1 This stub represents chapter 1

1.1
2 Introduction to 2D Modeling with Windows Presentation Foundation (WPF)  
Draft 10 August 2009

2.1 THE EVOLUTION OF GRAPHICS PLATFORMS

In chapter 1 we saw that the graphics software system, also called here a graphics platform, is an intermediary between the application and the display hardware, responsible for getting the GPU to display information on the screen and causing an appropriate callback in the application to be invoked when an interaction component is activated by the user. Graphics platforms have been going through the same evolution from low-level to high-level that programming languages and development platforms have experienced. The progression from low-level assembly language to early procedural programming languages (FORTRAN, COBOL, PL/I, Pascal, C), and then to more recent object-oriented languages (C++, Java, C# all of which owe a debt to Simula and SmallTalk from the 60s and 70s respectively) – not to mention more specialized languages such as LISP/Scheme, MATLAB, etc. – has been well publicized in the domain of software development platforms, but the parallel progression in the domain of graphics platforms has been equally profound, though not as well publicized.

In both domains, this evolution has moved both the specification and tedious bookkeeping burdens from the programmer to ever more powerful computing platforms. Both for programming languages and graphics platforms (a.k.a. graphics libraries, packages, or APIs), this is accomplished by operating at a higher level of abstraction that transfers many of the implementation details to the platform software. As we will see, this allows the graphics programmer to focus far more on what is to be accomplished, than on the more tedious details of how to accomplish it. As always, when one writes programs at a higher level of abstraction, there may be performance penalties due to the additional overhead; conversely, by having more information available, the platform may be able to do a good job of optimization. Such tradeoffs between higher levels of abstraction (and the concomitant loss of explicit control over implementation details) and performance will be reconsidered in section [??].

2.1.1 Immediate-mode, Integer-coordinate Raster Graphics

In the previous edition of this textbook, the corresponding chapter on 2D graphics introduced a simple instructional raster graphics package (SRGP) that could be considered the equivalent of an assembly language for the creation of 2D images. SRGP provides the ability to paint pixels on a rectangular canvas, using an integer coordinate system that maps 1-to-1 with screen pixels. The application paints a scene by calling procedures that draw geometric primitives (a.k.a. primitive shapes) such as polygons, ellipses, and arcs. Each visible application object is thus represented by a composite of one or more primitives. The appearance of each primitive is controlled via specification of the attributes (e.g. color, thickness) of the “pen” that will draw its outline and the “brush” that will paint its interior.

Basic raster-graphics packages like SRGP have existed since the dawn of raster displays, and the pioneering packages in this genre shared the same basic characteristics. One of the most well-known
and long-used examples is Microsoft’s GDI, introduced with Windows in the 1980s and still in use by thousands of legacy applications.

Consider a simple GDI application that displays and animates a clock showing the local time, as shown in Fig. 2-1. Below we show highlights from the source code; you can download the complete Visual Studio project from this location:

http://www.sklardevelopment.com/graftext/ChapWPF2D/projects/GDIclock

![Fig. 2-1](image)

Image of clock face generated by project “GDIclock”

In GDI’s simplest usage model, the application uses integer coordinates that map directly (1-to-1) to screen pixels, with the origin (0,0) located in the upper-left corner of the drawing region (e.g., a window’s content area), with x values ascending towards the right and y values ascending towards the bottom.

Here is the essence of the GDIclock application’s rendering method (PaintEntireClockImage() found in Form1.cs):

```csharp
//Specifying the “hardwired” geometry of the clockface:
//This section creates data structures representing geometry
// but does not do any actual drawing yet.
int lengthFaceRadius = 90;
Point ptCenter = new Point(150, 150);
Rectangle circleBounds = new Rectangle(
  ptCenter.X-lengthFaceRadius /*X of upper-left*/,
  ptCenter.Y-lengthFaceRadius /*Y of upper-left*/,
  2*lengthFaceRadius, 2*lengthFaceRadius); /*width,height*/

// Initializing the GDI object whose methods provide
// the actual painting functionality:
// A Graphic object can draw outlined/filled primitives,
// given the geometry specification and
// a set of appearance attributes (pen/brush).
Graphics grafobj = this.CreateGraphics();

// Painting the circular face via 4 steps:
// 1. Setting up a light-gray brush
Brush brGray = new SolidBrush(Color.FromName("LightGray"));
// 2. Using the brush to paint a filled circle
```
Note that each drawing method (e.g. `Graphic.FillEllipse`) receives both the shape’s geometry specification (in the form of the bounding rectangle) and all necessary appearance attributes (e.g., color, pen width).

