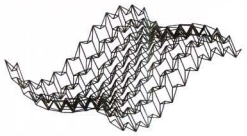
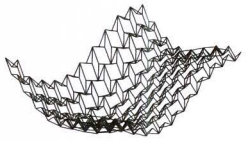
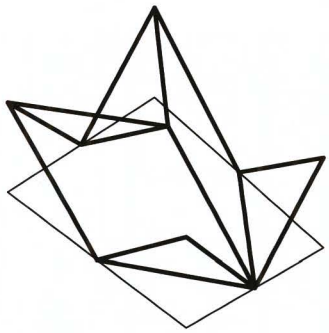
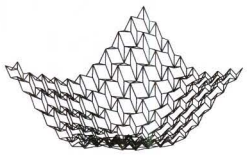


translational symmetry (symmetry group *p1*). Using two whole and four half parallelograms defines a module in which it is easy (or at least, easier) to relate the geometric boundary conditions to the proxy PLACE HOLDER.

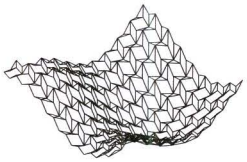


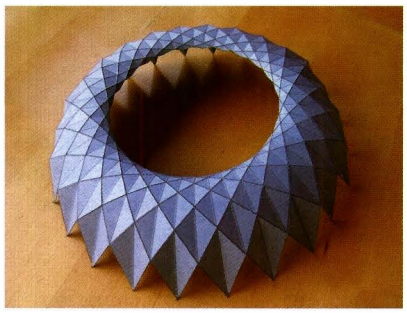


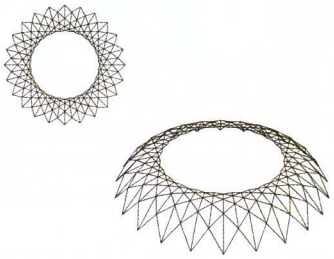


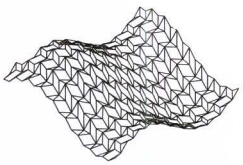
To connect adjacent modules requires coincidence along each edge and at each vertex at which modules join. Four edge points connect two modules each and four vertex points connect four modules each. The edge points are easy: they lie at edge midpoints on the PLACE HOLDER, so are guaranteed to coincide. Vertex point coincidence requires that, at each vertex, adjacent modules share a common vector from the vertex to the module point. Here, this vector is a global property of the surface. With slightly more work, the PLACE HOLDER object could hold individual direction vectors at each of its vertices.

A POINT COLLECTION – a rectangular point array by u and v parameters on a parametric surface – locates a collection of quadrilateral polygons. Using these quadrilaterals as PLACE HOLDERS, the modules cover the surface to look like origami. They aren't of course: on a general surface, edge lengths will differ from the initial paper and polygons will be non-planar. Adding constraints can allow true folded paper models. For instance, the GOAL SEEKER pattern can be used to find feasible configurations for folding models of Persian Rasmi domes from single sheets of paper (Maleki and Woodbury, 2008).









8.25: An approximation of folded paper geometrically attached to a surface.

8.24: A dome structure folded from a planar assembly of triangles. Source: Maryam Maleki

8.11 PROJECTION

Related Pattern • POINT COLLECTION • REPORTER • MAPPING

Source: Alexandre Duret-Lutz. Creative Commons Attribution Share-alike.

What. Produce a transformation of an object in another geometric context.

When. "Here" and "there" pervade design. Eyes, ears, the sun, lights, ducts, pipes, columns and beams all relate a "here" to some distant "there". Often a geometric line or curve provides the needed link. Use this pattern to construct coherent, reproducible relationships between "here" and "there".

Why. Projection is a simple, yet open-ended tool for producing new objects from old. Its origins lie in the Renaissance and before. For designers, it is most associated with the field of *descriptive geometry*, an 18th-Century invention (Gaspard Monge, 1827) and one which, until recently, was a mandatory part of design curricula worldwide. Descriptive geometry codified procedures for deriving two-dimensional drawings of three-dimensional objects by projecting the three-dimensional objects onto surfaces. Parametric modeling supports a much richer collection of projective ideas than was practical with older, manual techniques. With tongue somewhat in cheek, one could argue that parametric modeling is the 21st-Century replacement for descriptive geometry. The idea of projection has three parts: (1) a *source object* to be projected, (2) a *projector* or *projection method*, and (3) a *receiver*, the object on which the source object's projection appears. Its most simple form is orthogonal projection: points are projected onto a receiving plane such that the projection lines are perpendicular

to the plane. The projection and intersection tools common in most parametric modeling systems enable a wide range of projective form-making ideas. The two main effects of projection are indirection and separation. With it, a model can be the indirect cause of a sculptural effect. With it, different object aspects can be separated into distinct views that may enable special views and inferences on the object. A very common example is a light (essentially a point source) that *projects* through a patterned screen onto a surface.

How. Every PROJECTION has the three above parts: (1) the projected object, (2) the projection method and (3) the receiving object. The projected object is a point or any composite of points: a line, ray, line segment, curve, polygon, surface, or 3D object. The three most common projection methods are *parallel projection* in which all projecting rays are parallel; *normal projection* in which the projecting rays are normal to the receiving object; and *perspective projection* in which all projecting rays pass through a single point. There are a wide range of other methods. For instance, cartographic projections can be explained as the mapping of parametric coordinates from one surface to another.

The common receiving objects are planes, polygons, surfaces, lines and curves, as well as composites of these. While possible, PROJECTIONS to points and 3D objects seem to be less common in practice.

A wide variety of projections exist (Anderson, 2009). Computing projections typically involves either mathematical projection or geometric intersection. Mathematical projection provides direct solutions to relatively simple cases such as projecting one vector $\vec{u}$ onto another vector $\vec{v}$ with result $\vec{w} = \frac{\vec{u} \cdot \vec{v}}{|\vec{v}|} \vec{v}$. More complex situations involve intersecting objects. For example, projecting a point onto a surface amounts to computing the intersection between the surface and projecting ray.

For simple cases, a parametric modeler will provide direct tools for computing projections, for instance, projecting a line onto a plane. It is a fact of life though that designers will push these bounds. In these more complex situations, using the PROJECTION pattern involves three steps: (1) sampling key object points, (2) projecting these points onto the receiver, and (3) reconstructing the object as projected on the receiver.

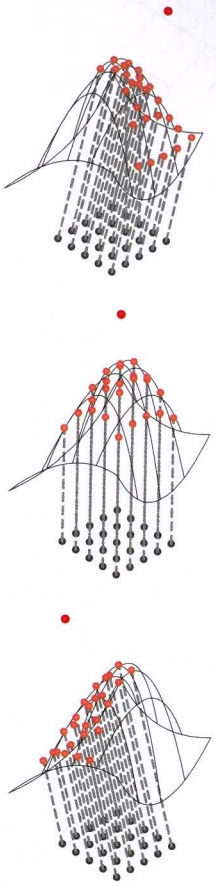
PROJECTION Samples

Surface Sampler

When. Project a collection of points onto a surface.

How. The mathematics of parametric surfaces ties both the surface shape and its uv-parameterization to the control polygon. Often, only part of a surface is actually needed in a design. Projecting a POINT COLLECTION onto the surface makes a subset of the surface with its own independent parameterization.

The source object is a point collection. In this sample, the collection lies on a plane and is a simple array, but other geometric and data arrangements can be used. The projector is parallel projection, with projecting rays being parallel to a line from the centroid of the collection to a controlling point in space. As an alternative, the projector could be a perpendicular line from the source plane and a control could allow the source to be moved within its plane. The receiver is the surface.

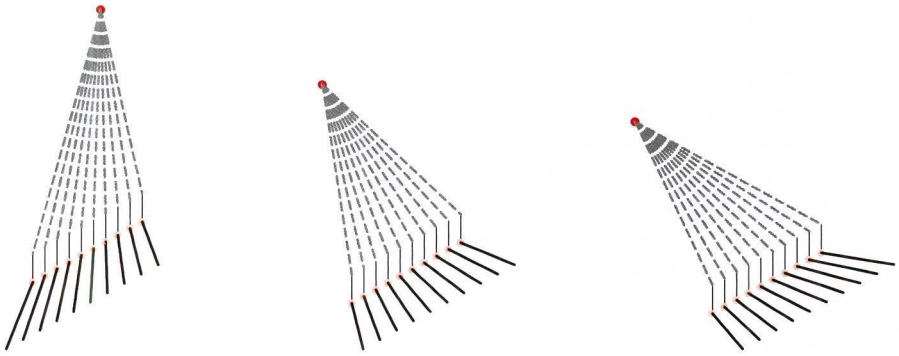


8.26: The planar array of points projects to the surface. The geometry is that of the surface, the data organization that of the array.

Shadows

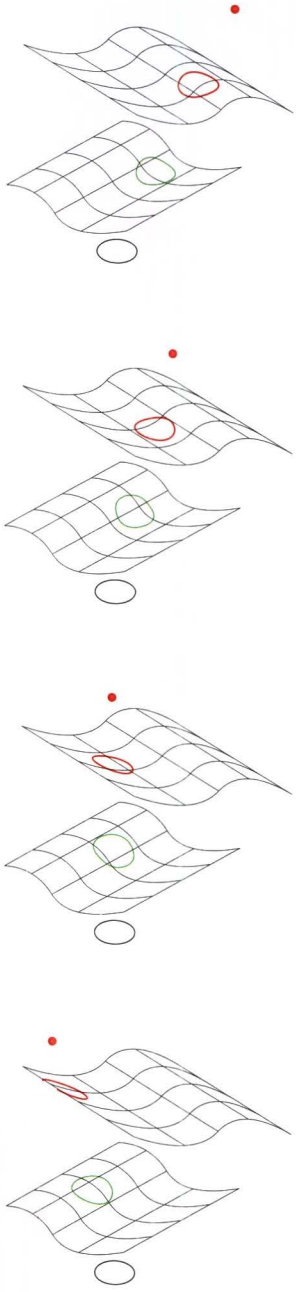
When. Simulate a row of posts casting shadows on the ground.

How. Start from a line (abstracting a post) standing vertically on the xy-plane. Define a free point as the moving light. The *shadow point* is the projection of the free point onto the xy-plane. The shadow is a line between the base of the post and its shadow point. Replicating the startpoint of the posts gives a row of posts, each with its own shadow. In this case, the source is the free point, the projection is a perspective projection through the source and the receiver is the xy-plane.

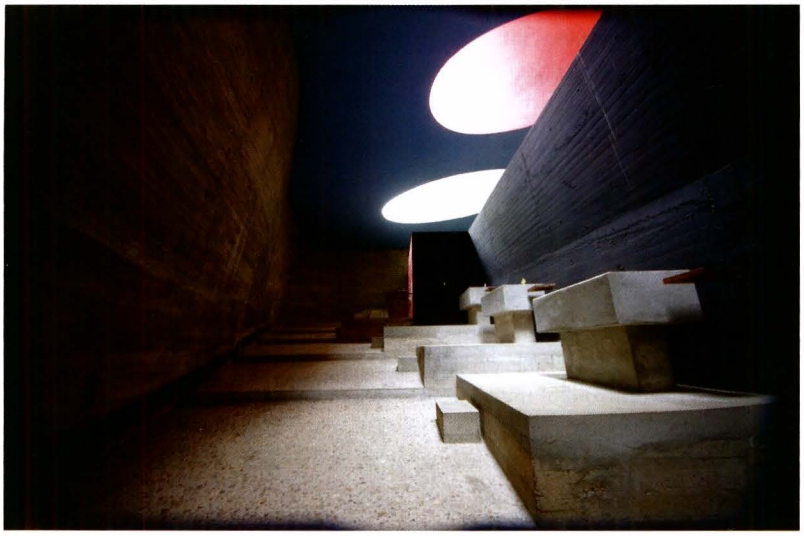


8.27: This very simple sample illustrates the basic idea of projecting a source to a receiver using a method. In this case the method is simple perspective projection (all rays pass through a point).

Skylight



8.28: The point in space controls the location of a parallel projection of a circle through the two surfaces.



Source: Pieter Morlion

When. Create a daylight "lens" that focuses on a circle.

How. Two free-form surfaces represent a roof and a ceiling. The xy-plane is the floor. A circle on the floor can be daylit by projecting it through the ceiling and roof surfaces. The direction of daylight is nearly uniform, but the two separated holes will act as a very fuzzy lens to focus daylight on the circle. Similar to the *Surface Sampler* sample, the projection direction is controlled by a free point. If the direction is constrained to be within the sun's annual range, for a specific model instance the sun will shine directly on the circle exactly twice a year. If the direction is chosen to lie on either of the two solstice paths, this reduces to exactly once per year. Fixed architecture can have difficulty in responding to moving phenomena.

The projection of a circle onto a surface, or even an angled plane, is no longer a circle. While some parametric modelers provide curve-onto-surface projection tools, a good approximation can be had by projecting sampled circle points and reconstructing the curve from the projected points. The resulting curve will not exactly coincide with the surface in which it should lie. Alternatively, if the modeler has surface trimming tools, trimming the surface with a sweep of the circle along the projection line will yield a new surface with a hole.

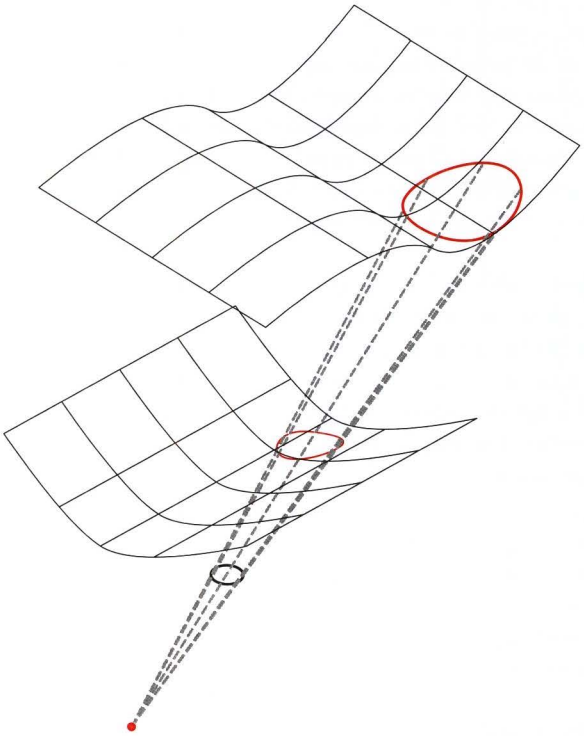
When rotating the model, you can see that three circles perfectly coincide at a specific viewing angle (in a parallel viewing projection).

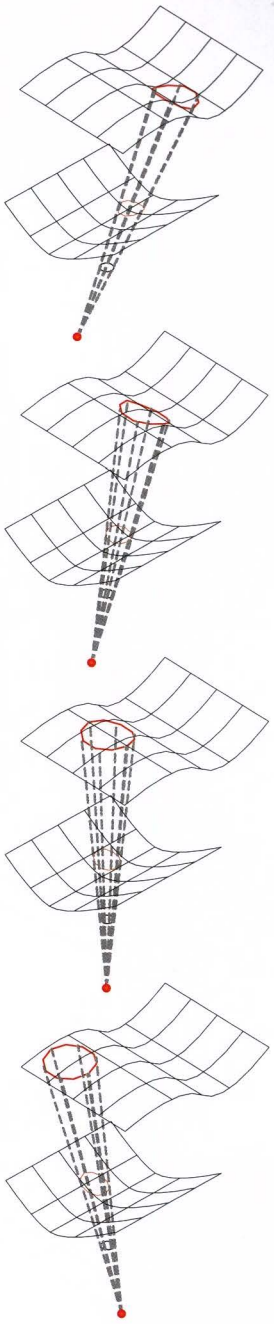
A famous example of projection used in form-making is Le Corbusier's 1953 *Monastery of Sainte-Marie de La Tourette* in France (shown in the image above).

