Chapter 3

Fundamentals of SketchUp Scripting

Chapter Topics

- The three fundamental SketchUp data structures
- The Edge and Face classes
- Extruding three-dimensional shapes with the pushpull and followme methods

In this chapter, we're going to put aside programming theory and get to the fun stuff: creating SketchUp shapes from code. This presentation starts with one-dimensional lines and proceeds to two-dimensional surfaces. By the end of the chapter, you'll be about to form your own three-dimensional figures using the same Push/Pull and Follow Me mechanisms available in SketchUp.

Along the way, this chapter will explain how to create Ruby script files and execute them in SketchUp. So far, you've (hopefully) entered commands in SketchUp's Ruby Console and seen the results. But with scripts, you can store your commands in files instead of having to re-enter them every time. Also, as will be made apparent in future chapters, you can accomplish a great deal more with scripts than you can with individual commands.

This chapter is primarily concerned with SketchUp's `Edge` and `Face` objects—once you understand how these work, you can build just about anything. But before you can insert these objects into a design, you need to become familiar with three other data structures: the `Sketchup` module, the `Model` class, and the `Entities` class.

3.1 The Three Basic SketchUp Structures

Nearly every SketchUp script starts by accessing three basic data structures: `Sketchup`, `Model`, and `Entities`. Once you understand how they work and work together, you'll be ready to construct SketchUp designs in code.

The Sketchup Module

Unlike the classes and objects described in the preceding chapter, the first data structure we'll encounter, `Sketchup`, is a *module*. Chapter 8 discusses modules in great detail, but for now, all you need to know is that a module is a collection of methods. Most of the Ruby scripts in this book start by calling one of the methods in this module.

The methods in the `Sketchup` module access properties related to the SketchUp application as a whole. To see how this works, open the Ruby Console (Window > Ruby Console in the SketchUp main menu) and execute the following command:

```
Sketchup.version
```

This displays the version number of the current SketchUp application. You can also enter `Sketchup.os_language` to determine the current language or `Sketchup.get_locale` to access SketchUp's environment designator. To see all the methods provided by the `Sketchup` module, find the `Sketchup` listing in Appendix A or enter `Sketchup.methods` in the Ruby Console.

The most important method in the `Sketchup` module is `active_model`. This returns the `Model` object corresponding to the currently open SketchUp design. The following command shows how this method works:

```
mod = Sketchup.active_model
```

This retrieves the current `Model` object and then sets `mod` equal to the `Model` object. The `Model` object is crucial in all nontrivial SketchUp routines, and the following discussion explains why.

The Model Object

Just as the Sketchup module represents the entire SketchUp application, the `Model` object represents a single SketchUp file (*.skp), or more accurately, the design information contained in the file. When you open a new file in SketchUp, the properties of the `Sketchup` module remain the same but the data in the active `Model` object becomes entirely different.

The methods in the `Model` class provide information about the current design. For example, the `modified?` method identifies whether the current design has been changed since the last time it was saved. The following commands show how it is used:

```
mod = Sketchup.active_model

mod.modified?
```

The `title` method returns the title of the current design and the `description` method returns its text description. The `path` method returns the path to the file containing the current design. Appendix A lists all the methods associated with the `Model` class.

In this book, the methods of the `Model` class we'll be using most are those that access the *container objects* in the current design. You can think of a `Model` object like a chest of drawers: it's a single object containing multiple compartments, and each compartment may contain multiple objects. Figure 3.1 shows six of the most important object "drawers" contained within a `Model` object.

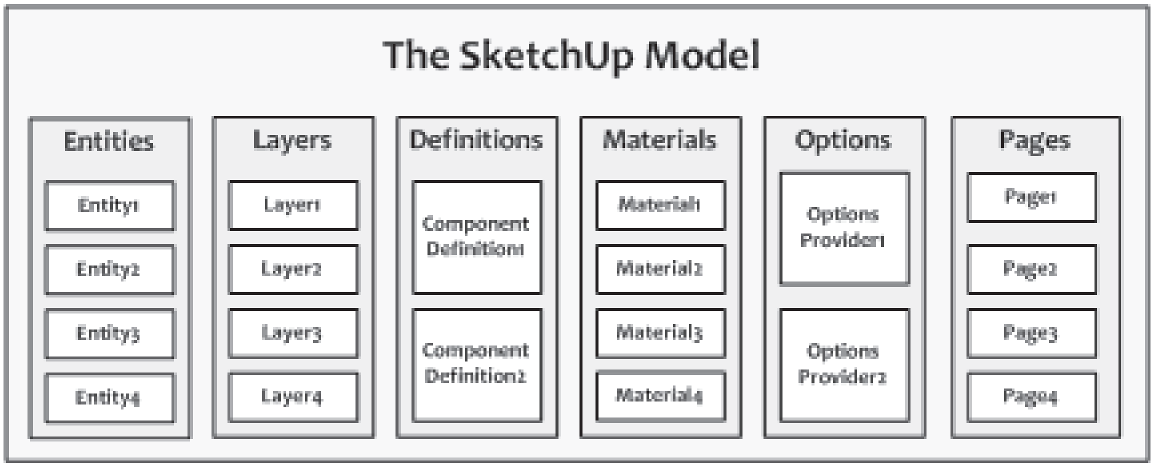


Figure 3.1: Content of a SketchUp Model Object (Abridged)

By modifying the objects in these six containers, you can configure nearly every aspect of a SketchUp design. To access them, you need to call the appropriate `Model` methods. These are:

1. `entities` - returns a `Entities` object that contains the shapes in the current design (introduced in this chapter and discussed throughout the book)
2. `layers` - returns a `Layers` object that contains the SketchUp layers in the current design (discussed in Chapter 7)
3. `definitions` - returns a `ComponentDefinitions` object holding all the component definitions in the current design (discussed in Chapter 7)
4. `materials` - returns a `Materials` object that manages the materials used in the current design (discussed in Chapter 6)
5. `options` - returns an `OptionManager` that provides access to multiple `OptionsProvider`s (discussed in Chapter 9)
6. `pages` - returns a `Pages` object that contains the pages of the current design (discussed in Chapter 10)

Most of these methods have plural names such as `entities`, `layers`, `definitions`, and `materials`. They return objects whose classes have similar names: `Entities`, `Layers`, `Definitions`, and `Materials`. Each plural object functions essentially like an array. As you might guess, an element of the `Entities` array is an `Entity`. Similarly, the `Layers` array contains `Layer` objects, the `Materials` array contains `Material` objects, and so on.