The clock hands are drawn by custom functions (not shown here, but present in the downloadable project) that perform geometric computations to create arrays of 2D points that are presented to `Graphic.FillPolygon` along with a navy-colored brush. The computation activity is of course necessary to support arbitrary rotation of the clock hands, so the clock can display any time of day.

One’s first impression might be that GDI provides a reasonably good foundation for such an application. However, there are three key characteristics of GDI (and most other early-generation raster-graphics packages) that can cause problems:

1. **GDI operates in immediate mode**, meaning the platform keeps no record of the primitives that compose the scene it has drawn. If anything in the model changes, the application must re-specify all the primitives in the scene.

2. **GDI’s geometry-specification paradigm bears several closely-related limitations:**
   a. The coordinate system is limited to integers, supporting only the ability to refer to pixels on the output device – thus, there is an inherent lack of precision.
   b. It does not include built-in functions for performing the mathematics required to transform primitives geometrically (e.g. to scale/resize, rotate, or move/translate them – if required, the application must do this explicitly).

3. **GDI primitives are completely independent from one another; there is no mechanism for grouping sets of related primitives as named, composite shapes to allow them to be manipulated (e.g., repositioned or rotated) as single entities.**

We now present a variety of problems that emerge from these characteristics.

**Problem 1: Animation**
Suppose we want the clock display to animate in order to continuously update the time. How would we go about making the hands appear to move?

The simplest strategy (demonstrated by the GDIclock sample application) is to regenerate the entire image with the hands in their “new” positions. This strategy is easy to implement, but it is inefficient for scenes composed of very large numbers of graphics primitives.

How would we go about trying to “incrementally” refresh the image (updating as little of the image as possible)? Can we design a technique for moving only the red second hand, e.g. to advance the clock by exactly one second? We might consider a two-step sequence: first “erasing” its current image and then drawing it in its new location.

However, if we erase the current image of the second hand by simply re-painting it using a background color, we risk damaging some part of the clock’s appearance (e.g., when the second hand happens to partially occlude the hour hand, as shown in Fig. 2-2).

In response, the application must do damage repair, by reconstructing any portion of the image that may have been damaged. Thus, the only accurate way to “erase” an object (like the second hand) is to redraw all primitives representing objects that lie underneath the object being erased, in the order in which they were originally drawn.

The 2nd strategy is clearly far more complex than the “regenerate-in-full” strategy, but the two strategies are equivalent in one way: the application must maintain a database of all objects contributing to the scene, since the immediate-mode package retains no information about the primitives that compose the scene.

**Problem 2: Interaction**

Suppose we want to support user interaction (e.g., to let the user set the time simply by dragging the hands into place). When the user clicks on an object’s image to pick it, the location of that click
must be mapped to the relevant application object (e.g., the minute hand). This process is called **pick correlation**, and here again, the application bears the burden for this activity because the immediate-mode platform retains no record of any drawn primitives. To perform pick correlation, the application must find the top-most primitive that covers the cursor’s location. Moreover, since the typical graphics object (e.g. a clock hand) is composed of multiple primitives, the application must additionally identify the composite to which that primitive belongs. (If the composite hierarchy is of arbitrary depth, the result of the correlation activity is optimally the full “stack” of graphics objects, from the primitive all the way up to the highest-level composite of which it is a component).

**Problem 3: Scalability**

Since GDI applications specify primitives using an integer coordinate system that maps directly to the pixel grid on the display device, the resulting size of the primitives’ images depends upon the **resolution** of the graphics device’s screen.