Spotlight

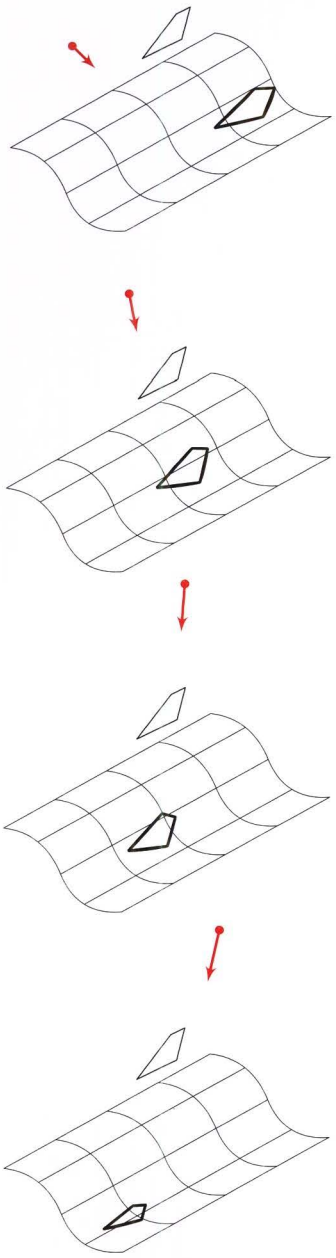
When. Model a metaphorical spotlight projecting a circle onto several surfaces.

How. This sample is very similar to the previous one. The main difference is that all the projecting rays intersect at the light point. Each projecting ray starts from the light point and goes through the sampled points of the base circle. While rotating the model in a parallel projection view, you can see the projected circles do not coincide at any angle. If you use a perspective view the projected circles coincide when the camera and light source coincide.





8.29: Projection through a point creates a cone of intersection through the surfaces. Two objects: (1) a projection point and (2) a circular "lens", control the projecting cone. In this case, the circle is parallel to the xy-plane – giving an ellipse as the result.



8.30: The boundary of the polygon projects onto the surface. The red vector controls the direction of projection. Even when the polygon has straight sides, the projection will be curved on the surface.

Solar Polygon Shadow

When. The shadow of a polygon cast by the sun on a curved surface.

How. The source is a polygon, the receiver a free-form surface. The projector is the sun, therefore the projecting rays are parallel.

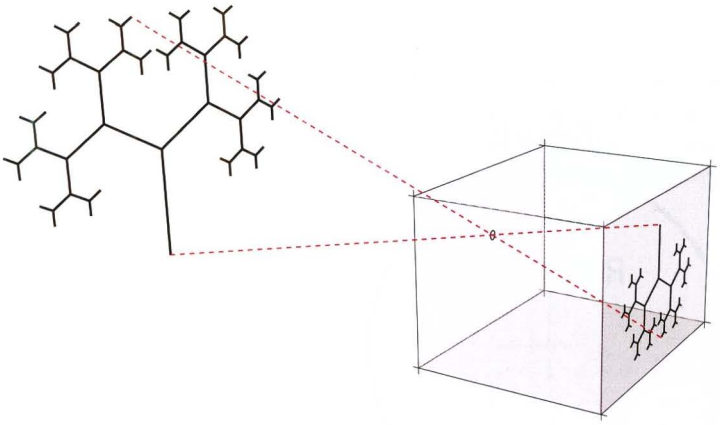
Straight lines project as curves onto a surface. For specific surface types, such as conic sections, closed-form equations for these curves exist. For free-form surfaces, approximations must suffice. Even though your favourite parametric modeler may have a curve projection tool, approximation techniques remain important tools in a modeler's kit. The key is sampling. Sample each source line with a sequence of points. The choice of how many points depends on the complexity of the receiving surface: high curvature and rapidly changing surface normals require more samples. Project the sampled points onto the surface and reconstruct a curve "on" the surface from the sampled points. The word "on" is advisory – the curve will not lie exactly on the surface. Much representation is approximation. If the curve and surface must exactly coincide, either sample very densely or find a modeler that supports exact curve-to-surface projection.

It is clear that the shadow is no longer bounded by four straight lines, but by four curves. Note too a further simplifying assumption. Non-planar polygons can be thought of as defining a minimal surface. If the source polygon is non-planar then its orientation must be such that no part of this minimal surface projects outside of the projection of the polygon's edges. Else, the shadow will not model reality. This may be good enough – again, much representation in design is approximate.

Pinhole Camera

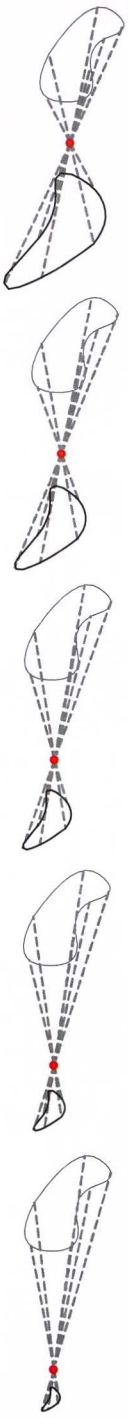
When. Model a pinhole camera.

How. A pinhole camera replaces a conventional glass lens with a tiny hole. Such a hole in very thin material can focus light by confining all rays from a scene through a what is effectively a single point. To produce a reasonably clear image, the aperture diameter has to be less than about 1/100 of the distance between the pinhole and the screen.



The principle of a pinhole camera is that light rays from an object pass through a small hole to form an image on the screen (shown in image above). To model this effect, simply place a point (modeling the pinhole) between the source and receiver and project from the source through the pinhole to the receiver. Use either direct model reconstruction or the sampling technique from the *Solar Polygon Shadow* sample above to reconstruct the source on the receiver.

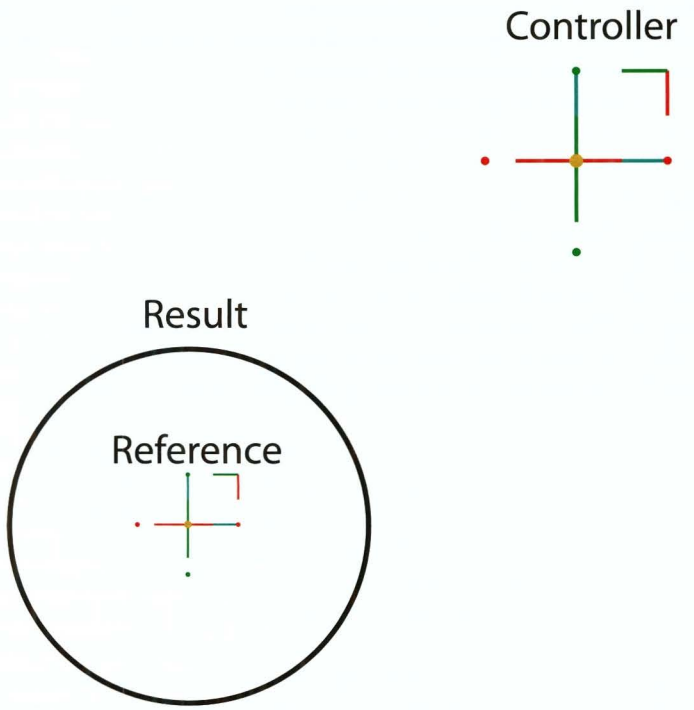
Note that the image is reflected both top-to-bottom and left-to-right. This is equivalent to a 180° rotation about the axis normal to the receiving plane and through the pinhole (providing the receiver is a plane).



8.31: Putting the projection point between the source and destination objects produces a pinhole camera.

8.12 REACTOR

Related Pattern • CONTROLLER • GOAL SEEKER



What. Make an object respond to the proximity of another object.

When. The essence of parametric modeling is expressing object properties in terms of the properties of upstream objects. A problem arises when the relation between the object and its upstream precedents is based on proximity. The new location for the object becomes based on the old location for the object, making the object definition become circular! Propagation graphs cannot have cycles.

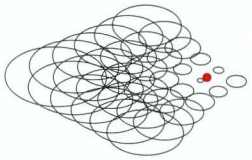
Use this pattern when you want to make an object respond to the presence of another object.

Why. Designers often use the metaphor of *response* in which one part of a design depends upon the state of another. Reversing the perspective, is as if part of a design becomes a tool for shaping the other. This situation is very much like that encountered in the CONTROLLER pattern, but with a key difference – the controlling property is proximity.

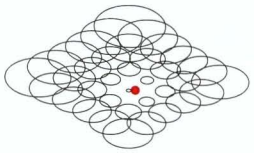
How. The essential idea: connect an *interactor* to a *result* through a *reference*.

The trick is to join the *interactor* and the *result* through a mediating and usually fixed object, which we call a *reference*. The *interactor* and the *reference* interact to produce the *result*.

For example (see sample *Circle Radii and Point Interactor* below), you have a point and a circle, and you want the circle to get bigger (or smaller or elliptical) as you move the point closer. This can be done by using the REACTOR pattern. As you might have guessed, the point is an interactor and the circle is a result (which can be replicated to give us an array of circles). The circles have to be somewhere. This somewhere is the reference. The reference is usually hidden in a REACTOR.



Position is a complex property that can manifest in many ways. Position is any combination of location and direction, for instance, the length or direction of lines, parameter or number or position of points, direction of planes, radius of circles and even additional properties such as colour if they are made to depend on position.

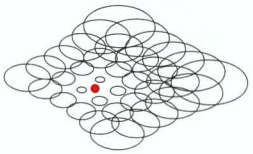


REACTOR Samples

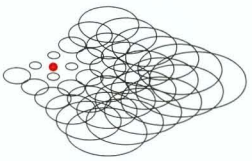
Circle Radii and Point Interactor

When. Control the size of a set of circles by proximity to a point.

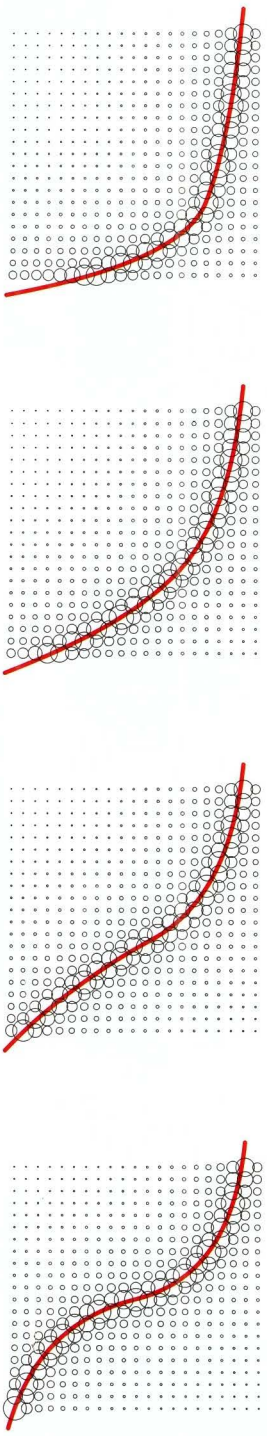
How. Perhaps the simplest definition of a circle requires just its *centre* and *radius*. The *centre* (a point) is the reference and the *radius* is the result. The free point is the interactor. Making the *radius* a function of the distance between the *centre* and the controlling point completes this instance of the REACTOR pattern. In this case, the function is a direct relationship – its value shrinks as distance shrinks. As the interactor moves closer to the circle, the circle gets smaller.



Replicating the reference causes all of the circles to react to the movement of the interactor. Hiding the reference creates an illusion of direct control from interactor to result.



8.32: The collection of circles reacts to the prescence of an interactor point.



8.33: In this sample the *interactor* is an entire curve, itself controlled by a set of points.

Circle Radii and Curve Interactor

When. Control the size of a set of circles by proximity to a curve.

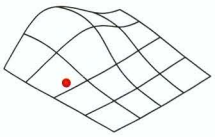
How. In modeling terms, this sample hardly differs from the previous one. In both, the radii of circles in an array change with proximity to an interactor. The only differences are that the interactor here is a curve and the radii grow with proximity rather than shrink. The distance from a reference point to the curve is the distance between the point and its projection onto the curve – this is the shortest distance between the point and curve.

Hiding the reference removes each circle's visual fixed point. The eye focuses only on the changing displayed part.

Source: NASA Earth Observatory image created by Jesse Allen, using Landsat data provided by the University of Maryland's Global Land Cover Facility.

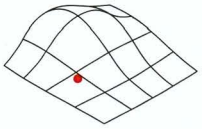
Lift

When. Make a line's length increase as you move a point closer to its start point.

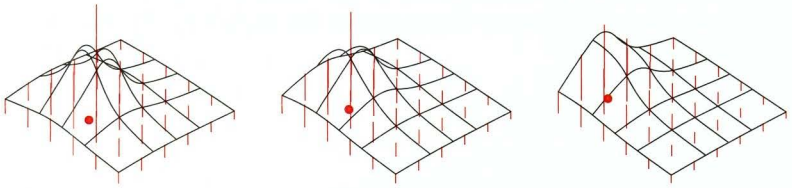


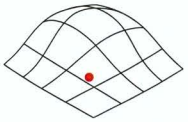
How. Define two points; call one the reference and one the interactor. On the reference define a vertical line: the result.

The length l of the result must increase as the distance between the reference and the interactor shrinks. Choosing a "good" function for l takes work. In this sample $l = 1/(\text{distance}(\textit{interactor}, \textit{reference}) + 0.1)$. The small amount of 0.1 that is added within the function prevents the line from becoming infinitely long as the distance between the points approaches zero. The MAPPING pattern gives a process for reliably using other functions.



Replicate the start point and hide it. Now you have a set of lines that react to the movement of the interactor. In turn, use the line endpoints to define the shape of another object using the lines, such as a roof surface.

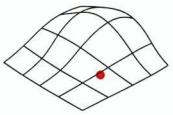




8.34: With a reactor a single point can replace the 16 points needed for general control of a surface. Of course, generality is lost; some surfaces cannot be modeled.

Repeller

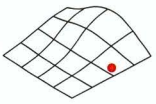
When. Make a point move away from a controlling point.



How. Define two points; call one the reference and one the interactor. The result will lie on the infinite line defined by the reference and the interactor.

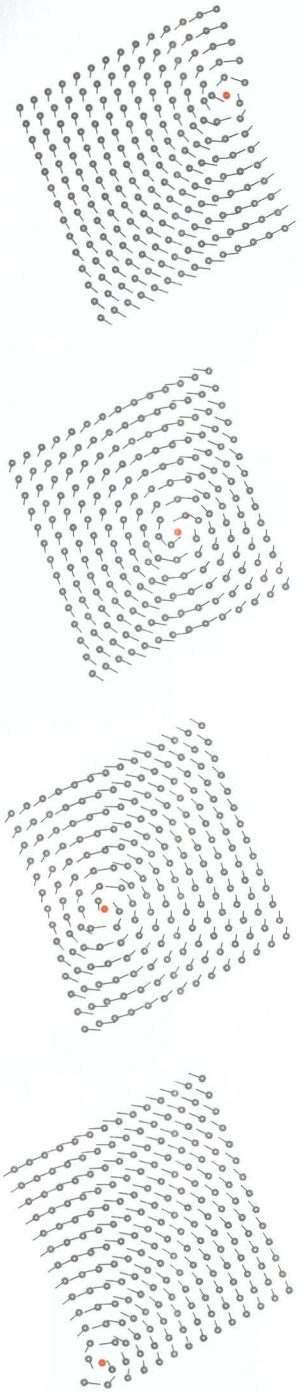
The result is the sum of the reference and a vector. The vector's direction is the same as the vector between the interactor and the reference. Its length results from a function of the distance between the interactor and the reference: as the interactor moves towards the reference, the length increases.