Note: This can be confusing, so pay close attention. A `Materials` object contains multiple `Material` objects. That is, a `Materials` object contains `Material`s. If the "s" is in code font, the container is being referenced. If the "s" is in regular font, the individual objects are being referenced.

At the moment, all we want to do is draw basic SketchUp shapes. To begin, the first step is to call the `entities` method of the `Model` class and access the `Entities` object of the current design. The following code shows how this is performed in code:

```
mod = Sketchup.active_model
ents = mod.entities
```

You can accomplish the same result by chaining the two methods together:

```
ents = Sketchup.active_model.entities
```

A large portion of this book is concerned with adding and modifying `Entity` objects within the current design's `Entities` container. This is discussed next.

The Entities Object

Every geometrical object in a SketchUp design is represented by an `Entity` or a subclass thereof, including lines, faces, images, text, groups, and components. To manage or modify `Entity` objects in a design, you need to access the design's primary `Entities` container. This container serves three main purposes:

1. Adds new `Entity` objects to the current SketchUp design
2. Moves, scales, rotates, and erases `Entity` objects in the design
3. Stores `Entity` objects in an array that can be accessed by index

The first role is the most important. The `Entities` class contains many methods that add new `Entity` objects to the current design. The simplest addition methods (`add_X`) are listed as follows:

- `add_line` - creates an `Edge` object from two points
- `add_edges` - forms an array of `Edge` objects from a series of points
- `add_circle` - forms an array of `Edge` objects that combine to form a circle
- `add_ngon` - forms an array of `Edge` objects that combine to form a polygon
- `add_face` - creates a `Face` object from edges or points
- `add_text` - adds a label to the design at a given point

When it comes to shapes, the `Edge` and `Face` are the most important of the `Entity` objects. Each time you add one to the `Entities` container, a corresponding shape appears in the SketchUp window. Then, when you save the design, the `Edge`s and `Face`s will be included in the stored `Model` object. To fully understand the `Edge` and `Face` classes, you need to be familiar with their superclasses, `Entity` and `Drawingelement`.

3.2 The Entity and Drawingelement Classes

The `Entity` class is the superclass for all drawable shapes in SketchUp. Figure 3.2 shows the hierarchy of `Entity` subclasses that will be discussed in this book.

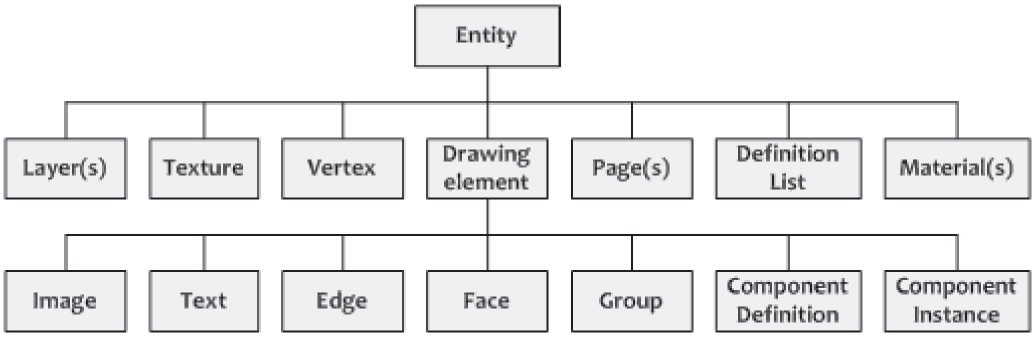


Figure 3.2: The Entity Class Hierarchy (Abridged)

The primary subclass of `Entity` is `Drawingelement`, which is the superclass of many of the classes we'll be studying in this and future chapters. Before continuing further, we'll take a brief look at these two important superclasses.

The Entity Class

In Figure 3.2, the `Entity` class sits at the top of the hierarchy. The methods contained in this class are available to every subclass beneath it. Many of these methods provide basic information about the `Entity`, and they include the following:

- `entityID` - returns a unique identifier for the `Entity`
- `typename` - identifies the geometric type of the `Entity` (`Edge`, `Face`, etc.)
- `valid?/deleted?` - identifies whether the `Entity` can still be accessed
- `model` - returns the design's `Model` object

The following commands show how these methods are used in code:

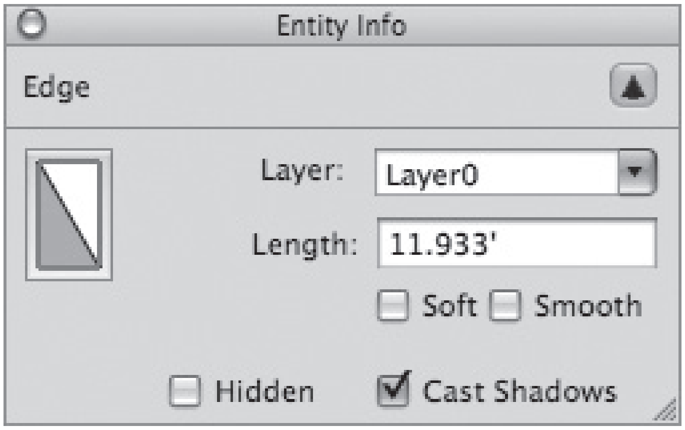
```
test_line = Sketchup.active_model.entities.add_line [0,0,0], [1,1,1]
       → #<Sketchup::Edge:0x767be50>
test_line.typename
       → Edge
test_line.entityID
       → 1895
```

In addition to these methods, each `Entity` can access user-specified information by invoking the `attribute_dictionaries` method. The `Entity` can then retrieve, modify, or delete these attributes. Chapter 9 discusses the usage and operation of the `AttributeDictionaries` class in detail.

Lastly, an `Entity` may have one or more `EntityObserver` objects associated with it. Observers monitor the state of the `Entity` and respond to changes. Chapter 9 explains how observers work and how to create them in code.

The Drawingelement Class

The `Drawingelement` class is the superclass of the `Edge`, `Face`, `Group`, `Image`, `Text`, `ComponentDefinition`, and `ComponentInstance` classes. Many of the methods in this class control how the element appears in SketchUp, and set properties such as shadowing, material composition, and whether the element is hidden or visible. These are the same properties defined by SketchUp's Entity Info dialog, shown in Figure 3.3.