Suppose our clock application was originally designed for a 72-dpi display screen. If the application were tested on a very-high-resolution screen, the clock’s image would appear extremely small, and could even be unusable. Moreover, if the target display were changed to a low-resolution PDA or smart phone, the clock’s image might well become too big, with only a small portion of it visible.

Therefore, if our GDI clock application were required to support a variety of output devices and produce consistently sized results, we as developers would have to bear the burden of performing coordinate-system mapping. The application would store and manipulate the clock’s geometry using “logical” coordinates (preferably floating-point for maximum accuracy) internally, and would render the clock’s image by mapping from logical to “physical” coordinates, using conversion factors selected based on the characteristics and resolution of the target physical device. This burden is quite significant when one considers GDI’s lack of floating-point support and lack of transformation features.

**Problem 4: Handling complexity**

Our clock is a very simple scene composed of only six primitives: three ellipse primitives making up the clock face plus one polygon primitive for each of the three hands. Let’s consider the difficulty of trying to solve problem #3 for this simple scene. The application must:

1. Represent the geometry internally using an abstract coordinate system.

2. Support scaling the geometry to adapt to the target device resolution.

---

1 Throughout this book, we use the term “resolution” to refer to a display device’s density of “pixels” or “dots”, often measured in PPI (pixels per inch) or DPI (dots per inch). Note that this use of the term differs from that of the digital-camera community, which uses “resolution” to refer to the total number of pixels in an image. Note that our use of the term (to mean density) is the long-standing standard for the computer-graphics community.
3. Support moving the clock so it need not lie at the origin but can be placed anywhere on the output device to ensure visibility.

4. Animate each hand individually via rotation, to show the current time.

What geometric transforms need to be performed on each primitive in the scene?

- Each ellipse in the face must be subjected to a scale transform to size it properly, and then a translate transform to move it to the desired position.

- The minute hand needs to experience a custom rotation based on the current minute, then the same scaling transform, and finally the same translate transform.

- Each of the other two hands has its own distinct transform sequence as well, starting with a custom rotation based on the current time.

In all, there are four different transform sequences involved in rendering and animating just this simple scene. Now, let’s consider the two axes on which the complexity of this scene might grow:

- **Number of primitives:** In order to improve the clock’s appearance, we might want to build clock hands that are more “ornate,” composed of several primitives, not just one. Similarly, we might want to improve the clock face to include hash marks, numbers, or a geometric design.

- **Number of transformation sequences:** Each new “moving part” in a scene adds a new transformation sequence to the application. For example, for humorous effect, we might add a swinging pendulum that hangs below the clock. The construction and animation of the pendulum would add one more (at least) custom transform sequence to the application.

Now consider the “bookkeeping” that the application must perform to draw and maintain this simple scene:

- It must keep a list of the transform sequences. To compute each new frame in the animation, it must visit each transform sequence and recalculate its net effect.

- It must keep a list of the primitives, each entry including a pointer to the relevant transform sequence. To compute each new frame of the animation, it must visit each primitive and transform it based on the computed net effect of its relevant transform sequence.

It should be obvious that the necessary data structures and logic are non-trivial, but also that once constructed they provide a foundation for the generation and animation of a wide variety of 2D scenes. As we shall see in the next section, modern graphics packages have indeed absorbed this functionality through features like hierarchical modeling and transforms, thus simplifying 2D (and for that matter, 3D) display.
2.1.2 Evolution in Modern Graphics Platforms

We have enumerated some of the burdens commonly borne by applications based on early-generation raster-graphics platforms. The characteristics that made such platforms difficult to use – immediate-mode operation and resolution-dependent specification – resulted from the platforms being too closely tied to the architecture of the underlying hardware. (The similarity to the architectural dependencies of assembly languages, and the resulting burdens placed on developers using them, should be noted.)

The remedy is a classic one: shifting duties from the application to the platform, allowing the programmer to specify what is to be done at a higher level of abstraction. Over the past three decades, 2D platforms have followed a path of progressive improvements, as shown in figure 2-2(A):

1. Virtually all modern 2D graphics packages allow primitives to be specified in a device-independent floating-point coordinate system. The package takes care of converting from “model-space” coordinates to “device-space” coordinates.