The function in this sample is $SD/(\text{distance}(\textit{interactor}, \textit{reference}) + SD * 0.01)$, where SD stands for *Standard Distance*. As SD grows, so does the distance over which the pattern has an effect. The small quantity of $SD * 0.01$ added to the distance prevents the result from moving infinitely as the interactor approaches the reference.



8.35: The only effective difference between this and the previous sample is that the line on which the result lies is directly defined by the interactor and reference.

Replicate and hide the reference. The result points now appear to respond to the interactor. In turn, use the result to define other objects, such as a surface.



8.36: The point appears to directly control the vector field. Hidden, as with most REACTORS, is the reference, in this case, a point and a frame located at the point.

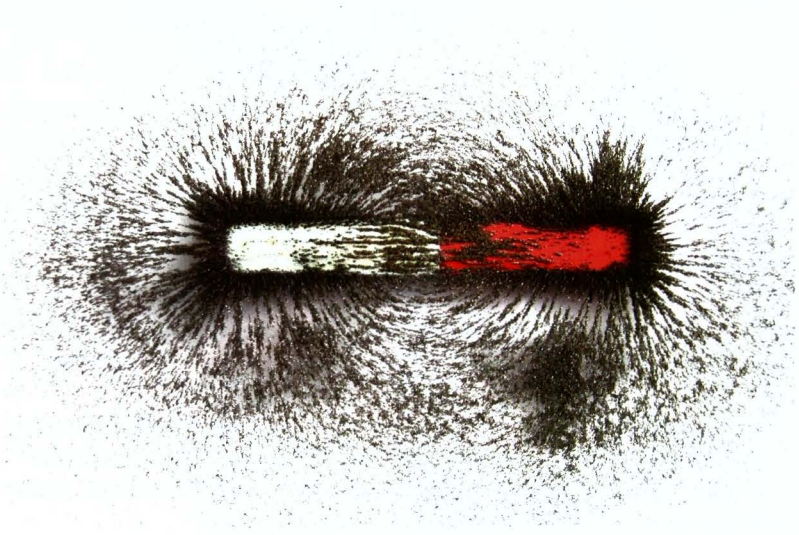
Vector Field

When. Rotate a bound vector as a controlling point moves, so that it always has the same angle to the point. Replicate to define a vector field.

How. Define two points: an interactor and a reference. The goal is to define a vector bound to the reference such that it is right-handedly perpendicular to the imaginary line that connects the two points. Create a frame on the reference point by using the interactor point to locate its x-axis. Making the result vector on the y-axis of this frame ensures that it will always be perpendicular to the x-axis and consequently to the connecting line.

Note that the reference can be more than one object. In this case both the start point of the vector and the frame comprise the reference. The frame and start point could combine, simplifying back to a single reference. Such reduction is not always possible.

Replicate the reference point in one, two or three dimensions and hide it. All of the result vectors will react to the position of the interactor and portray a continuous vector field.



Source: Dayna Mason

Dimple

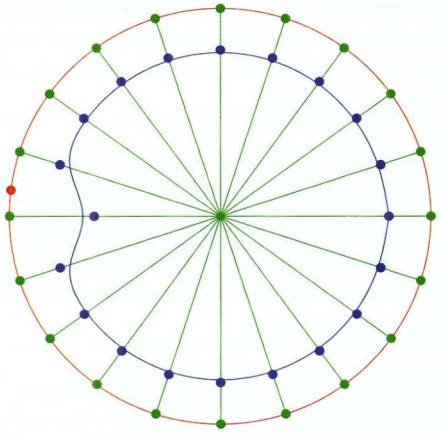
When. Make the local shape of a closed curve react to a nearby point.

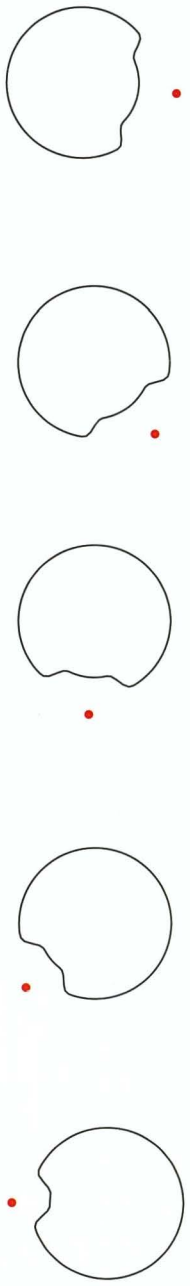
How. Define a circle and two points on the circle. Call one of these points the reference and the other the interactor. Draw a line from the reference point to the circle's centre. Place the result, a parametric point on this line, that moves toward the center if the interactor gets *too close* to it. The trick is to assign the parameter of the result point a higher value (here 0.4) if the distance between the interactor and the reference (measured by the modular distance between their parameters) becomes less than a value d (explained later), otherwise set it to another value (here 0.2). This distance condition can be defined as follows:

```
function modular01Distance (double t0, double t1)
{
 object result = t0-t1;
 return
  result > 0.5 ?
   1 - result :
   result >= 0 ?
    result :
    result > -0.5 ?
     Abs(result) :
     result + 1.0;
}
```

The parameter d here can be a number less than or equal to 0.5 and greater than or equal to half of the distance between each two references. The lower limit is needed so that the test is always true for at least one point. With *count* equally distributed references, a minimal value is $d = 1/(2 * count)$.

After replicating the reference, create a closed curve interpolating the result points. This curve will look like a circle deformed by the interactor.

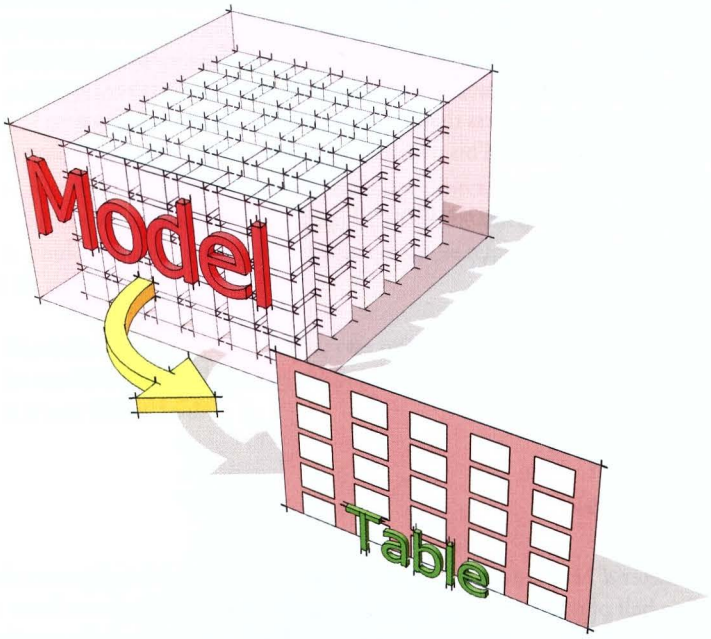




8.37: A simple interaction hides a complex mediating reference structure. The point appears to directly control a dimple on the circle.

8.13 REPORTER

Related Pattern • CONTROLLER • PROJECTION • SELECTOR



What. Re-present (abstract or transform) information from a model.

When. Models can be complex. Finding and using the relevant parts of a model can be tedious and error-prone. Further, some "parts" of a model may only be implied – computation may be needed to construct them from primary model data. Use this pattern when you need to use some aspect of a model in another process or another part of the model.

Why. Models can be complex and hard to understand. They can express far more information than they directly contain. Such implicit information must be uncovered through functions applied to the model. Using a REPORTER allows you to present only the information you need to another part of the model. This makes your model structure more clear and helps you work with other people who may use your model.

REPORTERS may abstract (simplify) or transform (re-present). They report a design or its parts from a different point of view. In analogy to a relational database, the REPORTER pattern is akin to a view table extracted from a database.

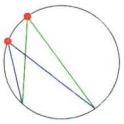
How. Data from a REPORTER must be conceived, extracted and envisioned. Deciding what to report requires judgment, for example, when reporting façade element planarity, the most effective report may, or may not, be the minimal vertex movement that restores planarity. Extracting the data may demand a complex algorithm, for example, a convex hull of a set of points. Envisioning that data so that it makes sense to the person receiving it has been the subject of entire books (Tufte) (1986; 1990; 1997) – it is likely that a simple, textual list will not be best. Of course, if the purpose of the REPORTER is to provide data to another program, a textual or numeric list might be exactly right.

You can use the REPORTER pattern in many different ways.

- Displaying properties of an object. For example, a collection of point coordinates might be displayed numerically in a table, or their minima and maxima might be instanced as two points.
- Defining an element in a different way. For example, a point defined in one frame can be reported in another.
- Conditionally selecting parts from your model. In this use, a REPORTER is very similar to a SELECTOR. For example, a roof surface comprising polygons might be reported by the degree of non-planarity in each.
- Creating new objects from reported objects. In this case of indirection, a property of an object is used to define another object. For example, a REPORTER on line segments might comprise new segments coincident with the originals' central third. Another REPORTER example produces a *dual* polygon mesh (the mesh generated by replacing centroids with vertices, and edges between vertices with edges between centroids).
- Sampling a model and then reporting this simplified version somewhere else and with different conditions to create a more complex model.
- Copying. Generally, copying is reporting the model as many times as needed in different places, therefore features such as copy, mirror, clone and rotate are all kinds of reporting.

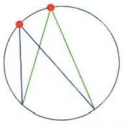
The REPORTER pattern typically combines with other patterns. This pattern feeds information to downstream objects or directly to the designer. In many ways, it is a normal part of parametric modeling in that models are defined in terms of other models. The difference is that a REPORTER is often not a part of the design, but rather a view on the design or an intermediary in the model construction process.

In some sense, the REPORTER pattern is an anti-CONTROLLER pattern. In a CONTROLLER, information flows from the control to the target – typically from a simple model to a complex one. In a REPORTER information flows the other way – it typically is an abstraction of a larger model.



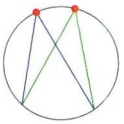
$\alpha = 39.01$

$\beta = 39.01$



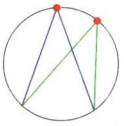
$\alpha = 39.01$

$\beta = 39.01$



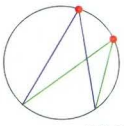
$\alpha = 39.01$

$\beta = 39.01$



$\alpha = 39.01$

$\beta = 39.01$



$\alpha = 39.01$

$\beta = 39.01$

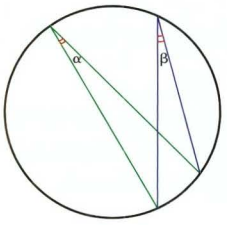
8.38: Angles subtended by a common chord are the same. Conversely two points subtending the same angle from a line segment lie on a common circle for which the line segment is a chord.

REPORTER Samples

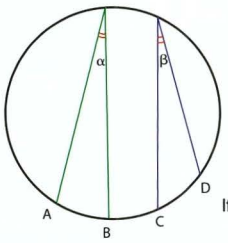
Subtended Angle

When. Envision information from the model as text.

How. In a circle, any inscribed angle subtended by a chord of a given length is constant. A good demonstration juxtaposes as text a suite of angles subtended by a common chord. This model comprises a circle and two pairs of lines that represent two angles and share a common subtending chord. Showing both of those angles together demonstrates this simple geometric theorem. Text is the simplest REPORTER. Sometimes it is actually effective.



$\alpha = \beta$

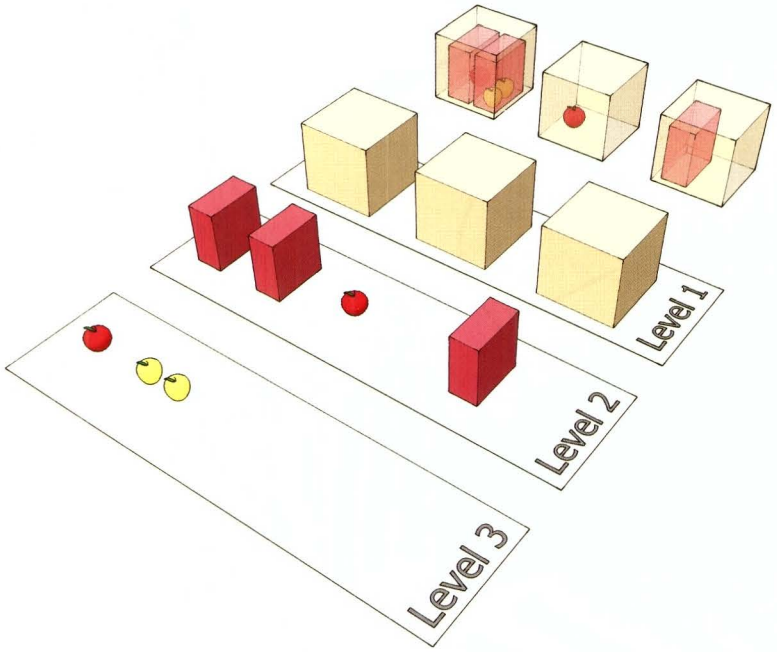


If AB = CD then $\alpha = \beta$

Array Depth

When. Extract properties of a collection of points organized as an array.

How. Collections of point-like objects define many properties, some pertaining the points themselves and some to their organization in a collection. Examples include the extremal coordinates of the points, the longest path in the collection (the array depth) and the number of elements in the collection. In this case, the REPORTER pattern extracts such data from the model. A function iterates over each element in a collection, accumulating the desired measures as it proceeds.

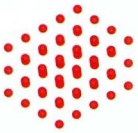


Rank = 3

Dimensions = { 5 ,6 ,2}

$BBox_{ll} = \{0.0, 0.0, 0.0\}$

$BBox_{ur} = \{4.0, 5.0, 1.0\}$



Rank = 3

Dimensions = { 4 ,5 ,3}

$BBox_{ll} = \{0.0, 0.0, 0.0\}$

$BBox_{ur} = \{3.0, 4.0, 2.0\}$

Rank = 3

Dimensions = { 3 ,4 ,4}

$BBox_{ll} = \{0.0, 0.0, 0.0\}$

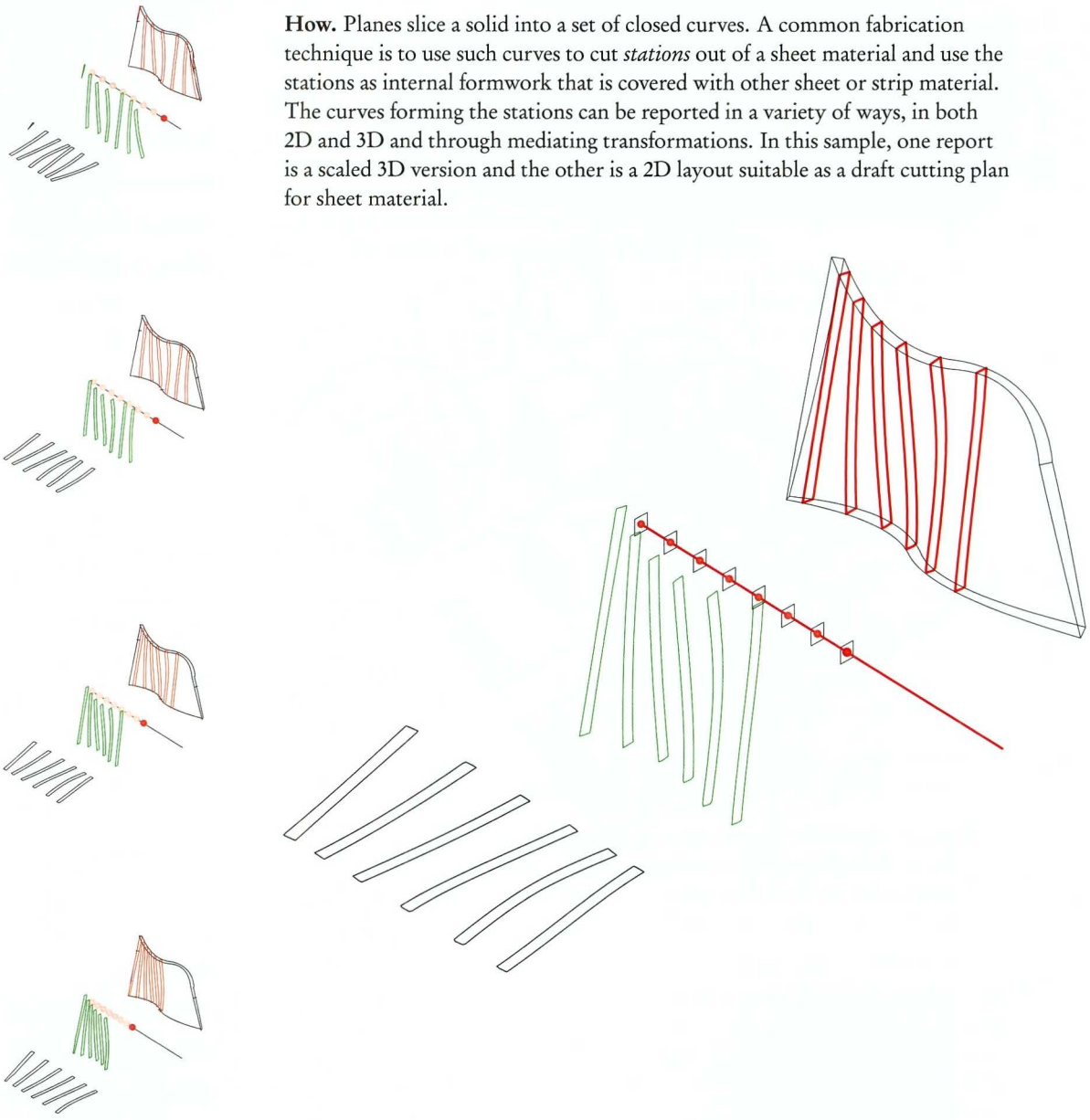
$BBox_{ur} = \{2.0, 3.0, 3.0\}$

8.39: Arrays are complex objects. Understanding their structure can be hard. This REPORTER puts a display of array structure directly in the three-dimensional model.