Layer: Layer0
Length: 11.933'
☐ Soft ☐ Smooth
☐ Hidden ☑ Cast Shadows

Figure 3.3: The Entity Info Dialog

The `Drawingelement` class provides a useful method called `bounds` that returns a `BoundingBox` object. This represents the smallest rectangular volume that can contain the `Drawingelement` while keeping its sides aligned with the x, y, and z axes. The following commands create a `BoundingBox` for a line drawn from [0, 0, 0] to [2, 2, 5].

```
new_line = Sketchup.active_model.entities.add_line [0,0,0], [2,2,5]
      → #<Sketchup::Edge:0x767ab18>
box = new_line.bounds
      → #<Geom::BoundingBox:0x76784d8>
```

Once the `BoundingBox` is obtained, its methods provide information about its dimensions, diagonal, corners, and maximum/minimum points. The following commands display the location of the center of a `BoundingBox` and the length of its diagonal (in inches).

```
box.center
       → Point3d(1, 1, 2.5)
box.diagonal
       → 5.74456264653803
```

`BoundingBox` objects are useful when you need to determine if the user clicked on a shape. Once you've acquired the `BoundingBox` of a shape, you can compare its dimensions with the position of the user's mouse click.

3.3 Edges

Of the many `Drawingelement` classes, the easiest to understand is the `Edge`, which represents a line segment between two points. `Edge` objects are created with the `add_line` method of the `Entities` class, followed by the locations of the line's start and end points. When you invoke this method, SketchUp draws a line between the two points and adds an `Edge` to the current `Entities` collection. For example, the following command creates an `Edge` that extends from [5, 0, 0] to [10, 0, 0]:

```
Sketchup.active_model.entities.add_line [5, 0, 0], [10, 0, 0]
```

Most of the methods in the `Edge` class fall into one of two categories:

1. Methods that configure the `Edge`'s appearance
2. Methods that access objects connected to the `Edge`

The methods in the first category configure the `Edge`'s visibility in the design window. In addition to the `hidden` method provided by the `Drawingelement` class, `Edge` provides `soft` and `smooth` methods. It's important to remember the difference between a hidden line and a smooth line: a smooth line combines adjacent surfaces into a single (usually curved) surface, while a hidden line doesn't alter adjacent surfaces.

In the second category, the `all_connected` method returns an array of all `Entity` objects connected to the `Edge`. Similarly, the `faces` method returns an array containing the `Face` objects connected to the `Edge`.

In SketchUp, the endpoints of an `Edge` are represented by `Vertex` objects. The `Edge` class contains a number of methods that interact with them:

- `vertices` - returns an array of the `Edge`'s two `Vertex` objects
- `start`/`end` - returns the `Edge`'s starting/ending `Vertex` objects
- `other_vertex` - given one of the `Edge`'s `Vertex` objects, this method returns the other
- `used_by?` - identifies whether a `Vertex` is connected to the `Edge`

There are two methods in the `Edge` class that don't fall into either category: `length` and `split`. The first returns the length of the line segment corresponding to the `Edge`. The second accepts a point on the line and creates a second `Edge` object. After `split` is called, the first `Edge` object continues only up to the given point and the second `Edge` object continues from the point to the end of the original line. The following commands show how these methods are invoked in practice:

```
line = Sketchup.active_model.entities.add_line [0, 0, 0], [6, 3, 0]

line.length

      → 6.70820393249937

new_line = line.split [4, 2, 0]

line.length

      → 4.47213595499958

line.start.position

      → Point3d(0, 0, 0)

line.end.position

      → Point3d(4, 2, 0)

new_line.length

      → 2.23606797749979

new_line.start.position
```

```
      → Point3d(4, 2, 0)
new_line.end.position
      → Point3d(6, 3, 0)
```

Figure 3.4 graphically depicts the results. The original `Edge` object runs from [0, 0, 0] to [6, 3, 0]. After `split` is invoked, the original `Edge` object runs from [0, 0, 0] to [4, 2, 0] and the new `Edge` object runs from [4, 2, 0] to [6, 3, 0].

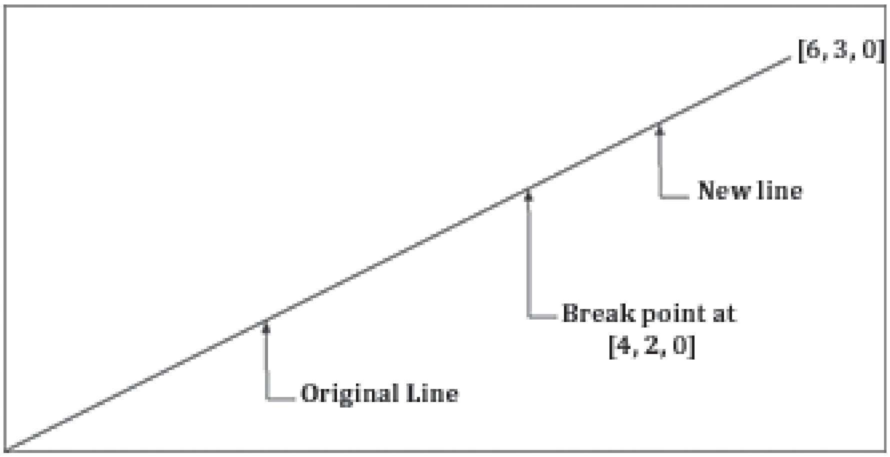


Figure 3.4: Splitting an Edge

3.4 Introduction to SketchUp Scripting

At this point, you should be an expert at using the SketchUp Ruby Console. The console is fine for executing simple Ruby commands, but it becomes tiresome when you need to perform complicated procedures. Rather than enter commands manually, it's much more efficient to group them in files and tell SketchUp to execute all of them in sequence. These files are called *scripts*.

Besides efficiency, there are many other reasons to use scripts, and the rest of this book will provide example code listings in script form. One of the advantages of scripts is that you can add human-readable notes that explain or describe aspects of the code.

Ruby Scripts

In general programming, a script is a file containing commands that control a running program. UNIX® scripts direct commands to the shell program, such as Bash. In Windows, scripts written in VBScript® interact with the operating system and run programs like DOS® utilities.

All the scripts discussed in this book are directed toward SketchUp. You can write a SketchUp script in any text editor you choose, but keep three points in mind:

1. Commands in a SketchUp script must be written in the Ruby programming language.
2. Unencrypted script files must have names that end with the .rb suffix. Names of encrypted script files must end with .rbs.
3. To execute a script, it must be loaded into SketchUp.