2. In addition, many graphics packages provide hierarchical modeling capabilities, allowing the application to construct composite graphic objects by combining primitives and other composites – building a scene as a hierarchy of graphic objects – and to manipulate composites as coherent objects.

3. Advancing even further, some graphics packages keep a record of the scene’s model, via what is often called retained mode, thus taking care of the respecification and damage-repair necessary for achieving dynamics.

4. Still others take it one step further, introducing “smart objects” by allowing graphic objects to be given innate behaviors and interaction responses (e.g. an instance of a Button class might automatically glow or highlight itself when the cursor enters its domain, and remove the highlighting when the cursor leaves its domain).
The remainder of this chapter introduces one of the more highly-evolved 2D graphics packages, Microsoft’s WPF, showing how modern graphics features simplify the construction of applications like our animated clock. We first demonstrate the utility of floating-point coordinates, transformations, and hierarchical modeling in the generation of a static clock image. We then show how retained-mode features simplify the process of animating the scene.

---

2 In computer-graphics literature, the term “vector graphics” is often used to refer to platforms with the types of functionality mentioned here: floating-point abstract coordinate system, transformations, retained-mode. This is a radical departure from the original use of the term, which refers to images generated on pen plotters and calligraphic output devices like the IBM 2250.
2.2 2D GRAPHICS MODELING AND ANIMATION USING WPF

2.2.1 The Role of Declarative Specification

Modern graphics platforms offer programmers with a variety of techniques for specifying the application’s appearance (both its user interface and its visual content):

- Procedural code written to an Application Programming Interface (API), e.g. Microsoft’s GDI+.
- Object-oriented code (construction of instances of classes, invocation of their methods, and access to their properties) written atop an OO platform, e.g. Java Swing, Microsoft .NET.
- Declarative specifications written using a documented standardized language, e.g. X3D, SVG.

Although a platform typically provides for only one specification method, the Windows Presentation Foundation provides developers with a choice of layers for performing model specification.

![Software Engineers](image1)

**Fig. 2-3** WPF application/developer interface layers

The three layers that form this “application/developer interface” are:

- **Lowest layer: object-oriented API**
  The core layer is a set of classes providing all WPF functionality. Programmers can use any of Microsoft’s .NET languages (e.g., C# or Visual Basic) to specify application appearance and behavior at this level. Any WPF application can be created via this layer alone, but the other two layers provide improvements in developer efficiency and comfort, and the ability to include designers and implementers in less technical roles.

- **Middle layer: XAML**
  The second layer provides an alternative way to specify a large subset of the functionality of the API, via a declarative language called XAML, which uses syntax that should be readily understandable by anyone familiar with HTML or XML. The functionality available at this
The declarative nature of this alternative form of specification improves upon procedural specification in several areas: it is much more succinct, provides support for rapid prototyping via immediate (interpretive) execution that avoids the cumbersome edit-recompile-run cycle, and is more conducive to use by non-programmers (in the same way that HTML is more “approachable” than C#). However, the specification of reactive behaviors (e.g., responding to the user picking a shape or UI control) will likely require some procedural code in the callback.

The subset of functionality available via XAML includes dynamics (via animation and some event handling); thus, designers can construct “executable design specifications” in pure XAML. These executable programs can be used, for example, at team meetings and focus-group sessions as rapid prototypes to demonstrate and modify in real time the designers’ goals for the application’s appearance, fundamental interaction behaviors, and navigation/flow.

- **Highest layer: tools**
  As with any language, even a declarative one, there is a learning curve associated with its use. The third and highest layer of the WPF application/developer interface takes advantage of Microsoft’s choice of XML (which is easily generated and parsed) as the basis for XAML. This layer includes the tools that developers (in both roles) can use to generate XAML code and/or procedural source code – tools for drawing graphics (Microsoft Expression Design, Adobe Illustrator), creating/converting 3D models (ZAM3D, 3DPaintBrush), laying out user interfaces (Expression Blend), etc.