Fabrication

When. Transform design data for fabrication.

How. Planes slice a solid into a set of closed curves. A common fabrication technique is to use such curves to cut *stations* out of a sheet material and use the stations as internal formwork that is covered with other sheet or strip material. The curves forming the stations can be reported in a variety of ways, in both 2D and 3D and through mediating transformations. In this sample, one report is a scaled 3D version and the other is a 2D layout suitable as a draft cutting plan for sheet material.

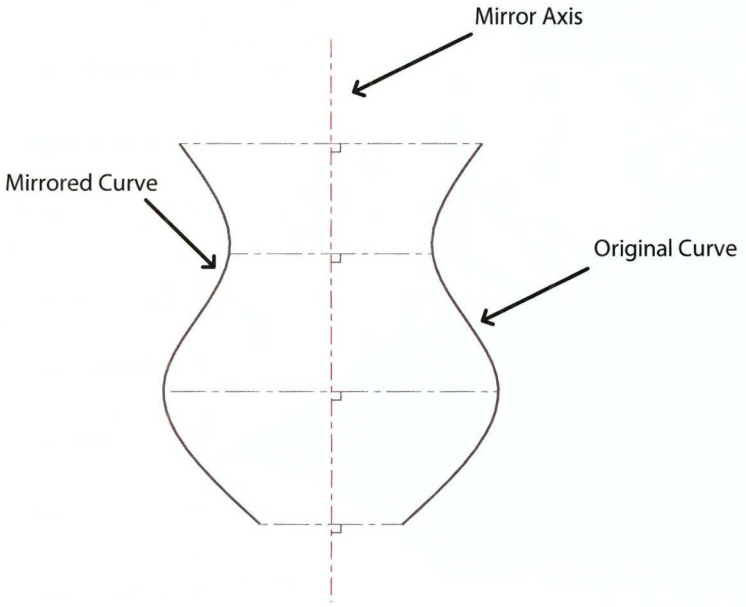


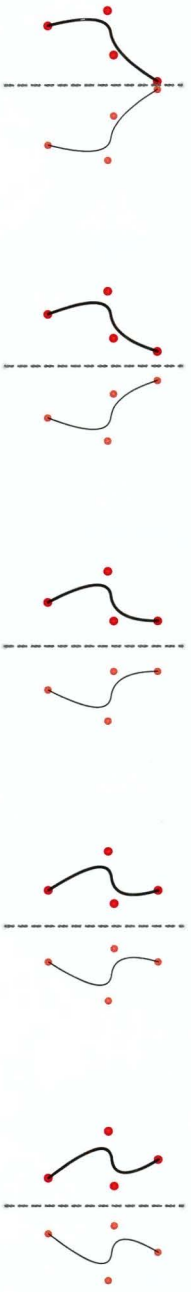
8.40: The model reports data for fabrication. Parametric systems typically provide such outputs. The principle though is simple: transform parts of the model into a separate view or location.

Mirror

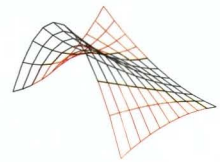
When. Mirror a curve through an axis.

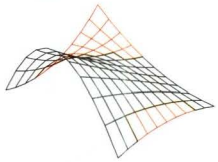
How. Define a curve based on a point collection, and a line to use as a mirror axis. In 3D the line would be a plane. Now reflect the curve poles through the axis to create poles for a new curve. Finally create the new curve from the new poles; this curve is a mirror of the first curve. In general, mirrored objects are *enantiomorphs*, that is, identical except for handedness.

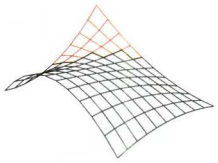


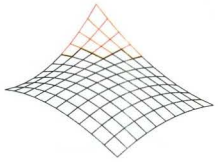


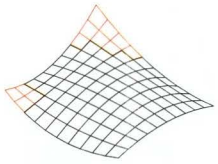
8.41: A mirror can be seen as either a REPORTER or a PROJECTOR.







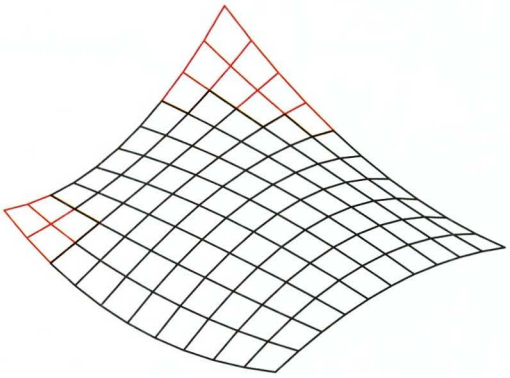




Out of Plane

When. Report the out-of-plane polygons of a surface both by colour and text.

How. Create a curved surface and subdivide it with polygons. Some polygons may be non-planar. The amount of non-planarity depends on the local surface. Iterate over the collection, extracting the polygons that exceed a threshold out-of-plane measure. Visualize the resulting polygons both *in situ* with colour and in a tabular report.



The out-of-plane polygons are:

1 . polygon01[0][0]

2 . polygon01[0][1]

3 . polygon01[0][2]

4 . polygon01[0][8]

5 . polygon01[0][9]

6 . polygon01[1][0]

7 . polygon01[1][1]

8 . polygon01[1][8]

9 . polygon01[1][9]

10 . polygon01[2][0]

11 . polygon01[2][1]

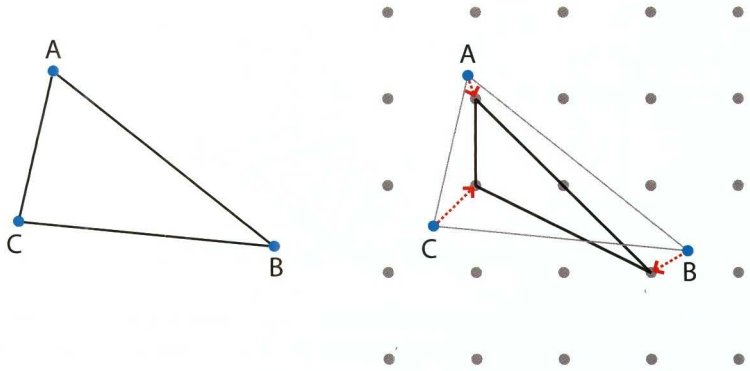
12 . polygon01[3][0]

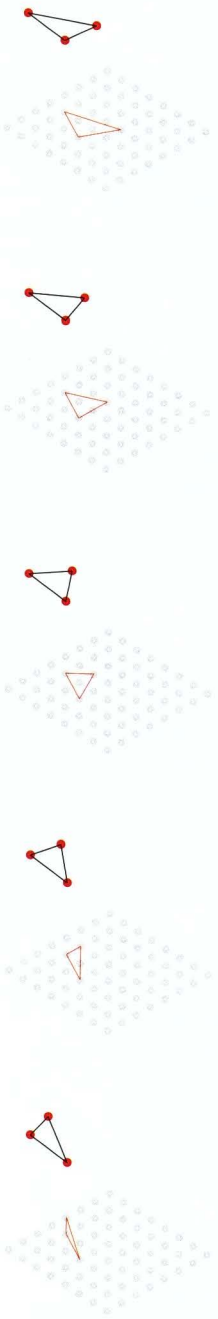
8.42: Understanding and controlling anomalies is both enabled and produced by parametric modeling. REPORTING directly in the model can provide much more effective feedback than a table of text.

Snapper

When. Report a triangle snapped onto a grid.

How. Parametric "sketches" (yes, a parametric model can be sketch-like!) are continuous. They may be abstracted into a discrete system, such as a grid. This REPORTER maintains the original model and ability to smoothly interact with it and reports the model as it would appear were it snapped onto a specified grid. The reporting takes two steps. First, report the original triangle's vertices in a new frame. Second, select and report the nearest point on the grid to each triangle vertex. Construct the reported triangle on these abstracted points.





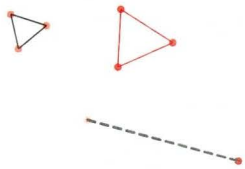
8.43: This REPORTER builds the usual system-level snapping interaction directly into a model.

Triangle

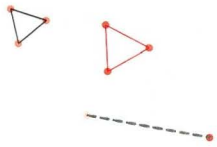
When. Rotate and scale a triangle.

How. This sample separates the shape of an object from the space in which it is embedded, giving independent control of each. It reports a triangle developed in one frame (the baseCS) in another system (the reporterCS) and also provides rotation and scaling controls for that new frame.

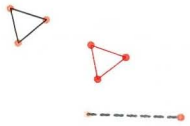
Ironically, this REPORTER uses a REPORTER internally. In order to compute the rotation of the reporterCS system, a point is defined in the baseCS system at the origin of the reporterCS system, but one unit along the x-axis of the baseCS system. Reporting this point in the reporterCS system provides the arguments needed for the function Atan2 to fully compute the rotation angle.



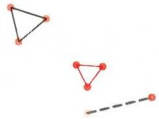
Scale: 1.63
Rotation: 15.35°



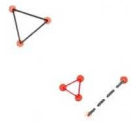
Scale: 1.3
Rotation: 10.33°



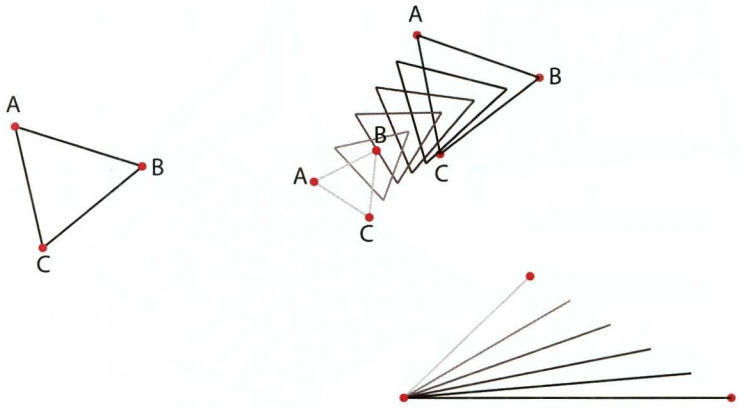
Scale: 0.98
Rotation: 1.91°



Scale: 0.7
Rotation: 13.89°



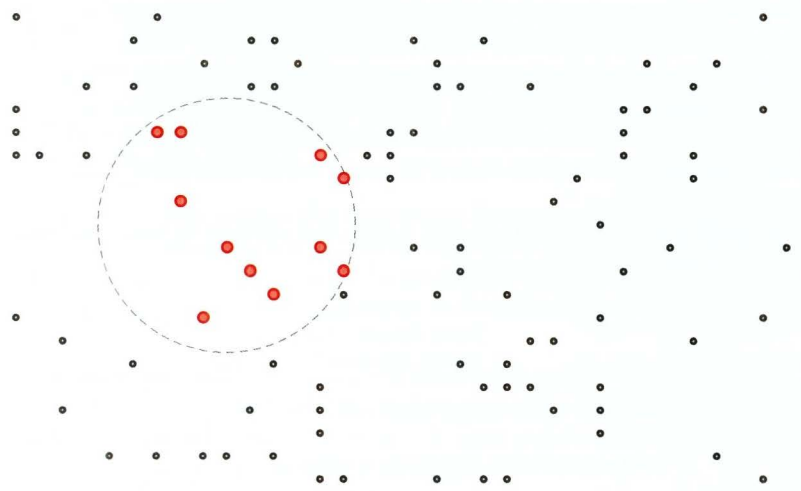
Scale: 0.53
Rotation: 44.29°



8.44: This REPORTER separates local editing (of the triangle) from its actual location in space.

8.14 SELECTOR

Related Pattern • CONTROLLER • REPORTER



What. Select members of a collection that have specified properties.

When. Selection is a universal in interactive systems. In parametric systems, it can be part of the model itself. At each update of the propagation graph, objects can select the arguments to their update methods. Use this pattern when you want to locally and dynamically restructure a model depending its state.

Why. Creating objects and using objects are different acts. You may specify, say, a set of points by giving their Cartesian coordinates. When using these points, you might be interested only in those that are close to a line. The SELECTOR pattern allows you to separate object creation and later use, and express these two common operations in the terms most suitable for each.

How. In the SELECTOR pattern, there is always a collection of given objects and a collection that is the outcome of the SELECTOR'S action. We call the first list the *target* and the second list the *result*. The SELECTOR mediates between these by determining which elements of the target to include in the result. The properties that objects must have or conditions they must meet in order to be selected we call the SELECTOR'S *behaviour*.

For example (see the sample *Distance between Points* below), the target may be a list of points and the SELECTOR a compound of a point, circle and function. The point specifies the location of the SELECTOR; the circle the distance within which selection will occur; and the function how points will be selected and how selected points will be constructed, that is, the SELECTOR'S behaviour. The function must select those points whose distance to the SELECTOR point

is less than (or perhaps greater than or approximately equal to or within some range of...) the parameter *d* of the SELECTOR'S behaviour.

The simplest representation of a SELECTOR'S behaviour is a function treating each of the target points individually. For each target, it computes the behaviour and returns a copy of any objects that conform. For instance, with selection by distance, it compares the distance between the target and the SELECTOR against the threshold *d*. If the target object satisfies the condition, the function creates a coincident point and returns this point (acting in a sense as a REPORTER). As a result, the function's (and SELECTOR'S) output will be a new list of points.