This last point is important. One of the most useful Ruby commands to know is `load`. When executed in the console, `load` tells SketchUp to read a script and execute each of its commands in sequence. Listing 3.1 provides us with our first script example.

Listing 3.1: star.rb

```
# Access the current Entities object
ents = Sketchup.active_model.entities

=begin
Create five points in three dimensions
Each point is the vertex of a star shape
=end
pt1 = [0, 1, 0]
pt2 = [0.588, -0.809, 0]
pt3 = [-0.951, 0.309, 0]
pt4 = [0.951, 0.309, 0]
pt5 = [-0.588, -0.809, 0]

# Draw five lines in a star pattern
```

```
ents.add_line pt1, pt2
ents.add_line pt2, pt3
ents.add_line pt3, pt4
ents.add_line pt4, pt5
ents.add_line pt5, pt1
```

The example code for this book can be downloaded from **http://www.autosketchup.com**, and it consists of folders with names such as Ch3, Ch4, and Ch5. If you haven't already done so, I strongly recommend that you download this example code and extract it to SketchUp's top-level plugins folder. The location of this folder depends on your operating system:

- In Windows, the plugins folder is commonly located at C:/Program Files/Google/Google SketchUp 7/Plugins
- In Mac OS X, the plugins folder is commonly located at /Library/Application Support/Google SketchUp 7/SketchUp/plugins

If you place the example code directories directly inside the plugins folder (plugins/Ch3, plugins/Ch4, etc.), you can easily execute the star.rb script in Listing 3.1. In SketchUp, open the Ruby Console Window and enter the following command:

```
load "Ch3/star.rb"
```

This command tells SketchUp to access the star.rb script file and execute each of its commands. This creates a star shape in the x-y plane, centered around the origin. Figure 3.5 shows what the resulting shape looks like.

If you can't see the star pattern clearly, open the Camera menu in SketchUp and choose Standard Views and Top. Then, still in the Camera menu, select Parallel Projection. Lastly, click the Zoom Extents tool to focus on the new drawing.

You don't have to place your scripts in the plugins directory. To access a script outside this folder, call `load` with the script's full directory location. For example, if you saved the example code to C:/ruby_scripts, you can load star.rb with the following command:

```
load "C:/ruby_scripts/Ch3/star.rb"
```

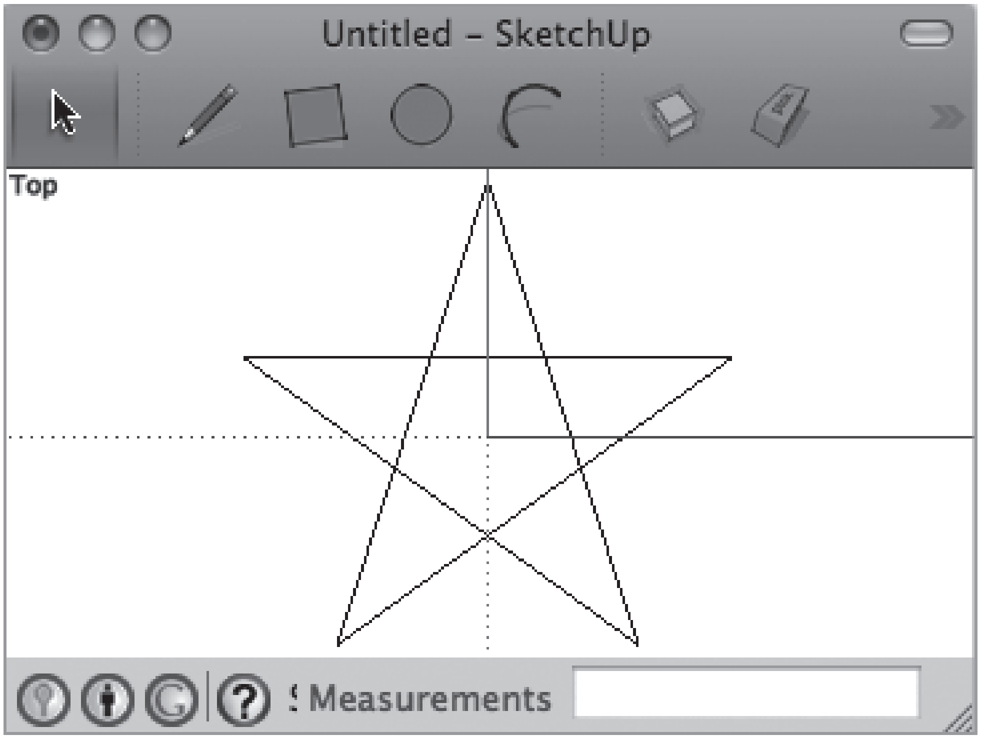


Figure 3.5: Shape Created by star.rb

The rest of the code in this book falls into one of two categories: command examples and code listings. Command examples are sets of commands that demonstrate simple concepts. They can be run directly from the Ruby Console Window.

Code listings, such as Listing 3.1, are contained in script files within one of the chapter directories. Each script file can be executed by running the `load` command in the Ruby Console Window.

Ruby Comments

The code in Listing 3.1 contains an aspect of Ruby that we haven't encountered before: *comments*. A comment is a note placed inside the script that describes what the code is doing. Ruby comments come in two types:

1. Single-line comment - Starts with `#` and continues until the end of the line
2. Multi-line comment - Starts with `=begin` and continues until `=end`

Any code placed in a comment will not be processed by the Ruby interpreter. Therefore, you can write anything you like in a comment so long as it doesn't terminate the comment prematurely.

There are three comments in star.rb. The first comment,

```
# Access the Entities object
```

explains the purpose of the first line of code. The second comment,

```
=begin
Create five points in three dimensions
Each point is the vertex of a star shape
=end
```

describes the purpose of the five variable declarations. The last comment,

```
# Draw five lines in a star pattern
```

explains what the last five lines accomplish. Frequent commenting is a vital aspect of professional programming—not just so others can read your script, but also to remind yourself how you intended your code to work.

3.5 Vectors

Before we discuss methods that draw curves, you need to have a basic understanding of vectors. The term vector has different meanings depending on whether you're discussing mathematics, physics, or engineering. As SketchUp designers, we use vectors mainly to identify direction. For this reason, we represent vectors graphically with arrows. Figure 3.6 depicts a number of vectors in a three-dimensional space.

