

other improvements to the language. These improvements have become so much a core of the current tool set that it is now difficult to imagine working without them.

One of the aims of this book is to give you the inside running on how to use the building blocks well, avoiding the pitfalls. From that point you will begin to use your skills and other resources to push the boundaries of GDL.

What can be Modeled using GDL?

3D modeling is the most creative aspect of GDL. Many of the resources outlined in the following pages are designed to help model objects ranging from very simple forms created using the simplest GDL building blocks, to highly parametric library parts and freeform bodies. To give you an idea of how the various resources can be used, consider the following objects.



The chain links of this playground swing were modeled using the *tube* command discussed in chapter 11 *3D Model*. The seat was modeled from *primitive elements* as explained in chapter 20 *Freeform 3D Modeling*, using the *curve approximation methods* of chapter 16 *Approximating Curves*.

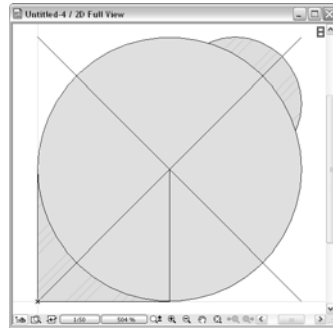
The chain links also draw on chapter 7 *Loops* and chapter 22 *Optimizing for Speed*.

The flowers on this daylily use the *cardboard cutout* technique described in chapter 20 *Freeform 3D Modeling*, and *images with transparency* as explained in chapter 14 *Working with Graphics*. The leaves are modeled using the *primitive elements* of chapter 20.



The branch system of this tree is constructed using *primitive elements* as explained in chapter 20 *Freeform 3D Modeling*. It draws heavily on techniques described in chapters 16 *Approximating Curves* and chapters 17 & 18 *Linear Algebra*. The tree can be *randomized*, and if this option is selected, every point on its surface must be re-calculated. The techniques of chapter 22 *Optimizing for Speed* are used to ensure the extra processing does not cause undue delays.

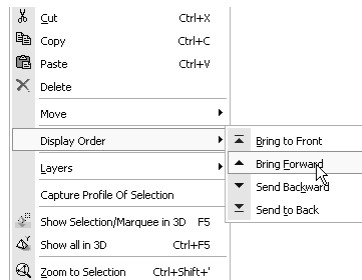
Click on the *2D Full View* button to see the result. As you can see, the two lines are in front of the fill polygons even though the polygons were placed after the lines. In contrast, the small round fill polygon is drawn behind the large round one, including the outline. But look at the small square polygon! While the fill pattern is obscured by the large round fill polygon, the outlines are sitting on top of it!



The `drawindex` command provides more dependable control. Use this command to force certain elements in front or behind others. It works a bit like the *bring-to-front* or *send-backward* options in *ArchiCAD's Display Order* menu. Within a set of elements sharing the same drawing index, elements are stacked as normal.

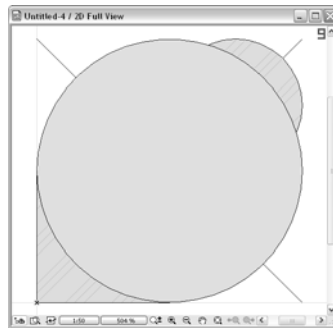
```
drawindex display_order_index
```

The value of `display_order_index` sets the order in which elements are displayed behind and in front of others. Oddly, there are only five values available `display_order_index` – 10, 20, 30, 40, and 50. Elements placed on `drawindex` 50 are drawn in front of all other 2D elements. By default, elements are placed on `drawindex` 10.



Add some `drawindex` commands into the script as follows.

```
!Draw the lines first
drawindex 10
...
!Draw a fill polygon with all edges
drawindex 20
...
!Draw a fill polygon with all edges
drawindex 30
...
```



Click on the *2D Full View* button to see how the `drawindex` commands effect the 2D symbol. Now the square fill and the lines are masked by the large round fill polygon, and the lines are masked by the small fills too.

Bezier Splines

Splines are mathematical curves that can be fitted to data. Traditionally, a spline consists of a set of *interdependent functions that must be solved simultaneously*. Each function starts at one data point, and ends at the next. Pairs of adjacent functions share the same slope and second derivative where they meet. As we'll see in the next section, there is only one solution to the resulting system of equations, so traditional splines choose their own course through the data points, making them difficult to control.

Bezier splines differ from traditional splines in that *the individual functions can be adjusted independently*. This makes them very easy to control, and ideal for certain types of freeform modeling.