The result is not a subset of the target. Rather, it is a new set of points, identical to the selected points in the target.

```
function selectByDistance(Point selector,
                          Point target,
                          double distance)
{
 for (value i = 0; i < target.Count; ++i)
  {
   if (Distance(selector,target[i]) < distance)
    {
     Point result= new Point(this);
     result.ByCartesianCoordinates(baseCS,
                                   target[i].X,
                                   target[i].Y,
                                   target[i].Z);
    }
  }
};
```

An actual call to the SELECTOR'S behaviour function would take the following form:

```
selectByDistance(selector,target,distance);
```

where selector and target are points (or point collections) in the model and distance is a value held in a model variable or object property.

SELECTOR Samples

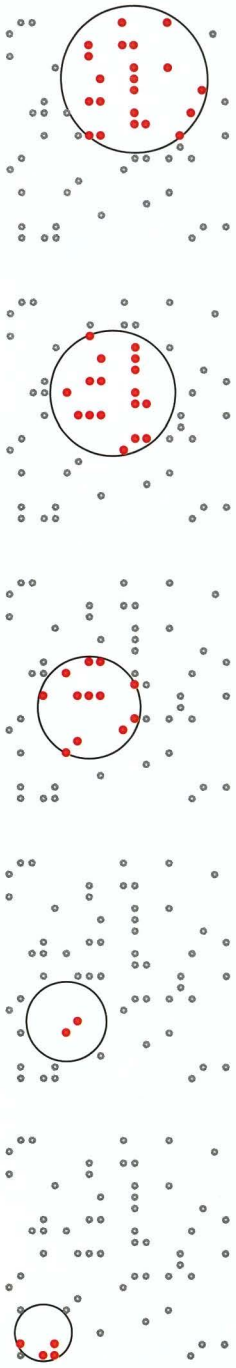
Distance between Points

When. Select points based on their distance from a SELECTOR point.

How. First write a function giving a condition for selecting target points. In this sample example, the distance between the target and the SELECTOR point must be less than variable d.

Write a behaviour function that iterates over the target points and compares the distance between each point and the SELECTOR point using the threshold d. For each member of the list, if the condition is met, the behaviour function creates a copy of the target point.

The behaviour function returns a new list of result points within distance d of the SELECTOR point. Note that the structure of the target and result may differ. For instance, the target may be a 2D array and the result a 1D array. You have to make and remember such choices.



8.45: Selecting from a collection based on distance from a target point is identical to selecting points within a circle centred on the target and with radius of the chosen distance.

Part of Curve

When. Select part of a curve depending on its distance to a point.

How. Place a collection of parametric points on a curve. Use the SELECTOR mechanism from the sample *Distance between Points*, that is, a function that checks each of the points and reports them if they satisfy the distance condition.

Placing a curve through the selected points yields an approximate copy of part of the original curve. Moving the SELECTOR point controls the part of the curve to be copied. The more points on the original curve, the more accurate the curve selected.

The original points sample the curve and so will seldom capture the exact point on the curve that is at the specified distance. Solve this problem by searching between the last selected point and the first non-selected point for the point at which the distance to the SELECTOR is exactly the threshold. However, if the original points are widely spaced, very small sections of the curve that are within the threshold might fail to be detected. Resolve this issue by projecting the SELECTOR point onto the curve. If the distance to the SELECTOR point is less than the threshold, search on either side of the projected point for the exact end points of the curve.

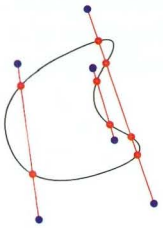
8.46: It is often easier to conceive an overall model from which the intended design component is extracted by selection. In this case, the SELECTOR returns a part of the curve near the target point.

Lines inside Curve

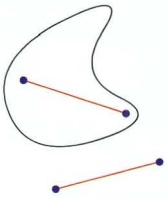
When. Select all lines that are completely inside a closed curve.

How. This SELECTOR has two parts.

First, if a line segment is entirely inside or outside a curve it does not intersect the curve. In order to check the position of a line against a curve, first compute the intersection. If the result is null, the line may be either inside or outside.

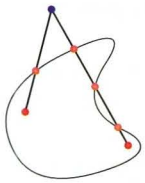


Lines intersect the curve.

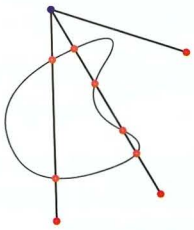


Lines do not intersect the curve.

Second, if an endpoint of a non-intersecting line is inside the curve, so is the line. The Jordan Curve Theorem states that a point is inside a closed curve if a line segment between the point and a point external to the curve intersects the curve an odd number of times. Use the Jordan Curve Theorem to determine the position of either end of the line segment against the closed curve. Don't count tangencies as crossings! If the number of intersections is odd, the line is inside the curve.

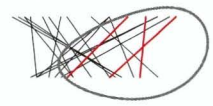


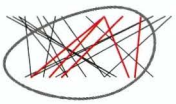
Odd number of intersections.
Point is inside the curve.

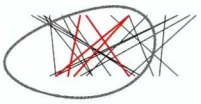


Even number of intersections.
Point is outside the curve.

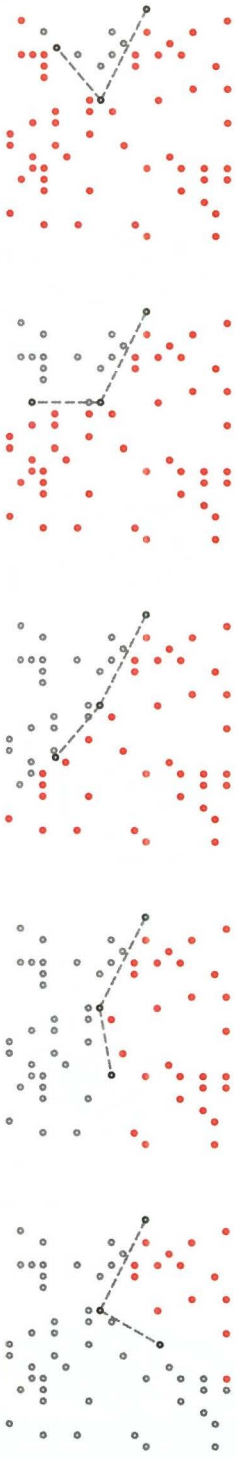
Write a function that performs both of the above tests in turn. If both succeed (non-intersection of the segment and odd intersections of a ray from an end point) copy and return the line.







8.47: The Jordan Curve Theorem, applied twice, gives a simple and correct test for lines inside closed curves.

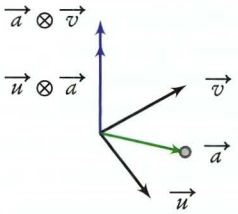


8.48: The cross and dot products combine in a robust point-in-sector test.

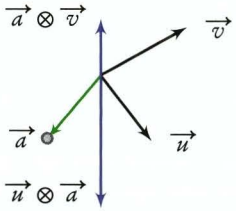
Points inside Sector

When. Select the points inside a 2D sector (two vectors bound to a point).

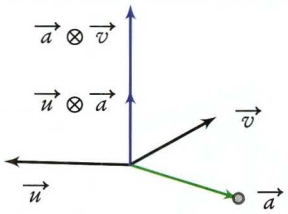
How. The sector comprises a base point and two vectors $\vec{u}$ and $\vec{v}$ bound to it. Taken in order, these define an angle between 0° and 360°. Imagine a target vector $\vec{a}$ for each target point connecting from the base point to the target point. Take the two cross products $\vec{u} \otimes \vec{a}$ and $\vec{a} \otimes \vec{v}$. Remember from Section 6.5.1 that the right-hand rule gives the cross product's orientation. If the angle between the vectors $\vec{u}$ and $\vec{v}$ is less than 180°, a target point is between two SELECTOR vectors if the z-components of both the cross products of its vector with the SELECTOR vectors are positive. If the angle is greater than 180°, the target point is in the sector if one or both of the cross products are positive.



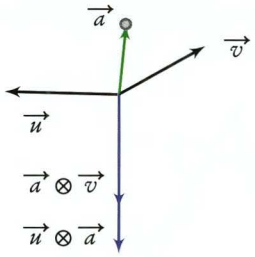
$\angle \vec{u}\,\vec{v} <= 180°$
$\vec{a}$ is inside the sector.
Both cross products positive.



$\angle \vec{u}\,\vec{v} <= 180°$
$\vec{a}$ is outside the sector.
One or both cross products negative.



$\angle \vec{u}\,\vec{v} > 180°$
$\vec{a}$ is inside the sector.
One or both cross products positive.



$\angle \vec{u}\,\vec{v} > 180°$
$\vec{a}$ is outside the sector.
Both cross products negative.

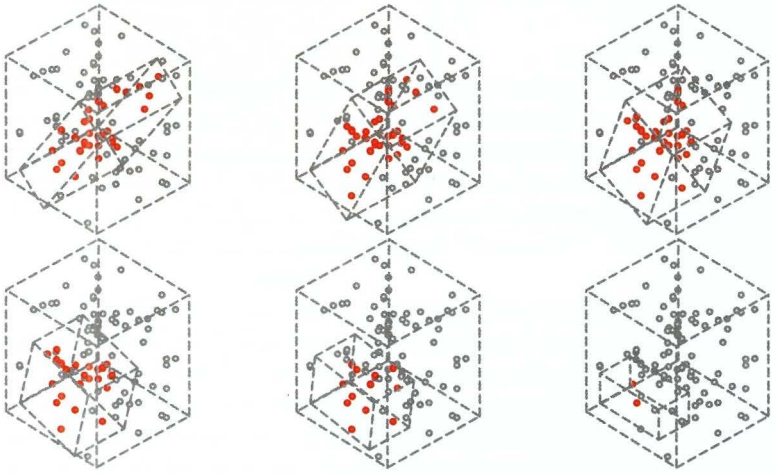
The cross product determines if the sector is greater than 180°. If the sector is between 0° and 180°, the z-component of $\vec{u} \otimes \vec{v}$ is positive. If the sector is between 180° and 360°, $\vec{u} \otimes \vec{v}$ is negative. When $\vec{u}$ and $\vec{v}$ are colinear, the sector is either 0° ($\vec{u} \bullet \vec{v} \geq 0$) or 180° ($\vec{u} \bullet \vec{v} \leq 0$).

The SELECTOR function iterates over the target points one by one. For each point, it computes the two cross products $\vec{u} \otimes \vec{a}$ and $\vec{a} \otimes \vec{v}$. Using the rules above it returns each point that lies within the sector.

Points inside Box

When. Select points that lie inside a box.

How. Write a behaviour function that checks the position of the target points against the box. The function reports their position in the coordinate system of the SELECTOR box one by one and compares their new coordinates to two opposite corners of the box. The function then reports them in the result list if they are inside the box.



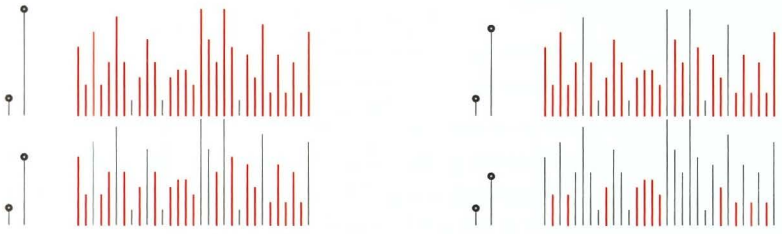
8.49: By defining a SELECTOR box in its own frame and reporting the target points in that frame, the box can have any orientation in space.

Length of Line

When. Select lines based on their length.

How. Given a list of lines, select those whose lengths are between the lengths of two SELECTOR lines.

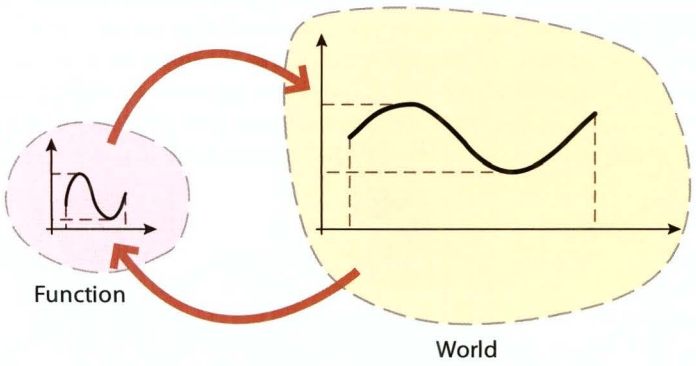
Use a function that iterates through the target lines. For each target line, if it has the desired length, the function reports it and puts it in a new list of result lines.



8.50: This SELECTOR contains a CONTROLLER. The actual SELECTOR object comprises the upper and lower threshold given by the CONTROLLER'S points.

8.15 MAPPING

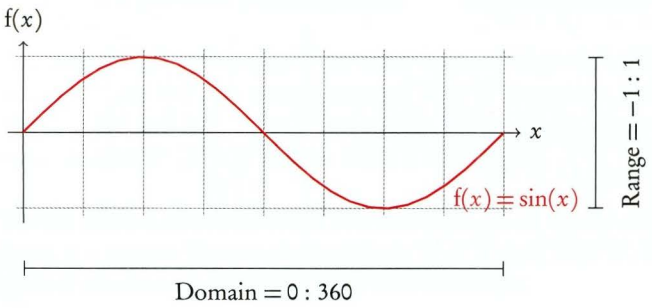
Related Pattern • CONTROLLER • JIG • INCREMENT • PROJECTION



What. Use a function in a new domain and range.

When. A function accepts inputs and produces a value. Geometric functions such as $f(x) = \sin(x)$, $f(x) = \cos(x)$, $f(x) = x^2$ and $f(x) = 1/x$ are extremely common in parametric modeling. In fact, they form an indispensable base for much modeling work. These functions are all naturally defined over their own domains and ranges. Use this pattern when you want to use a function in the domain and range specific to a model.

The terms *domain* and *range* need to be explained. The domain of a function is the set of input values over which it is defined or used. The range of a function is the set of output values it generates. For example, we might choose to use a domain of 0° to 360° for the $\sin(x)$ function. This corresponds to its natural repetitive cycle. The range generated by $\sin(x)$ over this domain is -1 to 1.

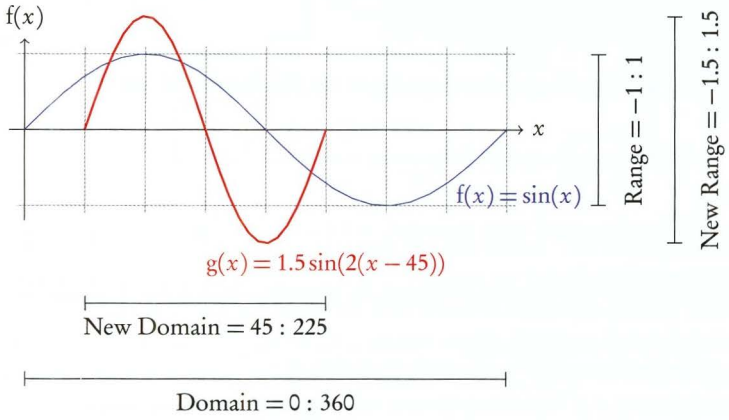


Why. Much form-making comes directly from relatively simple functions, for many reasons. The repetitive use of simple functions can unify a design across its parts and across design scales. If the ease and cost of fabrication is a concern, simple functions can help control complexity and cost (but do not necessarily do so). In contrast, the so-called *free-form* curves and surfaces provide interfaces to more complex functions, but cede some control of the form-making process to the underyling algorithms.

Simple functions come with natural domains and ranges. It is much easier to think about a function in its natural domain and range than in a transformed version.

To use a function in a model requires reframing it so that it makes sense in the model. Reframing turns out to be surprisingly difficult and error-prone for many designers. Yet, there is a universal method of precisely seven parameters that works for almost all reframing tasks. Further, this method is based on one simple equation – essentially the same equation that defines a one-dimensional *affine map* or, equivalently, an *affine function*.