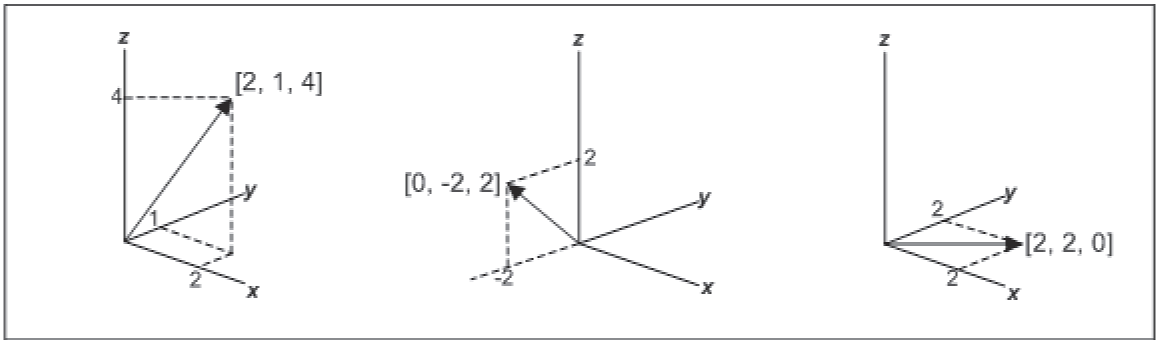


Figure 3.6: Vectors and their Three-Dimensional Components

In code, a vector is identified by three numeric values. The first identifies how much of the vector points in the x-direction, the second identifies how much the vector points in the y-direction, and the third identifies how much of the vector points in the z-direction. These values are called the vector's *components*. If a vector is represented by [a, b, c], a is the x-component, b is the y-component, and c is the z-component.

A normal vector is a specific kind of vector that identifies the direction a shape is facing. For example, if you draw a circle in the x-y plane and you want it to face upward, an acceptable normal vector is [0, 0, 1]. Here, only the z-component has a nonzero value, and this value is positive. Therefore, the vector points in the positive z-direction, or upward. If the normal vector is set to [0, 0, –1], the circle points in the negative z-direction, or downward.

A question arises. We didn't need vectors to draw the star in Listing 3.1, so why do we need vectors to draw curves? The answer is that the lines that formed the star are one-dimensional. They have length, but no area. Once you select a starting point and an ending point, each line is uniquely identified.

But if all you have is the center and radius of a circle or arc, you can not uniquely identify the curve. For example, an infinite number of circles that can be drawn with a given center and nonzero radius. This is shown in Figure 3.7.

To distinguish between the circles, you could identify multiple points on the circle. But it's easier to define the circle's normal vector. If you walk around the circle, the normal vector points in the up direction. Similarly, if you use SketchUp's Push/Pull tool to create a cylinder from the circle, this is the direction you'd pull in.

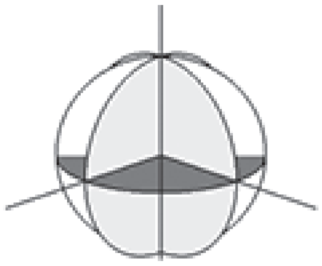


Figure 3.7: Concentric Circles with Equal Radii, Unequal Normal Vectors

3.6 Edge Arrays: Curves, Circles, Arcs, and Polygons

You can use SketchUp's Arc and Circle tools to draw shapes, but you're not really creating arcs and circles. Instead, each curve-like shape is a succession of tiny line segments. In code, the `Entities` class contains three methods that produce curve-like shapes, and each of them returns an array of `Edge` objects. The three methods are `add_curve`, `add_circle`, and `add_arc`. These methods are similar to a fourth method, `add_ngon`, which draws polygons.

Curves

The simplest of the curve-creation methods is `add_curve`. This accepts a succession of points and returns the array of `Edge` objects that connect the points. For example, the following commands produce the same star shape as the one created in the previous chapter.

```
pt1 = [0, 1, 0]
pt2 = [0.588, -0.809, 0]
pt3 = [-0.951, 0.309, 0]
pt4 = [0.951, 0.309, 0]
pt5 = [-0.588, -0.809, 0]
curve = Sketchup.active_model.entities.add_curve pt1, pt2, pt3,
   pt4, pt5, pt1
```

```
curve.class
      → Array
curve.length
      → 5
```

Here, the `add_curve` method produces an array of five `Edge` objects, which means the result is really a polyline instead of a curve. The more points you designate, the more closely the polyline will resemble a rounded curve. Usually, it's easier to create rounded curves with the `add_circle` or `add_arc` methods.

Circles

The `add_circle` method creates a circle with a given center, normal vector, and radius. The normal vector identifies the up direction—if someone walked on the perimeter of the circle and pointed upward, they would be pointing in the direction of the normal vector. As an example, the following command creates a circle with a center at [1, 2, 3], a normal vector equal to [4, 5, 6], and a radius of 7:

```
circle = Sketchup.active_model.entities.add_circle [1, 2, 3],
   [4, 5, 6], 7
circle.class
      → Array
circle.length
      → 24
circle[0].class
      → Edge
```

The `add_circle` method creates an array of 24 `Edge` objects, which amounts to one segment for every 15°. Twenty-four is the default number of segments in a circle, regardless of radius or orientation. However, you can add a fourth parameter to the `add_circle` method that customizes the number of segments in the circle.

For example, the following command creates a circle with 72 segments:

```
circle = Sketchup.active_model.entities.add_circle [1, 2, 3], [4, 5, 6],
        7, 72
```

Figure 3.8 shows how a circle's appearance is affected by the number of segments. The 72-segment circle more closely approximates a real circle than the 24-segment circle or the 12-segment circle, but operations on the circle's `Edge`s require more time.

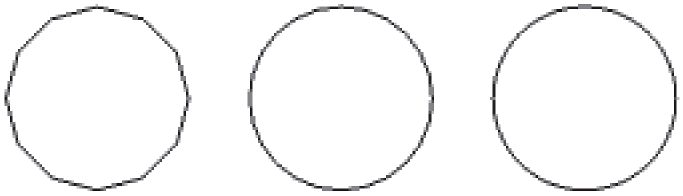


Figure 3.8: Circles with 12, 24, and 72 Segments

The fewer segments your circle contains, the more it will look like a polygon. If you specify a segment count of five, `add_circle` will create a pentagon. However, it's more common to invoke the `add_ngon` method to create polygons. This is discussed next.