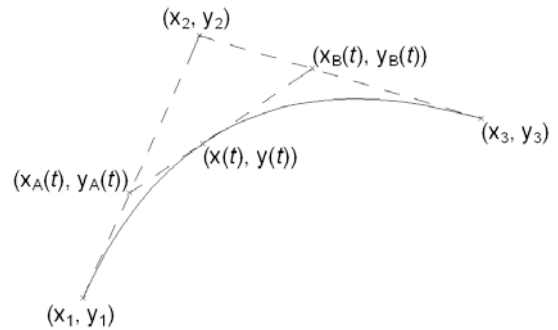
Whichever type of spline you choose, an *independent* spline equation can be generated for *each co-ordinate*. Equations are constructed that make x and y functions of an arbitrary variable t , rather than making y a function of x . This allows us to create curves in 3-space that form complete loops, or even knots.

In this section we will take a look at Bezier splines.

Quadratic Bezier Splines

The simplest of the Bezier splines is the Quadratic. To construct a quadratic Bezier spline, you need two *end points* (x_1, y_1) and (x_3, y_3) , and a *control point* (x_2, y_2) . The control point defines the curve's *tangents* at the end points.

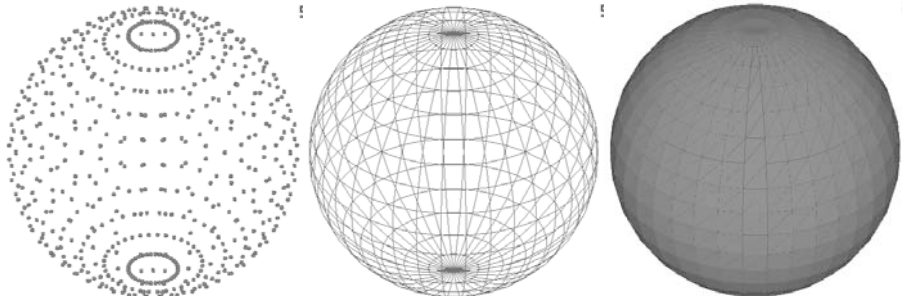
The easiest way to picture how the curve is drawn is to construct *control lines* between the end points and the control point. Now imagine a third line segment that stretches between our control lines. The two end points $(x_A(t), y_A(t))$ and $(x_B(t), y_B(t))$ of the new line segment are free to slide along the control lines $(x_1, y_1) - (x_2, y_2)$ and $(x_2, y_2) - (x_3, y_3)$ respectively. At time $t = 0$, the start point of the slider lies on (x_1, y_1) and the end point is located at (x_2, y_2) . As time progresses, the points slide with constant speed along the line segments until, at $t = 1$, the start point reaches (x_2, y_2) and the end point reaches (x_3, y_3) .



GDL Primitives

In this section, we will look at the elements used to construct freeform bodies, and briefly consider how they are used in practice. This is a theoretical section. We'll apply the theory in the following sections.

GDL *primitives* are used to construct a *body*. First a set of points or *vertices* is defined in space. Pairs of vertices are then joined to form *edges*. Next the edges are joined to define *polygons*. Finally, a closed net of polygons is grouped to form a solid body.



GDL primitives can also be used to map textures to existing bodies. For standard 3D elements such as prisms or cylinders, primitives provide limited control over how the texture is mapped onto the body's surface. Much greater control is available when constructing bodies from primitives.

Base

The GDL interpreter refers to each vertex, edge or polygon by an *index*.

The index of an edge is set by the order in which the edge was defined. The first edge that occurs in a GDL script is referred to by the index 1, the second edge has index 2, and so on.

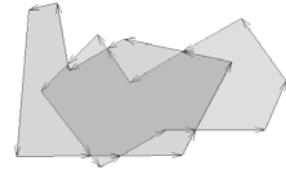
The same is true of vertices and polygons. The first vertex to be constructed has index 1, the second 2, etc. Similarly, the 53rd polygon to be constructed is referred to by index 53.

All the regular GDL bodies (prisms, spheres, cylinders, tubes and the like), are built from primitive components. As a result, when you come to construct a body using primitives, a large number of vertices, edges and polygons may have already been defined. Thus the vertex, edge and polygon indices are not necessarily set to zero.

It is essential that you use the `base` command to *re-set the indices to zero* whenever you begin to define a new body. This ensures that, when you refer to edge 1, you are referring to the first primitive edge element that you

Intersection of Two Polygons

The process of finding the intersection of polygons A and B, is similar to that of merging the polygons. We split the edges, choose which to include, and re-combine them to form polygons.



The main difference is choosing which edges to include in the resulting polygon. For an intersection, we mark each edge of polygon A as *included* if it lies *loosely inside* of polygon B. We mark each edge of polygon B as *included* if it lies *strictly inside* of polygon A.

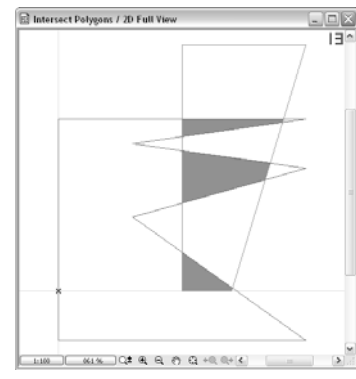


Re-combining the edges to form polygons yields the intersection of the two original polygons.