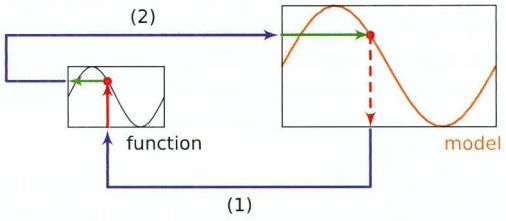
The term "affine map" is extremely common in linear algebra and its dependent fields (computer graphics and geometric modeling). It has a crisp mathematical meaning, and we use this meaning here – precision of language is important! An affine map is a *linear transformation* followed by a translation. In turn, a linear transformation preserves vector addition and scalar multiplication – you can add or multiply before or after the transformation and the result is the same. In one dimension (the real number line), the only linear transformation is scaling. A 1D affine map changes a function's scale and moves it along the real number line. The function $f(x)$ becomes $g(x)$ under the affine map.



The problem for modeling is that determining the new function $g(x)$ is not easy. If you watch a modeler trying to get such a mapping to "work" you will see much trial and error. Looking at the function in the figure above, that is, $g(x) = 1.5\sin(2(x - 45))$ gives some indication why the task is so hard. Which number does what? Why?

This pattern replaces all of the function-specific changes with a uniform structure and set of parameters. You never have to work with a function in any but its simplest form. MAPPING separates where the *model* uses a function from where the function is defined.

How. Consider two rectangles, called the *function* and the *model* respectively. The function rectangle is given by the domain and range of the function within it. The model rectangle can be any size and location you choose. The goal is to place the original function into the model. The basic idea is to always use the function rectangle when computing the function. To find a point on the model function, map from the model domain to the function domain (blue arrow 1 below), compute the function (red and green arrows in the function box) and then map from the result back to the model range (blue arrow 2 below).



What follows precisely defines the diagram. Variables relating to a domain start with or have within a d; those over the range start with or have within an r. In one dimension, an affine map has a single equation, which we introduce first over the domain interval 0 to 1. Imagine a parametric function f(d) over this interval. An affine map r(d) from the interval 0 to 1 to the interval $[r_l, r_u]$ with parameter d has the equation

$$\mathrm{r}(d) = r_l + \mathrm{d}(r_u - r_l),\ 0 \le d \le 1$$

Reversing the map to go from the range (r) to the domain (d) gives

$$\mathrm{d}(r) = \frac{r - r_l}{r_u - r_l},\ r_l \le r \le r_u$$

The map has domain d with bounds 0 to 1 and range r with bounds r_l and r_u.

Generalizing, to any domain produces the maps between d and r as follows:

$$\mathrm{d}(r) = \frac{(r - r_l)(d_u - d_l)}{r_u - r_l} + d_l,\ r_l \le r \le r_u$$

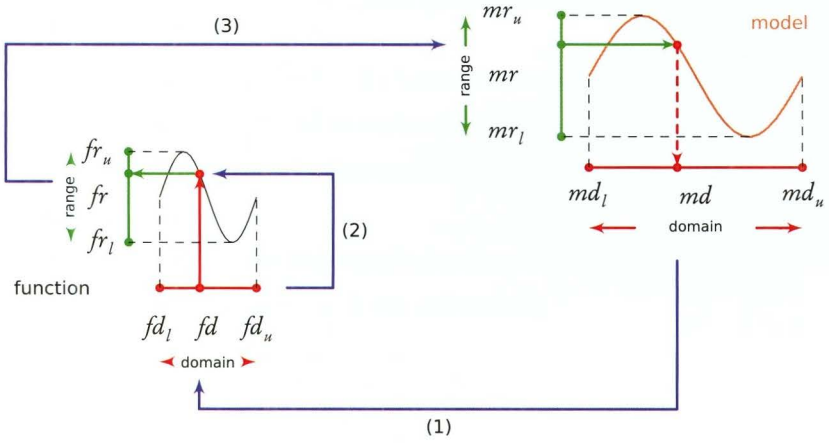
$$\mathrm{r}(d) = \frac{(d - d_l)(r_u - r_l)}{d_u - d_l} + r_l,\ d_l \le d \le d_u$$

Mathematically, these are simple equations, but require attention every time they are used. The purpose of this pattern is to abstract these equations so that designers can freely use generic and simple functions in their models.

WARNING. When using affine maps to apply a function, there are actually two maps to consider. One goes between the domain of the model and its domain in the function. The other goes between the range of the function and its range in the model.

Since the range of the function is determined by the function itself, this means that mapping between function and model had exactly seven parameters: the lower and upper bounds of the function's domain (fd_l and fd_u); the lower and upper bounds of the model's domain (md_l and md_u); the lower and upper bounds of the model's range (mr_l and mr_u); and the function parameter d itself.

These seven parameters describe every possible situation in which you want to use a function in a model. *Every possible situation!* Applying them though takes some insight. The archetypal problem is this: you have a value in the model and need to find the corresponding value of the function in the model. Giving more detail to the diagram above, the solution has three parts: (1) an affine map from the model to the function; (2) an application of the function; and (3) an affine map from the result of the function to the result in the model. The diagram below illustrates this flow, showing how the various parameters and bounds relate.



For example, imagine a roof whose profile is a sine curve. The roof spans from a point $\dot{p}$ in the model to another point $\dot{q}$. A point $\dot{m}(t)$ with parameter t gives the location of the roof point along the line between these two points. The maximum roof height is given by a parameter *height*. Between $\dot{p}$ and $\dot{q}$ the roof goes through two complete cycles of the sine function. Then the following are the six mapping parameter settings. The seventh parameter is t, which gives the parametric location of a varying point on the roof.

$f(x)$	=	$\sin(x)$	
fd_l	=	0.0	
fd_u	=	720.0	
fr_l	=	-1.0	(defined by the function – computed automatically)
fr_u	=	1.0	(defined by the function – computed automatically)
md_l	=	0.0	($\dot{m}(t)$ is parameterized by t over the domain 0 to 1)
md_u	=	1.0	
mr_l	=	0.0	(the height is added to the z-value of $\dot{m}(t)$)
mr_u	=	*height*	(the height is chosen in the model)

In summary, the whole process comprises these three steps:

- Given a model domain value, find the equivalent domain value in the function.
- Apply the function to get a value in the function's range.
- Find the equivalent model range value.

MAPPING Samples

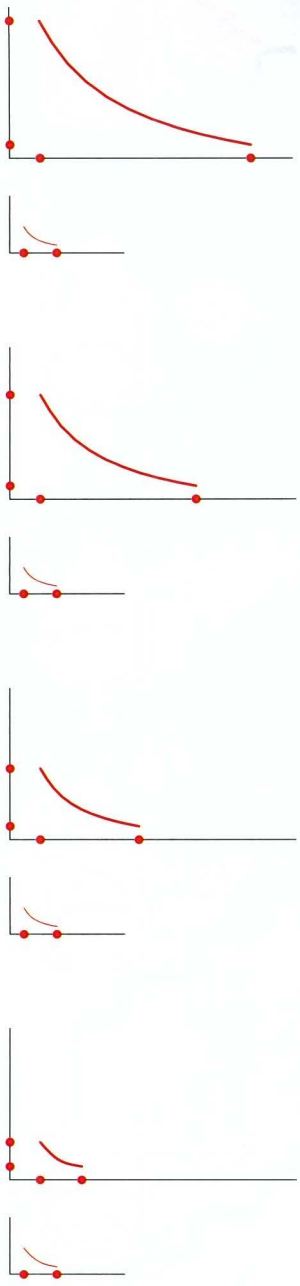
Reciprocal

When. Feather a curve. Make a curve taper gently.

How. The multiplicative reciprocal is the function $f(x) = 1/x$. It is seductive as it provides a simple function that tapers to a non-zero value (is asymptotic to the x-axis), making it possible to feather effects on a curve or surface. It traps the unwary – it increases exponentially as its argument approaches zero and is undefined at zero. Thus it is important to set the function domain so that zero and values close to it are not included. In general, using this function produces poor results. It is included here to demonstrate that sometimes the function domain choice is crucial.

Here are the parameter settings for a useful mapping. The model point $\dot{m}(t)$ is parametric over the domain 0 to 1.

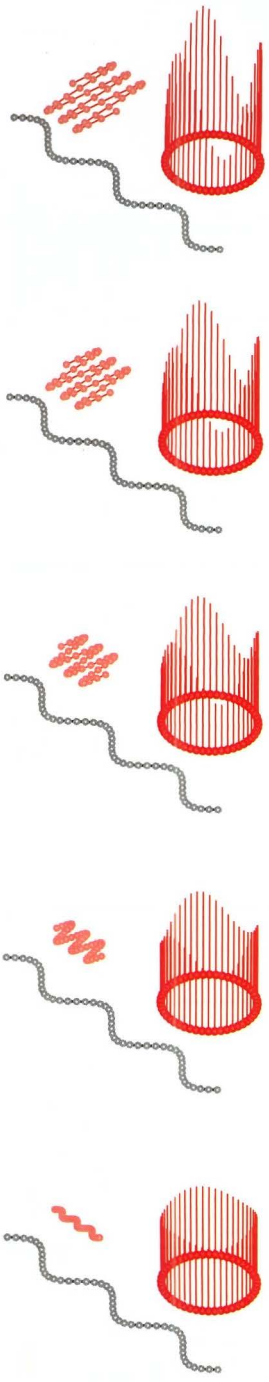
$f(x)$	=	$1/x$	
fd_l	=	0.25	
fd_u	=	100.0	
fr_l	=	0.01	(defined by the function – computed automatically)
fr_u	=	4.0	(defined by the function – computed automatically)
md_l	=	0.0	($\dot{m}(t)$ is parameterized by t over the domain 0 to 1)
md_u	=	1.0	
mr_l	=	0.0	(the height is added to the z-value of $\dot{m}(t)$)
mr_u	=	*height*	(height is chosen in the model)



8.51: Beware the reciprocal.

$f(x) = 1/x$

It seems to taper nicely along the x-axis but goes to infinity along the y-axis. Using it usually leads to anomalies in the model.



8.52: Three cycles of the cosine function mapped into space at different scales. Vertical lines from points on a circle use the y-coordinates of the mapped points as their length.

Sine and Cosine

When. Use sine and cosine functions in complete periodic cycles.

How. The basic trigonometric functions $\sin(x)$ and $\cos(x)$ repeat in periods of 360° and span 180° between function minima and maxima. Both functions have domains from $-\infty$ to ∞ and ranges from -1 to 1. Choosing a finite part of the domain such that the function starts and ends at a minimum, maximum or zero-crossing yields clean control over the end-conditions in the model.

Here are sample parameter settings. The model point $\dot{m}(t)$ is parametric over the range 0 to 1. In this case, $\dot{m}(t)$ is bound to a circle, and vertical lines from each instance of $\dot{m}(t)$ have length given by the function result.

f(x)	=	cos(x)	
fd_l	=	0.0	
fd_u	=	1080.0	(yields three complete cycles)
fr_l	=	−1.0	(defined by the function – computed automatically)
fr_u	=	1.0	(defined by the function – computed automatically)
md_l	=	0.0	($\dot{m}(t)$ is parameterized by t over the domain 0 to 1)
md_u	=	1.0	
mr_l	=	$height_{min}$	(the minimum length of the lines on $\dot{m}(t)$)
mr_u	=	$height_{max}$	(the maximum length of the lines on $\dot{m}(t)$)

When sampling periodic functions and those with minima or maxima within the chosen domain, the choice of sampling interval is crucial if the sampled points will be used to regenerate the mapped function in the model. Samples must be chosen to coincide with function minima and maxima. Poor sampling can dramatically affect the shape of the reconstructed curve.

Function Parts

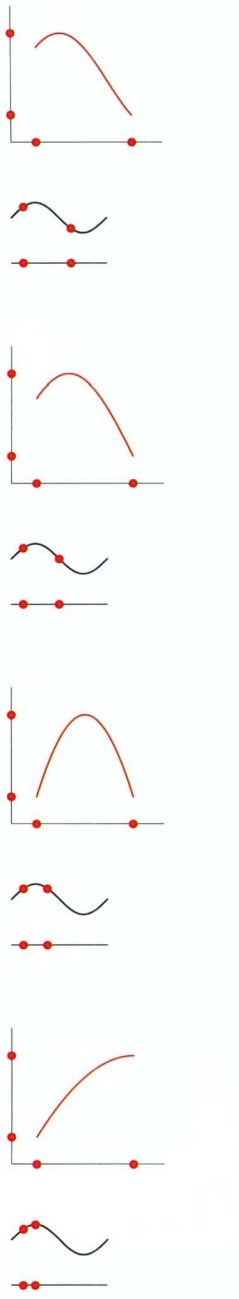
When. Choosing a small part of a function can yield an intended form.

How. Simple functions can yield surprisingly rich forms. A classic example in three dimensions is Foster + Partner's extensive use of parts of the torus as a generator of form (see Chapter 5).

The function is the sine function: $f(x) = \sin(x)$. The trick is that an appropriate choice of function domain can yield local segments of the sine curve that do not display the archetypal repetition of the entire curve.

Here are sample parameter settings. The model point $\dot{m}(t)$ is assumed to be parametric over the range 0 to 1. In this case, the model range minimum and maximum would be chosen by the modeler to suit the application to hand. The figures on the side bar vary only by $FDu = 180.0$ the upper function domain bound.

$f(x)$	=	$\sin(x)$	
fd_l	=	45.0°	(increases in steps of 22.5°)
fd_u	=	180.0°	
fr_l	=	−1.0	(defined by the function – computed automatically)
fr_u	=	1.0	(defined by the function – computed automatically)
md_l	=	0.0	($\dot{m}(t)$ is parameterized by t over the domain 0 to 1)
md_u	=	1.0	
mr_l	=	$height_{min}$	(minimum height is chosen in the model)
mr_u	=	$height_{max}$	(maximum height is chosen in the model)



8.53: Choosing part of a function can lead to surprising and useful results. If the underlying function has known "nice" properties, such as maxima or minima at known inputs, so will the selected function.

8.16 RECURSION

Related Pattern • POINT COLLECTION

What. Create a pattern by recursively replicating a motif.

When. Hierarchy in design conflates wholes and parts. The parts copy and transform the whole. This naturally leads to a hierarchcial information structure of *layers*, where the properties of a layer derive from the layer immediately superior to it. Recursive algorithms are natural traversers of such hierarchical structures. Use this pattern when you are working with hierarchy in design.

Why. Some complex models such as spirals, trees or space-filling curves can be elegantly represented with a recursive function. In fact, it can be so difficult to represent such structures non-recursively that designers give up and try easier forms. A recursive function typically takes a motif and a replication rule and calls itself on the copies of the motif that the rule generates. A word of warning – recursive algorithms are often hard to understand in the abstract. We humans struggle to envision a pattern from a motif and a replication rule (Carlson and Woodbury, 1994). Another word of warning. Recursive algorithms as update methods can be slow. Typically, limiting the depth of the recursion is the only way to maintain an adequate interactive update rate.

How. RECURSION requires a motif (a geometric object) and a replication rule. The simplest rule is merely a coordinate system with some combination of translation, rotation, scale and shear. Other more complex rules are possible, including ones that change or abstract the motif itself.

The recursive function uses the replication rule to generate a collection of clones of the motif. It then calls itself on each clone, typically (but not necessarily) with the same replication rule. So, in each step, the recursive function takes an existing motif, replicates it and calls itself. Every recursive function requires a termination condition to specify when to stop this process.

RECURSION Samples

Square

When. Nest a sequence of squares inside an initial square.

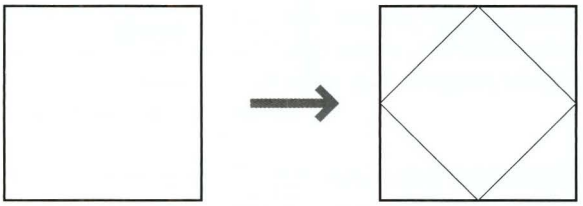
How. Start with a square (actually, any polygon will do). This is the initial motif – the input to the recursion.