Polygons

The `add_ngon` method is almost exactly like `add_circle`, and the two methods produce essentially the same shapes. The only difference is the segment count: with `add_circle`, the default number of segments is 24, but you can optionally set a different number of segments. The `add_ngon` method has no default segment count—you always have to designate the number of sides of your polygon.

To make this clear, the script in Listing 3.2 produces four shapes: a polygon with 8 sides, a circle with 8 sides, a polygon with 24 sides, and a circle with 24 sides.

Listing 3.2: poly_circle.rb

```
ents = Sketchup.active_model.entities
normal = [0, 0, 1]
radius = 1

# Polygon with 8 sides
ents.add_ngon [0, 0, 0], normal, radius, 8

# Circle with 8 sides
ents.add_circle [3, 0, 0], normal, radius, 8

# Polygon with 24 sides
ents.add_ngon [6, 0, 0], normal, radius, 24

# Circle with 24 sides
ents.add_circle [9, 0, 0], normal, radius
```

To execute this script, make sure the Ch3 folder is in the plugins folder of your top-level Sketchup directory. Then open the Ruby Console and execute the following command:

```
load "Ch3/poly_circle.rb"
```

To see the resulting shapes, open the Camera menu item and choose Parallel Projection. Then go to Camera > Standard Views and select Top. The result is shown in Figure 3.9.

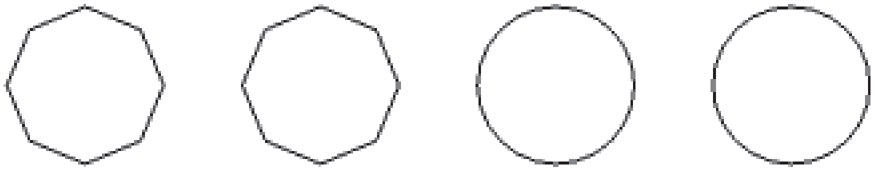


Figure 3.9: Polygons and Circles

Arcs

Creating an arc is similar to creating a circle, but additional parameters are necessary. First, you have to specify the starting and ending angles. These angles must be measured (in radians) from an axis, so you need to identify the vector that serves as the zero-radian direction. The full list of parameters for `add_arc` are as follows:

- `center` - A point identifying the center of the circular arc
- `zero_vec` - A vector identifying the direction of the zero angle
- `normal` - A vector perpendicular to the circular arc
- `radius` - Radius of the circular arc
- `start_angle` - Starting angle, measured from the `zero_vec` vector
- `end_angle` - Ending angle, measured from the `zero_vec` vector
- `num_segments` (Optional) - Number of line segments in the arc

The following command creates an arc centered at [0, 0, 0] that intercepts an angle from 0° to 90°. The angle is measured from the y-axis, so the vector at 0° is [0, 1, 0]. The arc has a radius of 5 and lies in the x-y plane, so its normal vector is [0, 0, 1]. The number of segments is left to its default value.

```
arc = Sketchup.active_model.entities.add_arc [0,0,0], [0,1,0],
   [0,0,1], 5, 0, 90.degrees
arc.length
      → 6
```

This 90° arc is composed of six `Edge` objects, which means the arc contains one `Edge` for every 15°, just as with the circle. More `Edge`s can be added by adding an optional parameter to the `add_arc` method.

Appendix B presents a method for creating an arc from three ordered points. This is more convenient than the `add_arc` method, which requires that you know both the center and radius of the arc's enclosing circle.

3.7 Creating Figures in Three Dimensions

Now that you've created lines and curves, you're ready to start constructing faces. These are the closed two-dimensional surfaces that SketchUp fills in with color. Once you've created a `Face` object, it's easy to extrude it into a three-dimensional volume using mechanisms similar to SketchUp's Push/Pull and Follow Me tools.

Constructing a Face

`Face` objects are created by the `add_face` method of the `Entities` class. This is similar to the `add_curve` method described earlier: it accepts a series of points or a series of `Edge`s, and either can be provided in a comma-separated list or in an array. For example, the pentagon.rb script in Listing 3.3 constructs a pentagonal `Face` from five points:

Listing 3.3: pentagon.rb

```
# Create the five points of the pentagon
pt1 = [0, 1, 0]
pt2 = [-0.951,  0.309, 0]
pt3 = [-0.588, -0.809, 0]
pt4 = [ 0.588, -0.809, 0]
pt5 = [ 0.951,  0.309, 0]

# Draw the face
pent = Sketchup.active_model.entities.add_face pt1, pt2, pt3,
   pt4, pt5

# Display the locations of the stored vertices
puts "Point 0: " + pent.vertices[0].position.to_s
puts "Point 1: " + pent.vertices[1].position.to_s
puts "Point 2: " + pent.vertices[2].position.to_s
puts "Point 3: " + pent.vertices[3].position.to_s
puts "Point 4: " + pent.vertices[4].position.to_s
```

Figure 3.10 shows what the result looks like in the SketchUp window. Unlike the arrays of `Edge`s displayed earlier, the entire surface of a `Face` object is filled in.

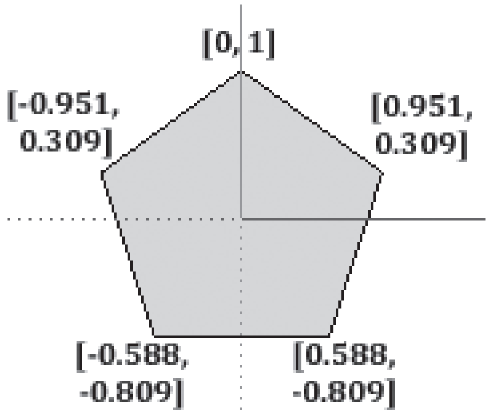


Figure 3.10: Pentagonal Face Created by pentagon.rb

Leaving out the dimensions, the printed results are as follows:

```
Point 0: (-0.951, 0.309, 0)

Point 1: (0, 1, 0)

Point 2: (0.951, 0.309, 0)

Point 3: (0.588, -0.809, 0)

Point 4: (-0.588, -0.809, 0)
```

When you define points to create a `Face`, order is important. If you switch `pt2` and `pt3` in the parameter list, the shape won't look anything like a pentagon. However, the orientation of the points (clockwise or counter-clockwise) is not important. Whether the points are listed from `pt1` to `pt5` or from `pt5` to `pt1`, the resulting `Face` object will be the same.