### 2.2.2 The Structure of a XAML Application

Let’s examine a simple XAML application that displays the working analog clock from section 2.1, having the enhanced appearance shown in the figure below:

![Image of clock](image.png)

**Fig. 2-4**
Image from WPF version of the clock application

If you are familiar with HTML syntax, you will find XAML to be quite familiar. An HTML file specifies a multimedia document/webpage by creating a hierarchy of *elements* (with the root being `<HTML>`, its children being `<HEAD>` and `<BODY>`, all the way down to paragraph and low-level text-wrapper elements for formatting, such as `<B>` and `<I>`, specifying boldface and italics, respectively). There are also elements that support the presentation of images, audio/video clips, etc.
XAML similarly specifies a graphics model by creating a hierarchy of elements using the same syntax (see the “XAML Tips” sidebar below) found in HTML. However, the set of element types is distinct to XAML. For example, the root element, instead of always being <HTML>, can be selected from a set of “top-level” XAML element types. For this application, we will use the Canvas, which is simply a “blank slate” upon which you can construct a scene composed of 2D shapes.

Before we go any further, we must give a caveat: the XAML code we show in this chapter is a simplified flavor of XAML that omits some “syntactic vinegar” that is required by the actual XAML parser but that may be distracting to those who are seeing XAML for the first time. The “full” XAML for this sample is shown in the online laboratory that accompanies this chapter’s description of the WPF clock application. The lab, which allows you to peruse, edit, and execute the XAML for various revisions of the clock application, is available at this location:

http://www.sklardevelopment.com/graftext/ChapWPF2D/WPF2Ddemo.xbap

Let’s start with a Simplified XAML application that creates an empty canvas, not yet populated with shapes:

```xml
<Canvas Name='ClockCanvas'>
</Canvas>
```

XAML Tip:

In XML, which is the syntax behind XAML, an element is represented by:

- a start tag, which declares the element’s name and then optionally sets the element’s attributes (to override their default values) in any order...
  ```xml
  <ElementName attribute1='value1' attribute2='value2' ... >
  ...
  followed by the XML specification of the element’s “internal” content (if any), which can include other “subordinate” XML elements and/or textual content to an arbitrary nesting depth...
  ...
  then terminated by an end tag: </ElementName>
  ```

If the element has no internal content, the end tag can be absorbed into the start tag by placing a “/” before the “>”:

```xml
<ElementName ... attributes ... />
```

Watch for this abbreviation technique in the next section, for the Ellipse element.

2.2.3 Creating and Placing Visible Elements

We now will start placing visible elements into the canvas to start forming the clock face. Note that when constructing 2D scenes, the order in which you create the components is important – an element Y, constructed after element X, can occlude X if they overlap, but X can never occlude Y. (This is why objects created using 2D vector graphics are sometimes said to be “two-and-a-half dimensional”.)

3 All “XBAP” demos in this textbook require Windows 7/Vista and Internet Explorer 7+. If your computer does not meet the requirements, your browser will merely show some XML code.
First, let’s create our clock’s background, which is composed of a circle. The syntax for the `Ellipse` element is:

```xml
<Ellipse
    Width='...' Height='...'
    Left='...' Top='...'
/>
```

where “Left” and “Top” specify the x and y coordinates for the upper-left corner of the primitive. (E.g. Left=0 and Top=0 represent the origin of the coordinate system).

Of course, in order to determine what numbers to use for its various geometry attributes, we need to determine the characteristics of the coordinate system. It might help for us to review that a planar coordinate system has these characteristics:

- The semantics for the unit of measurement, e.g. a very precise “conversion” semantic like “one unit = one micron” for a VLSI-circuit CAD application, or a more abstract semantic like “one unit = approx radius of the Earth” for a planetology application
- The location of the origin point (0,0)
- The orientation of the axes (e.g. does positive y point “up” or “down”?)

The WPF native coordinate system, shown in Fig. 2-5, has its origin (0,0) at the upper-left corner of the drawing canvas, with the positive x axis extending to the right and the positive y axis extending down.