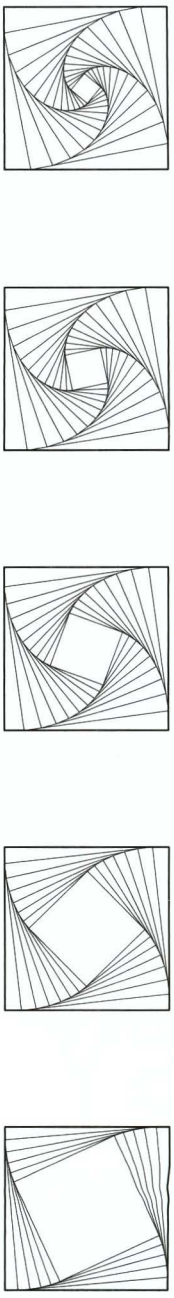
The recursive function must transform the motif as a copy and call itself on the copy. In this sample, the transformation is specified as a parameter t. The new motif's vertices are parametric points on the original motif's edges with parameter t.

In a computer recursions must stop (though this is not true in mathematics!). They either stop by design or by overflow of some data structure, typically the recursion stack in the programming language. In this sample, the variable *depth* controls the recursion and is thus an argument to the function. Each subsequent call to the function reduces the depth argument by one. Inside the function, a test for depth returns the motif unchanged if *depth* is equal to one, else proceeds with the recursion.

Assigning each of the squares an elevation results in a 3D "stack" of polygons. Using this stack as the arguments for a surface gives a twisted vertical pyramid.

This sample defines one copy of the motif at each recursive call. The result is a linear list of motifs – the structure of the list mirrors the geometric structure of the result. Defining more than one motif within the function results in a *tree*.





8.54: A simple one-dimensional recursion.

Tree

When. Define a tree – both graphically and as a data structure.

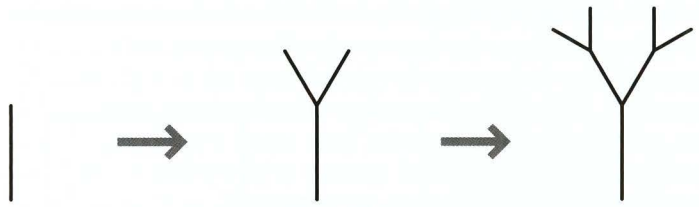
How. In this sample, the motif is a single line. The focus here is more on the geometric layout and data structure and less on the resulting pattern.

The motif transformation places two copies of the motif, each starting at the endpoint of the original motif. The copies are rotated by the parameter *rot* and scaled by the parameter *scale*.

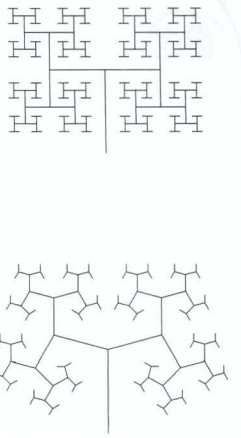
As for any recursive function, you need to make sure that the recursion will stop. As in the prior sample, the stopping condition is set by the depth of the recursion.

With the hindsight gained by seeing the recursion in operation, we can form a good idea of what this particular function will produce as the *rot* and *scale* parameters change. With compound motifs and more complex transformations, such intuitions completely dissolve (Carlson and Woodbury, 1994).

A tree is a data structure that has branches. Each branch is either null or itself a tree. The recursive function determines the data structure. The preferred structure provides a *path* to each of the motifs that mirrors the geometric structure. For example, a sensible path might be one that starts with the root of the tree and that records which branch of the tree is followed to reach the motif. Thus, for a tree with depth 4 (with the base motif at depth 0), one path to a node would be tree[1][2][2][1]. Interpret this as taking the right branch on a 1 and the left branch on a 2. The data structure is thus a list where the first item in the list is the right branch of the tree and the second item in the list is the left branch of the tree. The motif itself must be stored somewhere and is assigned to the 0^{th} branch of the data structure. So a path to a motif needs an index of 0 at its end, for example, tree[1][2][2][1][0].



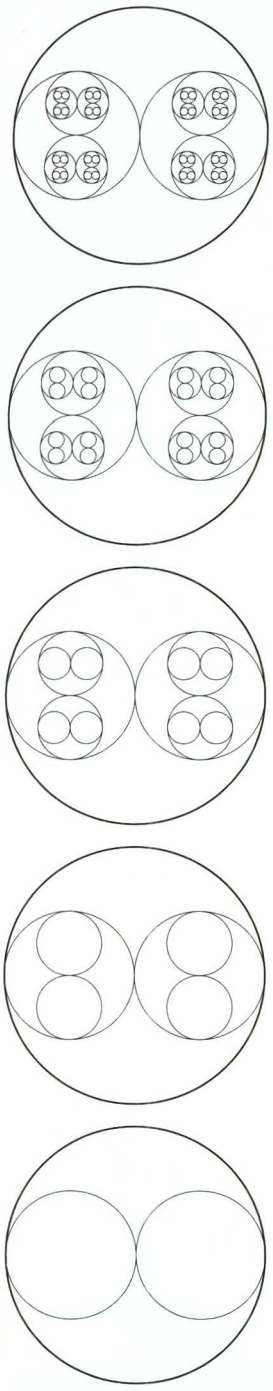
```
treeFn function (CoordinateSystem cs,
                 Line startLine,
                 int depth, double rotation, double scale)
{
 Line resultLine = {};
 if (depth < 1){
  resultLine[0] = null;
  resultLine[1] = null;
 }
 else{
  CoordinateSystem rightCS = new CoordinateSystem();
  rightCS.ByOriginRotationAboutCoordinateSystem
          (startLine.EndPoint,
           cs,
           rotation,
           AxisOption.Y);
  CoordinateSystem leftCS = new CoordinateSystem();
  leftCS.ByOriginRotationAboutCoordinateSystem
         (startLine.EndPoint,
          cs,
          -rotation,
          AxisOption.Y);
  Line rightLine = new Line();
  rightLine.ByStartPointDirectionLength
            (rightCS,
             rightCS.ZDirection,
             startLine.Length*scale);
  Line leftLine = new Line();
  leftLine.ByStartPointDirectionLength
           (leftCS,
            leftCS.ZDirection,
            startLine.Length*scale);
  resultLine[0]=treeFn(rightCS,
                       rightLine,
                       depth-1, rotation, scale);
  resultLine[1]=treeFn(leftCS,
                       leftLine,
                       depth-1, rotation, scale);
 }
 resultLine[2]=startLine;
 return resultLine;
};
```



Note the internal calls to the treeFn function. The two separate calls ensure that the right and left branches of the tree are themselves trees. The base case of the recursion occurs when the depth is less than one and results in a tree with null branches being returned. At the end of the function, the motif is stored in the second member of resultLine, completing the data structure, which now stores both the tree structure and all the motifs.

A word to the wise. Writing recursive functions so that they return useful data structures is careful, error-prone work. Doing it well is key to making sense of the results.

8.55: The tree recursive function should produce a data structure that maps to the tree geometry. Each of the figures above shares a common data structure – the geometric variation is due to parameter settings alone.

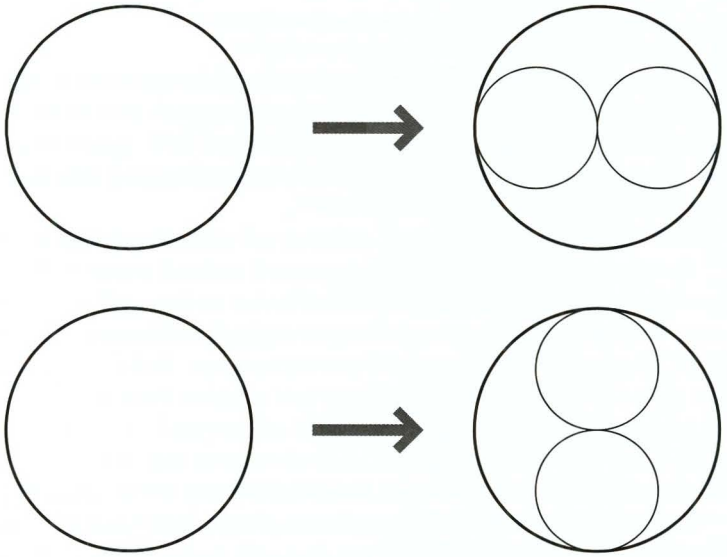


8.56: In this recursion much of the coding effort is devoted to changing the orientation of the circle pair at successive recursive levels.

Circles

When. Nest circles within a circle, changing the nesting direction at each level.

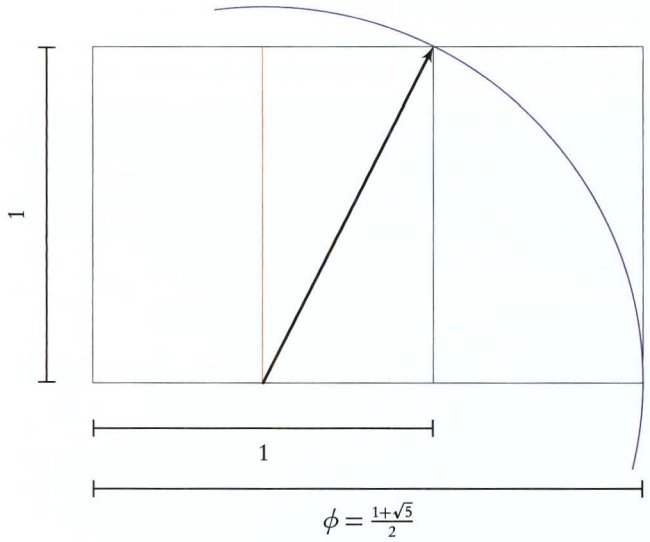
How. The motif is a circle. It replicates twice inside itself, with the two new circles of half the radius and tangent to both themselves and the original circle. The line joining their centres is initially horizontal. At each recursive level, the line needs to alternate between horizontal and vertical. This requires a check on the parity of recursion depth within the function. As with the prior samples, the recursion is depth-limited.



Golden Rectangle

When. Subdivide a golden rectangle into a square and another golden rectangle.

How. A *golden rectangle* is a rectangle whose side lengths are in the golden ratio $1 : \phi$, that is, approximately $1 : 1.618$. A distinctive feature of this shape is that when a square section is removed, the remainder is another golden rectangle, that is, having the same proportions as the original.

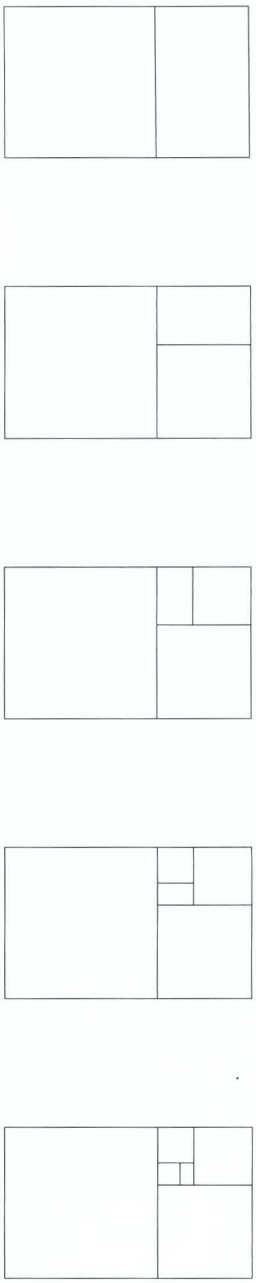


This sample represents a golden rectangle of length l as a sequence of rectangles. If there is only one member of the sequence, it is a golden rectangle with the short side of length l and long side of length $(1 + \sqrt{5})/2$. If there is more than one member, each member but the last is a square, starting with side length l and reducing by $1/\phi$ at every member. The final member is a golden rectangle. Each successive square is located $l * \phi$ away and rotated 90° from the last.

In this sample, the recursion condition is area. When the area of the next square would be less than a threshold $minArea$, the function produces a single golden rectangle and returns. Such a constraint is more realistic than recursion depth – in design there may be a minimum feature size for fabrication.

The data structure produced is simple: a linked list of rectangles, all but the last being a square.

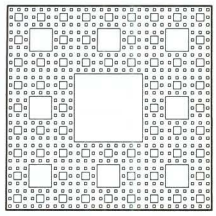
You can also create golden rectangles from the "inside out" by adding squares to a seed rectangle.



8.57: The golden rectangle is an archetypal recursive form. At each level, the rectangle comprises a square and another, smaller golden rectangle.

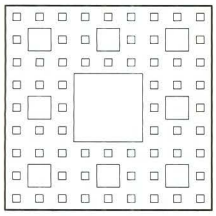
Sierpinski Carpet

When. Define a Sierpinski carpet.



The Sierpinski carpet is a plane fractal, that is a recursive, self-similar form. Wacław Sierpiński discovered it in 1916. Its construction begins with a square. Conceptually, the square is cut into nine congruent subsquares in a 3×3 grid, and the central subsquare is removed. The same procedure applies recursively to the remaining eight subsquares.

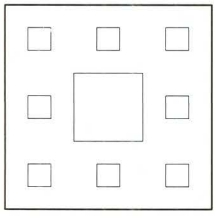
How. At each recursive level, the carpet comprises a motif: a square of 1/3 the size of the carpet at that level and placed as the centre of the carpet. At all but the base levels, it has eight additional carpets arranged around the motif. Both the recursive function and the data structure should reflect this arrangement.

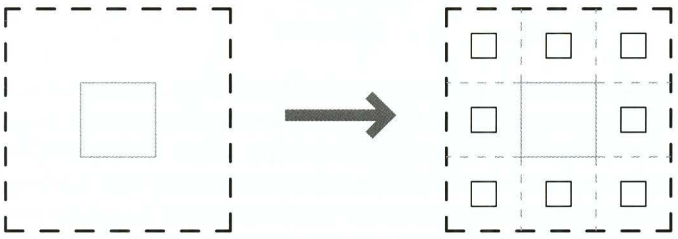


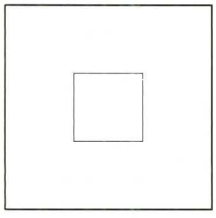
The number of times a recursive function calls itself at a single level is called its *branching factor*. A level 0 carpet has a single motif. A level 2 carpet has nine motifs. The number grows very quickly. A level 6 carpet has

$$1 + 8 + 64 + 512 + 4096 + 32768 + 262144 = 299593 \text{ motifs.}$$

Clearly, you have to be careful to limit recursion depth when faced with high branching factors.







8.58: At each recursive level the Sierpinski carpet "cuts away" 1/9 of the remaining square and applies itself to the rest.

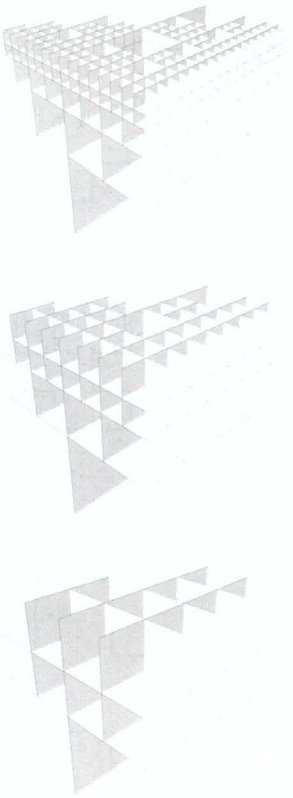
3D Planes

When. Recursively divide space with three perpendicular polygons.