To make this clear, look at the output of pentagon.rb. You can see that the `Face` object stores its points in an entirely different order than that with which they were listed. In this case, the add_face method arranges the points in a clockwise orientation, and the normal vector always points down. Returning to the pentagonal Face, this can be shown with the following code:

```
pent.normal
       → (0, 0, -1)
pent.reverse!
pent.normal
       → (0, 0, 1)
```

The normal vector of a Face determines its direction of extrusion. For example, if the normal vector points in the –z direction, the pushpull method will pull it downward. Many designs expect three-dimensional shapes to grow upward, so you may need to invert the normal vector by invoking the reverse! method.

Geometric Methods of the Face Class

Once you've created a Face, you can invoke its methods to examine its properties or extrude it into a three-dimensional figure. Some Face methods deal with materials and textures, and they will be explored in a later chapter. Right now, we'll look at the methods that analyze the Face's geometric properties.

Most of the methods of the Face class provide information about the nature of its shape: the edges method returns an array of Edge objects that form the Face and vertices returns an array of the Vertex objects on the Face's boundary. The area method returns the area of the face and normal returns its normal vector.

The classify_point method accepts a point and identifies where the point is located relative to the Face. This is useful when you need to detect collisions or determine which surface the user clicked. The method returns one of six values:

- 0 - Unknown point
- 1 - Point inside the face
- 2 - Point on one of the face's edges
- 4 - Point is one the face's vertices

- 8 - Point on the plane containing the face, but not on the face
- 16 - Point not on the plane containing the face

The following commands create a square `Face` centered on the origin, and show how the `classify_point` method locates points relative to it:

```
face = Sketchup.active_model.entities.add_face [-1, -1, 0], [-1, 1, 0],
   [1, 1, 0], [1, -1, 0]
face.classify_point [0, 0, 0]
      → 1
face.classify_point [1, 1, 0]
      → 4
face.classify_point [1, 2, 0]
      → 8
face.classify_point [1, 1, 1]
      → 16
```

The `outer_loop` method of the `Face` class returns a `Loop` object containing the `Face`'s edges. The `loops` method returns an array of `Loop`s adjacent to the `Face`. `Loop` objects are useful in topology, but won't be explored in this book.

The pushpull Method

The `Face` class provides two methods that extrude three-dimensional figures from two-dimensional surfaces: `pushpull` and `followme`. They perform the same operations as SketchUp's Push/Pull and Follow Me tools.

The `pushpull` method is simple to use and understand. It accepts an integer, and if the integer is positive, the method pulls the surface along the `Face`'s normal vector, creating a three-dimensional figure. If the number is negative, the method pushes the surface in the direction opposite the `Face`'s normal vector. If a surface of a three-dimensional figure is pushed all the way to the rear of the figure, the volume is cut away from the design.

To remove part of a three-dimensional figure, create one or more `Edge` objects that bound the portion to be removed. This bounded portion becomes a new `Face`. Invoke `pushpull` on this `Face` with a negative value to cut it away from the original figure.

The cutbox.rb script in Listing 3.4 shows how this works in practice. It starts with a rectangular Face and extrudes it to form a 3-D box. Then it draws a line across the upper-right corner and calls `pushpull` to remove the corner from the box.

Listing 3.4: cutbox.rb

```
# Create the box
ent = Sketchup.active_model.entities
main_face = ent.add_face [0,0,0], [6,0,0], [6,8,0], [0,8,0]
main_face.reverse!
main_face.pushpull 5

# Draw a line across the upper-right corner
cut = ent.add_line [6,6,5], [4,8,5]

# Remove the new face
cut.faces[1].pushpull -5
```

The last command is worth examining closely. The second `Face` in the corner isn't explicitly created in code. Instead, it's constructed automatically when the new `Edge` is drawn across the corner of `face`. Once drawn, this `Edge` is connected to two `Face`s: one representing the main face and one representing the corner face. Each `Edge` can access an array of its adjacent `Face` objects, and the new `Face` is at index 1. Therefore, the command

```
cut.faces[1].pushpull -5
```

pushes the second face downward, removing the corner volume from the figure.

The followme Method

When you call `pushpull`, you can extrude in one of only two directions: the direction of the `Face`'s normal vector or the opposite direction. With `followme`, the extrusion is still performed along a vector, but now you control the vector's direction. That is, you specify the path the extrusion should take.

The extrusion path can be defined with one or more `Edge`s. If the path contains more than one `Edge`, two requirements must be met:

1. The `Edge`s that form the extrusion path must be connected.
2. The `Edge`s that form the extrusion path must not all lie in the same plane as the plane containing the surface to be extruded.

Once you've determined the path, you can invoke `followme` with an array of `Edge`s. This will extrude the `Face` along each `Edge` of the extrusion path.

The followme.rb script in Listing 3.5 shows how this works. It creates a circular `Face` and extrudes it along a rectangular loop.

Listing 3.5: followme.rb

```
# Access the Entities container
model = Sketchup.active_model
ent = model.entities

# Create the primary face
circle = ent.add_circle [0,0,0], [0,0,1], 2
circle_face = ent.add_face circle

# Create the path
path = ent.add_curve [10,0,0], [10,0,5], [10,5,5],
   [10,5,0], [10,0,0]

# Extrude the circle along the path
```

```
circle_face.followme path
```

The left side of Figure 3.11 shows the original Face and the rectangular extrusion path. The right side shows the three-dimensional extrusion created by `followme`.

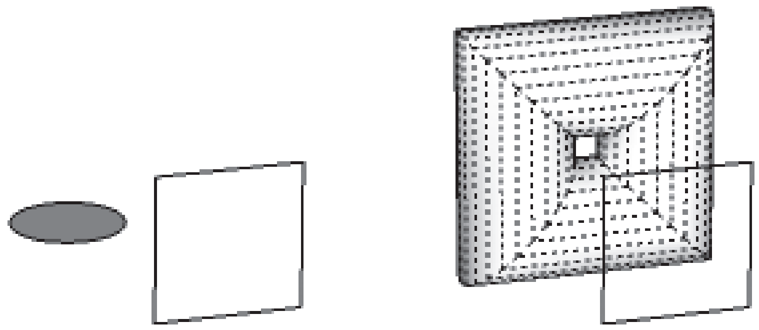


Figure 3.11: Extrusion Created by followme.rb
(dotted lines added for clarity)

Let's look at another usage of `followme`. The code in Listing 3.6 creates a lathed figure by rotating a `Face` around an axis. Here, the `Face` is constructed from the array of points returned by the `add_curve` method.