![Fig. 2-5](image)

This coordinate system uses floating-point numbers, and the unit of measurement is 1/96th of an inch\(^4\). A WPF application can choose to use an alternative coordinate system of its own devising,

---

\(^4\) WPF’s designers chose this for historical reasons: 1/96 inch has long been the Windows standard "device-independent unit" (DIU). In the mid-1980s, the typical resolution of VGA monitors was 72 dpi, so 1/72 inch was a natural choice (and indeed that was the DIU for early generations of the Macintosh). However, the GDI development team noted that the typical person positions herself 1/3 further from a display screen than from a printed page. Thus, to make text rendered at a given point size look approximately equivalent on screen and on paper, the DIU for GDI was set as though the typical monitor resolution was 33% higher, or 96 dpi. As a result, a Windows application specifying a line of one inch in length would yield a 96-pixel-long line, whereas a Mac application...
provided that it tells the platform how to map from that coordinate system to the platform’s native coordinate system.

For example, we might find it more natural to specify our clock face using a custom coordinate system (shown in Fig. 2-6) with the origin at the center of the drawing canvas, and one unit being equivalent to one inch.

Let’s specify the clockface’s background circle in this custom coordinate system, filling it with a solid color. Below is our Simplified XAML revised to include its first graphic primitive:

```xml
<Canvas ... >
  <Ellipse
      Left='-1.0' Top='-1.0'
      Width='2.0' Height='2.0'
      Fill='lightgray'/>
</Canvas>
```

If you are using the online laboratory, execute this application by selecting “V.01” in the lab’s Table of Contents. The result will be quite unexciting: a completely empty canvas image.

### 2.2.4 Using Transforms to Map Between Coordinate Systems

The reason for this disappointing result is that our circle’s diameter is only 2/96\textsuperscript{th} inch, and such a tiny primitive is invisible on any normal-resolution screen. We had neglected to instruct WPF to specifying the same would yield a line of only 72 pixels long. The Windows choice thus acted as an automatic “scale factor” on all rendered text and graphics.

It is also important to note that since the Windows graphics software platform does not have control over the accuracy of 3\textsuperscript{rd}-party graphics hardware and driver software, its dependence on a standard physical unit of measurement is approximate only. E.g., a line of 96 units drawn on a WPF canvas using its native coordinate system will appear to be one inch long on an “ideal device”… but when displayed on a real device, results may vary.

\[\text{Going forward, in each XAML fragment we will embolden the new tags to make it clear what is being added in that example and thus deserving of your attention. Also, we will use ellipsis notation (…) to replace code segments presented in previous fragments, and italics to present inline commentary.}\]
convert, or transform, our preferred coordinate system into WPF’s native coordinate system. To perform this task, we add transformation elements to the canvas’ contents.

Obviously, we must convert coordinates from our preferred system (1 unit = 1 inch) to WPF’s system (1 unit = 1/96th inch). We do this by multiplying our coordinates by 96 on both the x and y axes, via use of a scale transform. Here is the revised Simplified XAML:

```xml
<Canvas … >
  <RenderTransform>
    <ScaleTransform ScaleX='96' ScaleY='96' CenterX='0' CenterY='0'/>
  </RenderTransform>
  <Ellipse … />  
</Canvas>
```

Note that when you specify a 2D scale operation, you must specify the center point, i.e. the point on the plane that will not move, while all other points are moved away from (or towards) it as a result of the scale. Also note that since the XAML is executed sequentially, all primitives following the transform (here only the ellipse) are affected by it.

If you’re using the online laboratory, visit revision V.02 to see the rendered result (Fig. 2-7) of the execution of the XAML as it now stands.

![Rendered result of revision V.02 of the online clock-application lab.](image)

Why are we only seeing one quadrant of our circle? This is due to another difference between the two coordinate systems: the location of the origin. In WPF’s native system, the origin lies at the upper-left corner of the canvas, and only image portions lying in quadrant I (positive values for both x and y) are visible. But in our preferred system, the origin (0,0) needs to be in the “middle” of our canvas, because our clock’s image is centered on the origin and thus occupies all four quadrants.