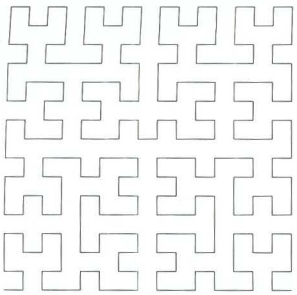
How. The motif comprises a 3D cruciform of square polygons. The cruciform divides the space it occupies into eight cubes and, at each level, the recursive function places a scaled copy of the cruciform into three of these.

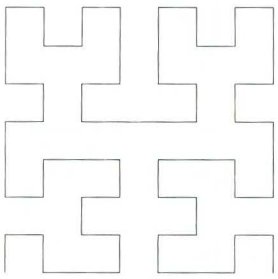
This sample displays more architectural reality than the prior samples. In fact, it resembles a recurrent motif in the work of Arthur Erickson, for example, the image below shows the Simon Fraser University academic quadrangle.

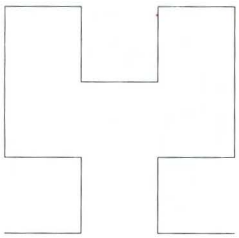
Academic Quadrangle, Simon Fraser University, by Arthur Erickson
Source: Greg Ehlers / Media Design, Simon Fraser University

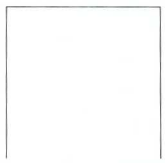


8.59: Recursion is the computational implementation of the design strategy of hierarchy. Of course, actual designs require recursive functions more specific than this simple sample.









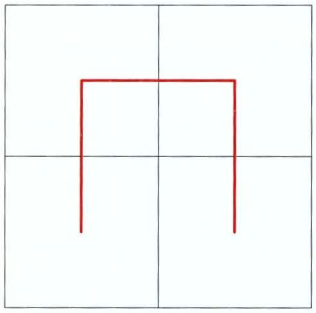
8.60: As the recursion level increases, the Hilbert curve progressively fills the space it occupies.

Hilbert Curve

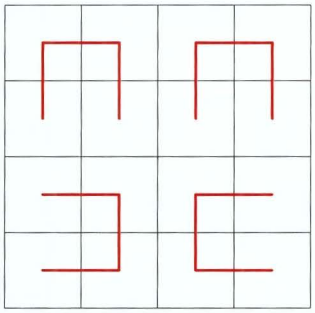
When. Progressively fill space with a single curve.

How. The Hilbert curve fills space. At the n^{th} recursive level it visits every point in a $(2^{n+1}) \times (2^{n+1})$ integer dimensioned space. For example, at the 0^{th} level it visits all points in a 2×2 space.

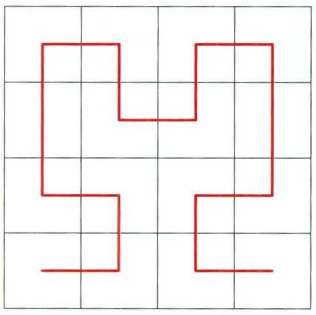
At the 0^{th} level, shown in (a) below, the curve comprises three lines, joining centres of the four quarters of the 2×2 space. At the second level, shown in (b), a new 2×2 space replaces each of the four points. The points of this space define four new curves. Joining the endpoints of each curve segment (c) with the start point of the next produces a continuous curve. At each subsequent level (d), a further 2×2 space replaces each point at the previous level. Successive levels of the Hilbert curve thus form a progressively elaborating sequence (d).



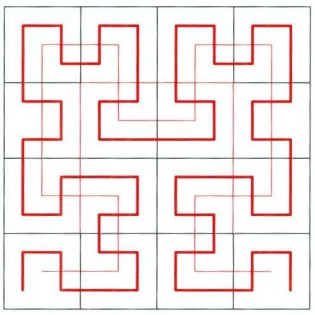
(a)



(b)



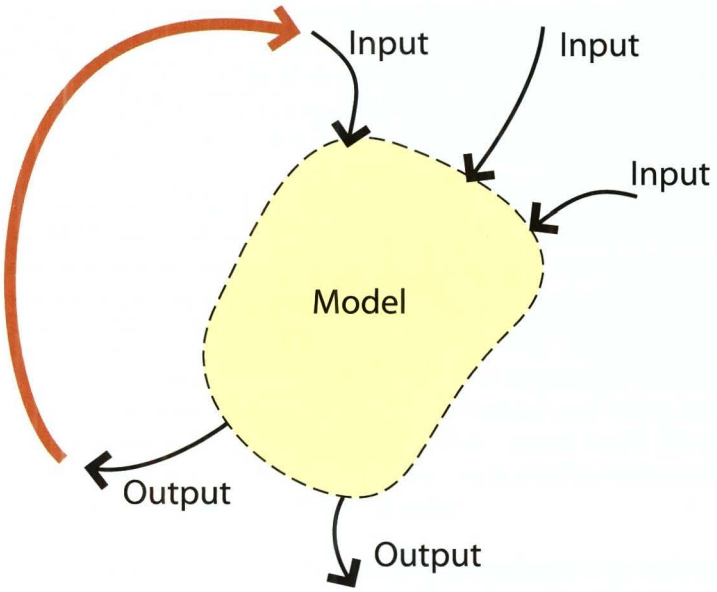
(c)



(d)

8.17 GOAL SEEKER

Related Pattern • REACTOR



What. Change an input until a chosen output meets a threshold.

When. Parametric models are acyclic – data flows downstream. In other words, you need to know parameter values to produce a result. Sometimes though, knowledge works the other way. You know a goal for a particular variable and want to discover a set of input values that will achieve it. Use this pattern when you want to adjust inputs until you reach a goal.

Why. Without values for its independent variables, a model is *undetermined*, that is, it does not contain sufficient information to give values to its dependent variables. Typically, a model can exist in an indenumerable infinity of states, depending on the choice of its input values. Sometimes, you know a property of such a state. You may even be able to adjust input values until the property is achieved. But accuracy and reproducibility are important. A GOAL SEEKER can compute the needed input values.

How. A model has inputs and some outputs. A GOAL SEEKER requires a choice of both: an output that will be evaluated and an input that will be adjusted. The output is called the *result* and input the *driver*. The threshold that the result should meet is called the *target*. The process of calculating the result from the inputs is the *update method*.

The GOAL SEEKER script runs the update method then checks the result. If it meets the target, the job is done and it returns. If not, it goes back, slightly changes the driver, runs the update method and checks the result again. This loop continues to run until the desired result is achieved.

A simple way of being systematic is to use a binary search, in which an estimate of the distance to the target determines changes to the driver. While searching, the incremental step change of the driver may cause the result to pass the target. If this happens, the script reverses and reduces the step size. Then it continues changing the driver until it passes the target again. It repeats the search process until the result meets the target (with adequate precision).

The simple GOAL SEEKERS presented in this pattern require that the model (or at least the result) changes smoothly with changes in the driver. If the result varied in sharp jumps, the strategy of slowly changing the driver to approach a result would not work. Such situations present complex problems of *discrete search* or *constraint satisfaction* that are beyond the scope of simple elements of parametric design.

GOAL SEEKER Samples

Local Maximum

When. Locate the point on a curve at which the curve is at a maximum.

How. Elementary calculus (or just looking at a curve) tells us that, at maximum (or minimum) points, the tangent to a curve is horizontal. The angle between the tangent and a horizontal line is zero. Searching for a fixed value simplifies the GOAL SEEKER script in comparison to looking for an unknown maximum. The tangent changes predictably as a point moves across a maximum. Its slope is greater than zero on one side of the maximum and less than zero on the other side. This gives a very simple rule for changing the driver: always move towards zero.

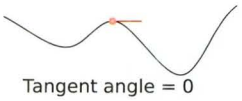
The essential idea is simple. Start at a known point on the curve. Always step upwards. At each step measure the slope. If it is zero, stop. If it changes sign, turn around, takes smaller steps and keep going. Not surprisingly, this is called a hill-climbing strategy. It has some problems. If there is a local hilltop in the direction you start walking, you will reach it and be trapped, even if you can see a taller hill nearby. If the hilltop is really small, that is, small in relation to the steps you are taking, you might miss it altogether.

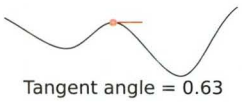
Key to writing a working GOAL SEEKER is understanding how to build the desired measure into the system. Understand how the result will change with changes to the driver. In this case, the tangent measure makes the choice easy. In other cases code may be needed to check the effect of change of the driver on the result and to choose the appropriate direction of change.

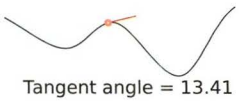
The code for this GOAL SEEKER is relatively simple. Unfortunately, other GOAL SEEKERS require significantly more complex code. There are two nested while loops. The outer loop implements the binary search, the inner one the "walk" towards the target. The inner loop has a test

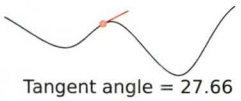
```
driver > driver.RangeMinimum && driver < driver.RangeMaximum
```

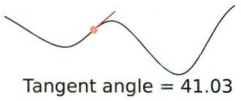
that ensures that the point remains on the parametric curve. If a curve end is a local maximum, the GOAL SEEKER will approach the end, but never overshoot, and will finally arrive at it.











8.61: To seek a local maximum move a point "uphill" along the curve. When the point passes the maximum, change direction and divide the size of the move by two.

Code for the LOCAL MAXIMUM sample.

```
function generalNumericTest (object booleanTest,
                             double a, double b)
 { //provides conditional test of two numeric variables
   //based on an input string.
  switch (booleanTest){
  case ">=": return a >= b;
  case "<=": return a <= b;
  case ">" : return a > b;
  case "<" : return a < b;
  case "==": return a == b;
  default: return true;
  }
 }

double currentDriver = driver.Value;
   //driver is a named variable in the model.
double target = 0.0;
double closeEnough = 0.000000001;
int giveUpWhen = 200;
int incrementAdded = 0;
int incrementSubdivided = 0;
double increment = 0.2;
object startingSide = "gt";
int incrementSign = 1;

if (result>target){
  startingSide = ">";
  incrementSign = 1;
 }
 else{
  startingSide = "<=";
  incrementSign = -1;
 }
while (increment > closeEnough &&
       incrementSubdivided < giveUpWhen)
 {
  ++incrementSubdivided;
  increment = increment/2.0;
  while (generalNumericTest(startingSide,result,target) &&
         incrementAdded < giveUpWhen &&
         driver > driver.RangeMinimum &&
         driver < driver.RangeMaximum)
   {
    ++incrementAdded;
    currentDriver = driver.Value;
    driver = currentDriver+(incrementSign*increment);
    UpdateGraph();
   }
  driver = currentDriver;
  UpdateGraph();
 }
```

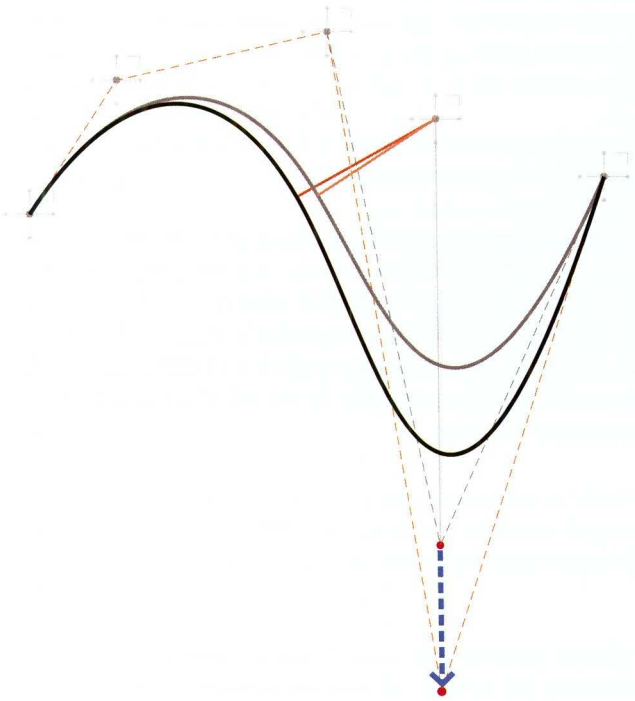
Curve and Point Distance

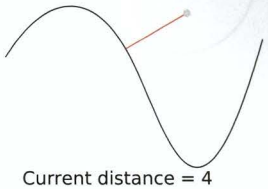
When. Adjust a curve until it is exactly a given distance to a point at its closest approach.

How. In this sample the goal is a minimal distance. Every point on a curve is some distance from a given point. One (or more) of the curve points lie at the least distance. Such points are *projections* of a point onto a curve.

Clearly, any of the control points on a curve can be changed. For each of these points, any direction of change could be used. Using a GOAL SEEKER requires choice of both point and direction of movement. Other choices of point and direction can yield vastly different curves, but they will be at the goal distance (if the GOAL SEEKER works).

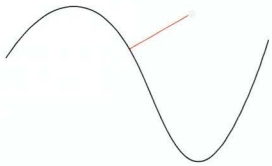
Once a control point and direction of movement are chosen, a GOAL SEEKER works as described above: walk towards the target until you overshoot; back up and take smaller steps; and keep doing this until you are as close as you can discern.





Current distance = 4

Target distance = 4



Current distance = 3.928

Target distance = 4

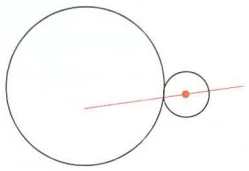
Current distance = 2.984

Target distance = 4

Current distance = 2.107

Target distance = 4

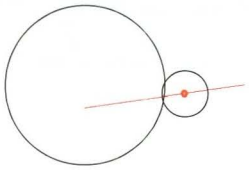
8.62: A curve can be moved in an infinity of ways. In this sample one of the control points moves along a chosen line.



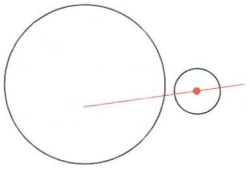
Area

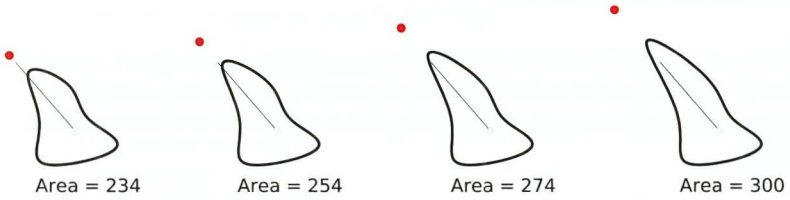
When. Adjust a control point of a curve until the curve encloses a given area.

How. The goal here is the area of a closed curve. As in the previous sample, any of the control points of the curve can be moved in any direction. The modeler must choose. This particular sample moves the chosen control point away from the centroid of all other control points. This is a useful approximation, but, with some work, any other direction could be chosen.



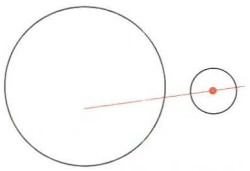
The GOAL SEEKER is almost identical to previous GOAL SEEKERS. The details of the curve, the chosen point and its direction are all factored into the single variable called *driver*.





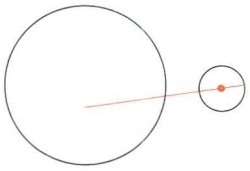
8.63: This GOAL SEEKER moves a control point of a curve along a line until the area of the curve reaches a threshold (in this case 300).

Two Circles



When. Given a circle constrained to move along a line, find the position of its centre such that it is tangent to another circle.

How. Computing tangency is easy if the circles are free. The two circle centres form a line. Move one circle along the line until its centre is plus or minus its radius from the intersection of the line and the other circle. This situation is different – the centre of one circle is constrained to lie on an arbitrary line. The circle centre is governed by the parameter t, which the GOAL SEEKER adjusts until the tangency conditions are met. The GOAL SEEKER must operate twice: once for each tangency condition.



8.64: The small circle's centre moves in increments along a line until the two circles intersect. The direction of movement reverses and the increment size halves. These two steps repeat until the two circles touch.