Save a copy of your *Add Polygons* object, and name it *Intersect Polygons*. Edit the script as shown below.

```
"#1 Intersect Polygons":
  nResults = 0
  !Add two polygons p1#1 and p2#1 and return the
  result in result[] []
  gosub "#2 Split Edges"
  !Remove any edge in polygon 1 that lies strictly
  outside of polygon 2
  pg#4 = 2
  for j#1 = 1 to pgonN[1]
    nxt#1 = j#1%pgonN[1] + 1
    xi#4 = (pgonX[1][j#1] + pgonX[1][nxt#1])/2
    yi#4 = (pgonY[1][j#1] + pgonY[1][nxt#1])/2
    gosub "#4 Polygon Contains Point"
    if isWithin#4 or isOnEdge#4 then
      includeEdge[1][j#1] = 1
    else
      includeEdge[1][j#1] = 0
    endif
  endif
```



```

-R1*cos(qi), R1*sin(qi), 79,
0, 180, 4079,
R1*cos(qi) + 2*R2*cos(qi), -
R1*sin(qi), 1079,
3*R1*cos(qi) + 2*R2*cos(qi),
R1*sin(qi), 1079,
-R1*cos(qi), R1*sin(qi), 1079
return

```

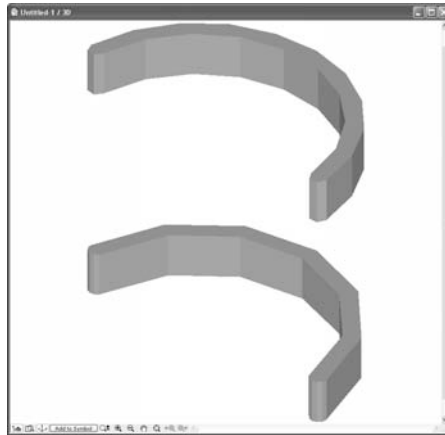
Click on the *3D View* button to view the 3D model you have created. If you examine the two prisms carefully, you will see that the lower one (created using *resol*) has a large number of polygons around the tight curves at the ends, and about the same number to define the large radius curve. The tight curves appear relatively smooth, while the large radius curves appear more fragmented.



The curve at the top, created using *toler*, is more uniform. Both the large and small radius curves are defined at a similar level of detail.

To see the difference more clearly, change the value of *resol* to 12, and the value of *toler* to 0.004.

The imbalance between the resolution of the small-radius and large-radius curves on the lower prism, is now more evident. The tighter curves still look quite smooth, while the large radius curves appear jagged and ungainly.



In contrast, the upper prism uses the same number of polygons to define its curved surface, but the effect is much more uniform.

A Rule of Thumb

Especially in cases where a single element includes a mixture of tight and wide curves, *toler* is preferable. This is also true when the exact dimensions of a curved element are parametrically defined. A cylinder should automatically gain resolution as its radius increases, and lose resolution as the radius decreases. This is very easily achieved using *toler*. As *ArchiCAD* is dedicated to producing architectural objects, a tolerance of 1mm is usually sufficient for every object. In some cases, you can increase the tolerance to 2 or 3mm.

Efficient development does not mean delivering a product quickly. In this type of development, the support overhead is multiplied by the sheer number of users. Support can become overwhelming, as many people are using the product. To avoid having to provide excessive support, it is essential that the product has been thoroughly tested, and that the user interface is as intuitive as possible.

Key benefits are what will motivate an *ArchiCAD* user to purchase the product. Two main categories of key benefits are productivity and design. On the one hand, Architects want to design freely without restrictions that may be imposed by the CAD software. The second stage of the process deals with documenting the design – in both cases, and for the construction project as a whole, greater efficiency means greater profits.

If the key benefits are exceptional, the balance may tip the other way, and the product may help motivate someone to purchase *ArchiCAD*.

The third factor, marketing, is outside of the control of the developer, and should be left to the experts.

Pro-Active Development

Rather than take on a project with the risk of failure, a percentage (perhaps 30-40%) of your time as a programmer should be dedicated to producing new resources, with an eye to specific future developments. Using this approach I will define an opportunity as *a project for which you possess all the required re-usable resources, and have the staff resources to deliver the product in the required time frame.*

You should also consider the nature of the new re-usable resources you will create. Will they in turn lead to further opportunity, or will they be seldom used? Some guidelines you might like to consider include:

- Develop new resources with an eye to future projects.
- Thoroughly test all resources before using them.
- Keep a list all the re-usable resources that you have, and that have been thoroughly tested. This is your re-usable resource pool.

The concept of resource is far wider ranging than mere fragments of programming code. All of the resources you need for programming need to be developed. This might include, for example, learning some mathematical strategies for solving problems, or reading a book like this one.

Required Resources

In order to identify an opportunity, you must list all of the resources you will need. To do this, the scope and nature of the project must be well defined.