Thus, we want to add another transform to our canvas to supplement our already existing scale transform, in order to move our coordinate system to match our preferred origin placement. We will use a transform of type `translate`, whose XAML representation is:

```xml
<TranslateTransform X='__' Y='__'/>
```

By how many units do we want to move our coordinate system? Well, that depends on what unit of measurement is in effect when the `TranslateTransform` element is being interpreted.

- If our preferred system’s units are in effect, we want to move one unit (inch) to the right and one unit down.
- If WPF’s native units are in effect, we want to move 96 units to the right and 96 units down.
How do we know which unit of measurement is in effect? It helps to keep in mind that when a canvas contains multiple transforms, they are applied in the order in which they are specified.

Thus, if we plan to specify the Translate after the Scale, the unit of measurement that will be in effect will be WPF’s native one. Thus, we need to translate 96 units to the right and 96 units down.

Here is the revised Simplified XAML showing both transforms, and Figure 2-8 shows the rendered result of this revisions of our application.

```xml
<Canvas ...
<RenderTransform>
    <ScaleTransform ScaleX='96' ScaleY='96' Center ...
    <TranslateTransform X='96' Y='96'/>
</RenderTransform>
<Ellipse ...
</Canvas>
```

![Figure 2-8](image)

**Fig. 2-8**
Rendered result of revision V.03 of the online clock-application lab.

**EXERCISE:**
The decision to perform the scale before the translate was purely arbitrary; a reverse order will work as well, with different values for the numeric properties. Using the online lab, visit V.03 and edit the XAML code to reverse the order of the two transforms. First change the order without adjusting the numeric properties, analyze the resulting incorrect image, and then do the correction.

**EXERCISE:**
(This exercise can be done either before or after the previous exercise.)
Note that the circle is “hugging” the top and left sides of the canvas, which may not be optimal aesthetically. Using the online lab, visit V.03 and edit the TranslateTransform’s property values to move the circle to a more “comfortable” (e.g., centered) location.

Figure 2-9 provides a schematic representation of the action we have just performed:
Note that in the XAML this transformation sequence is a direct child of the **Canvas**. However, it is not considered part of the canvas’ content, but rather is considered to be a **property** of the canvas, asserting its influence on all objects placed on the canvas. Therefore, as we continue to specify new components (e.g., the clock’s hands) into the scene, as children of the **Canvas** element, we must consistently use our custom coordinate system (1 unit = 1 inch).

### 2.2.5 Using Transformations to Model with Reusable Templates

The previous section’s introduction to transformation has focused on just one of its uses: mapping from one coordinate system to another. But transformations are also used to perform **modeling** by allowing a scene to be constructed by positioning and adjusting copies of pre-defined, re-usable components we can think of as stencils, or (in WPF terminology) **templates**.

Consider how we might approach defining the hour and minute clock hands. Both of these hands share a “lozenge”-like shape. Indeed, the minute hand is longer and skinnier, and the hour hand shorter and wider, but those can be considered just two minor adjustments to a single template. So let’s consider how we might construct and place those two hands by defining and using the template shown in Fig. 2-10.
The WPF element type **Polygon** is used to create an outlined or filled polygon via a sequential (either clockwise or counter-clockwise) specification of the vertices. Here is the XAML specification of our canonical clock hand, to be filled with a dark-blue color (note the use of spaces to separate coordinate pairs):

```xml
<Polygon
    Points='0,0 -0.2,0.8 0,1 0.2,0.8'
    Fill='Navy'/>
```

As mentioned previously, we want this **Polygon** element to be a reusable template, defined once and then *instantiated* (copied into a scene) any number of times. WPF templates are shape collections specified in the *resource section* of a XAML application’s root element (e.g. the **Canvas** element). Each template must be given a unique name (using the **Key** attribute) so it can be referenced for the purpose of instantiation.\(^6\) This is just the classic “declare before use” principle common to many programming languages.

