,
      HEIGHT,
      RADIUS,
      DIAMETER,
      ANGLE,
      PERIMETER,
      AREA,
      VOLUME:  NUMERIC
      Constant:  Pi = 3.14159

SQUARE:
      (b SIDE;
       d DIAGONAL;
       p PERIMETER;
       s AREA;
            (s = b^2)
            (d^2 = 2*b^2)
            (p = 4*b)  )
```

```
CIRCLE:
      (r RADIUS;
       d DIAMETER;
       c PERIMETER;                       circumference
       s AREA;
            (s = Pi*r^2)
            (d = 2*r)
            (c = 2*Pi*r) )
```

Note that the concepts of Perimeter and Circumference have been grouped into a more abstract concept of distance around the edge. The structural similarities between circles and squares can be abstracted using this map:

```
      unit       = (side, radius)
      transverse = (diagonal, diameter)
      perimeter  = (perimeter, circumference)
      area       = (area, area)
```

We can now condense the Basic Concepts into a smaller set, extending the similarity between Perimeter and Circumference to all other elements in this model.

```
SQUARE-CIRCLE-ABSTRACTION:

      PARAMETER-TABLE          SQUARE      CIRCLE
      unit                      side       radius
      transverse-parameter     Sqrt[2]      2
      perimeter-parameter        4          2*Pi
      area-parameter             1            Pi

            (transverse = transverse-parameter * unit)
            (perimeter = perimeter-parameter * unit)
            (area = area-parameter * unit^2)


                         Square-Circle
                        /             \
                     Square          Circle
                       |                |
                  (Sqrt[2],4,1)     (2,2*Pi.Pi)
```

Here we have what amounts to DIMENSIONAL ANALYSIS. Transverse and Perimeter are 1D, while Area is 2D. Both CIRCLE and SQUARE inherit the abstract structure. In the process of inheriting it, they set their local geometric parameters.

```
SPHERE:
      (r RADIUS;
       d DIAMETER;
       s AREA;
       v VOLUME;
            (d = 2*r)
            (s = 4*Pi*r^2)
            (v = (4/3)*Pi*r^3) )

CUBE:
      (b SIDE;
       d DIAGONAL;
       s AREA;
       v VOLUME;
            (d^2 = 3*b^2)
            (v = b^3) )

TETRAHEDRON:
      (b SIDE;
       s AREA;
       v VOLUME;
            (s = Sqrt[3]*b^2)
            (v = (Sqrt[2]/12)*b^3) )
```

The DIMENSIONAL ABSTRACTION includes 3D objects:

PARAMETER-TABLE	SQUARE	CIRCLE	SPHERE	CUBE	TETRAHEDRON
unit	side	radius	radius	side	side
transverse-parameter	Sqrt[2]	2	2	Sqrt[3]	1
perimeter-parameter	4	2*Pi	-	12	6
area-parameter	1	Pi	4*Pi	6	4
volume-parameter	-	-	(4/3)*Pi	1	Sqrt[2]/12

```
            (volume = volume-parameter * unit^3)
```

```
RECTANGLE:
      (b1,b2 SIDE;
       d DIAGONAL;
       p PERIMETER;
       s AREA;
            (d^2 = b1^2 + b2^2)
            (p = 2*(b1 + b2))
            (s = b1*b2)  )
```

```
RHOMBUS:
      (b SIDE;
       d1,d2 DIAGONAL;
       p PERIMETER;
       s AREA;
       a1,a2 ANGLE;
       h HEIGHT;
            (a1 + a2 = 90)
            (Cos[a2/2]*d1 = h)
            (s = d1*d2/2)
            (p = 4*b)
            (s = b*h)
            (2*Cos[a1/2]*b = d2)  )
```

Abstraction begins to fail to be useful when objects get too many asymmetries. The RECTANGLE provides no global unit, or rather two units,

```
            (b1 + b2) and (b1*b2)
```

The RHOMBUS incorporates many non-symmetrical concepts. It is completely determined by its diagonals, d1 and d2, but not by its angles, a1 and a2.

```
SECTOR:
      (r RADIUS;
       a ANGLE;
       s AREA;
       alternatives: in CIRCLE;
            (r = in.r)
            (s = in.s*a/360) )

SEGMENT:
      (arc NUMERIC;
       chord NUMERIC;
       a ANGLE;
       s AREA;
       in CIRCLE;
            (arc = a*in.p/360)
            (s = (1/2)*in.r^2*(a*Pi/180 - Sin[a]))
            (h = in.r - (1/2)*Sqrt[4*in.r^2 - chord^2])
            (chord = 2*Sqrt[2*chord*in.r - chord^2])  )
```

```
TRIANGLE:
      (b1,b2,b3 SIDE;
       a1,a2,a3 ANGLE;
       s AREA;
       p PERIMETER/2;                                      half perimeter
       r RADIUS;                                           of inscribing circle
       h1,h2,h3 HEIGHT;                                    of each side
            (a1 + a2 + a3 = 180)
            (b1/Sin[a1] = b2/Sin[a2] = b3/Sin[a3])    Theorum of Sines
            (b1^2 = b2^2 + b3^2 - 2*b2*b3*Cos[a1])    Theorem of Cosines
            (b2^2 = b3^2 + b1^2 - 2*b3*b1*Cos[a2])
            (b3^2 = b1^2 + b2^2 - 2*b1*b2*Cos[a33])
            (2*p = a + b + c)
            (s = Sqrt[p*(p - b1)*(p - b2)*(p - b3)])  Heron
            (s = b1*h1/2)
            (s = b2*h2/2)
            (s = b3*h3/2)  )
```

New defining relations are usually redundant, and are added only as an efficiency technique. They can be algebraically compiled out, but the remaining equations may not be simple. It's actually a usage issue; the internal form of the triangle should match the most common kinds of computational requests on it. This can be done algorithmically.

```
RIGHT-TRIANGLE:
      (x TRIANGLE;
            (a3 = 90)                 dereferencing a3 to x.a3 can be automatic
            (b3^2 = b1^2 + b2^2)
            (b3*Sin[a1] = b1)
            (b3*Sin[a2] = b2) )

ISOSCELES-TRIANGLE:
      (x TRIANGLE;
            (a1 = a2)
            (b1 = b2)
            (a3 = 180 - 2*a1) )

EQUILATERAL-TRIANGLE:
      (x TRIANGLE;
            (b1 = b2 = b3)
            (a1 = a2 = a3 = 60)  )
```

Alternatively,

```
EQUILATERAL-TRIANGLE: (x ISOSCELES-TRIANGLE;
                       (a1 = a3)
                       (b1 = b3) )
```

```
POLY-SIMILAR:
      (x1, x2 POLYGON;
       k,l NUMERIC;                        ratios of similarity
            (k*l = 1)
            (x1.a_ = x2.a_)                all the respective angles are equal
            (x1.b_ = l*x2.b_) )            all the respective sides are proportional
```