Listing 3.6: lathe.rb

```
# Access the Entities object
model = Sketchup.active_model
ents = model.entities

# Create the 2-D shape
curve = ents.add_curve [0, 0, 1.244209], [0.116554, 0, 1.238382],
   [0.160261, 0, 1.217985], [0.186486, 0, 1.188846],
   [0.1894, 0, 1.165536], [0.17483, 0, 1.145139],
   [0.142778, 0, 1.127656], [0.096157, 0, 1.118914],
   [0.093243, 0, 1.063551], [0.175152, 0, 0.996269],
   [0.175152, 0, 0.915269], [0.28237, 0, 0.871026],
   [0.375392, 0, 0.801741], [0.448486, 0, 0.711683],
   [0.497151, 0, 0.606398], [0.51839, 0, 0.492371],
   [0.510894, 0, 0.376625], [0.475126, 0, 0.26629],
   [0.413287, 0, 0.168161], [0.329188, 0, 0.088283],
   [0.228007, 0, 0.031575], [0.115978, 0, 0.001531],
   [0, 0, 0], [0, 0, 1.244209]
curve_face = ents.add_face curve

# Create the circular path
path = ents.add_circle [0, 0, 0], [0, 0, 1], 2

# Create the figure
curve_face.followme path
```

The left side of Figure 3.12 shows the two-dimensional half-bottle `Face` and its circular path. The right side shows the lathed figure produced by `followme`.

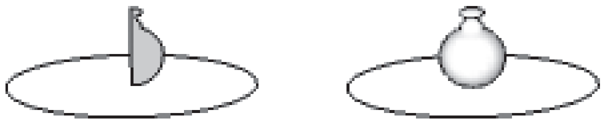


Figure 3.12: The Lathed Figure Created by lathe.rb

Like `pushpull`, the `followme` method can remove portions of a three-dimensional figure. To do this, form a `Face` on the figure by creating one or more `Edge` objects. Then form the path by choosing which `Edge`s of the figure should be cut away. Invoke `followme` with the path to remove the bounded portion.

The code in Listing 3.7 shows how this works. It creates a three-dimensional box and calls `followme` to cut away the box's top edges.

Listing 3.7: chamfer.rb

```
# Create the box
ents = Sketchup.active_model.entities
main_face = ents.add_face [0,0,0], [5,0,0], [5,8,0], [0,8,0]
main_face.reverse!
main_face.pushpull 6, true

# Draw a line across a corner
cut = ents.add_line [5, 7, 6], [5, 8, 5]

# Create the chamfer
cut.faces[0].followme main_face.edges
```

The left side of Figure 3.13 shows the box with the `Edge` drawn across its upper corner. The right side shows the chamfered box after `followme` is invoked.

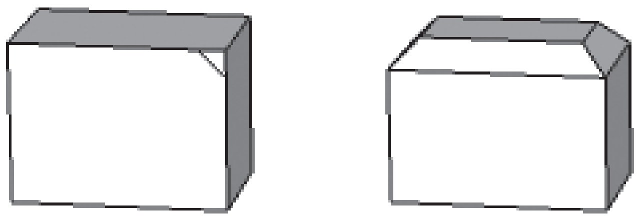


Figure 3.13: Using followme to Create a Chamfered Figure

The cutting path is obtained by calling the `edges` method on the top `Face` of the box. This returns the array of connected `Edge` objects that bound the top face. When you need to apply `followme` to every edge of a surface, its easier to call the `edges` method than to locate individual `Edge` objects.

In Listing 3.7, the `pushpull` method is followed by a second parameter set to `true`. Normally, `pushpull` deletes the `Face` used for the extrusion. But this optional argument ensures that `main_face` will remain accessible after `pushpull` is called. Remember this if you receive any "reference to deleted Face" errors in your scripts.

Creating a Sphere

Everyone should know how to create a sphere in SketchUp, and like the preceding examples, it can be easily accomplished with the `followme` method. In this case, the surface and the extrusion path are both circles with the same center. This is shown in Figure 3.14.

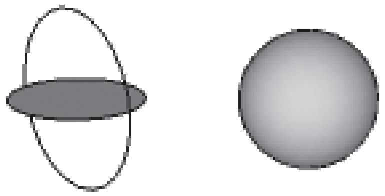


Figure 3.14: Extruding a Sphere

The code in Listing 3.8 creates the circular `Face` and the circular path, and then invokes `followme` to extrude the sphere. Note that, while both circles have the same center, their normal vectors are perpendicular to one another.

Listing 3.8: sphere.rb

```
# Access the Entities object
ents = Sketchup.active_model.entities

# Create the initial circle
center = [0, 0, 0]
radius = 5
circle = ents.add_circle center, [0, 0, 1], radius
circle_face = ents.add_face circle

# Create the circular path
path = ents.add_circle center, [0, 1, 0], radius + 1
```

```
# Create the sphere
circle_face.followme path

# Remove the path
ents.erase_entities path
```

The last line of the script removes the `Edge`s that form the extrusion path. This is an important consideration when you use the `followme` method.

3.8 Conclusion

This chapter has presented a great deal of information, from SketchUp's basic data structures to the objects that represent shapes in a design. The first part of this chapter discussed the `Sketchup` module, whose methods provide information about the SketchUp application. In contrast, the `Model` class represents only the current design, which may be saved to a *.skp file. A `Model` object serves as a container of containers: it contains objects such as `Entities`, `Materials`, `Layers`, `Tools`, and so on.

The `Entities` container is particularly important because it stores every graphical object in the current design. These graphical objects are called `Entity` objects, and the `Entity` class has a number of important subclasses. One subclass, `Drawingelement`, serves as the superclass of every shape in the SketchUp API.

The last part of this chapter discussed the actual shapes that make up SketchUp models: `Edge`s, `Edge` arrays, `Face`s, and three-dimensional figures. The classes and their methods are easy to understand, but it can be difficult to coordinate them to form a design. Therefore, I recommend that you practice creating these objects in scripts: form `Face`s from `Edge`s and use `pushpull` and `followme` to extrude the `Face`s into three-dimensional figures.