---

\(^6\) Instantiation of a stencil is identical conceptually to creation of a new instance of a class in an object-oriented programming language. Interestingly, this concept’s appearance in computer graphics is older than OOP itself, since Ivan Sutherland used this paradigm of “masters and instances” in his pioneering Sketchpad system that launched interactive computer graphics in the early 1960s.
This diagram represents the template that we have created:

![ClockHandTemplate](image)

**Fig. 2-11**
Schematic representation of our single-polygon clockhand template

If we were to execute our application with its new template specification, we would not detect any change; still, only the gray circular clock face would be visible. We need to instantiate that template to create an object to be shown on the canvas; in Simplified XAML, this is done via the TemplateInstance element.

Let’s create one instance of our new template: the one that will become the minute hand of our clock. First, we’ll instantiate the template “as-is” – using its default geometry – to produce revision V.04:

<![-- 1. Background of the clock: -->
  <Ellipse ... />
  <!-- 2. The minute hand: -->
  <TemplateInstance Name='MinuteHand' Template='ClockHandTemplate'/>

As is evident in Fig. 2-12, the hand’s canonical size is not appropriate when placed “as-is” on the clock face. Let’s reduce its length and thickness, and also rotate it to the 12 o’clock position. We do so by adding an instance transform as a child of the TemplateInstance element:

<![-- 2. The minute hand: -->
  <TemplateInstance Name='MinuteHand' Template='ClockHandTemplate'>
    <RenderTransform>
      <ScaleTransform ScaleX='0.25' ScaleY='0.9' CenterX='0' CenterY='0'/>
      <RotateTransform Angle='180' CenterX='0' CenterY='0'/>
    </RenderTransform>
  </TemplateInstance>
Figure 2-13 summarizes the effect of the operations we have performed thus far in section 2.2.5, and the rendered result as produced by revision V.05 of the online lab is shown in Figure 2-14.

![Application Coord System](image)

**Fig. 2-13**  
Instance Transformation:  
A transform sequence is attached to the instantiation of the template, allowing the template instance to be placed and resized as desired.

![Rendered result](image)

**Fig. 2-14**  
Rendered result of revision V.05 of the online clock-application lab, showing a now-transformed instance of the clockhand template.

Note that to specify a rotation, you must provide not only the amount of the rotation (in degrees, which is the standard unit of measurement for all angular properties in WPF), but also the “center of rotation”, i.e., the point around which the rotation is to occur. One of the nice features of our custom coordinate system is that (0,0) represents the center of the clock, so that origin point conveniently serves as the center of rotation for the clock hands (and also as the center point for the scaling operation).

Adding the hour hand is an equivalent process: creating a new instance of the canonical hand, with its own transformation sequence. Its scale factors will differ from those of the minute hands, to give it a somewhat shorter and “fatter” shape. Also, we rotate our hour hand a different amount, to have the clockface show 9 o’clock. Fig. 2-15 illustrates the two distinct instantiations schematically, and Fig 2-16 shows the rendered result.

```xml
<RenderTransform>
  <ScaleTransform ScaleX='0.35' ScaleY='0.6' CenterX='0' CenterY='0' />
  <RotateTransform Angle='90' CenterX='0' CenterY='0' />
</RenderTransform>
```
Adding the hand that measures seconds is something we’d like you to do (by starting with V.06 and using the online lab’s editor). Our own solution is available in the online lab as V.07. You may or may not want to use the same template for the second hand; feel free to define a different shape and/or color for it!

**EXERCISE:**
Start with lab section V.06. Add the hand that measures seconds. You can create your own polygon template for the second hand to give it a unique shape, or you can transform the same template we’ve been using for the other two hands. Our solution to this exercise is available in lab section V.07.

**EXERCISE:**
The “preferred” app coordinate system we’ve used for this lab used the same y-axis-points-downward orientation that is the WPF default. For this exercise, build this clock using an app coordinate system in which positive-y points upward. Start with your solution to the previous
exercise OR with our solution (V.07), and then make the necessary changes to transition to this more natural axis orientation.

But before you dive in to your first XAML authoring exercise, let’s stop for a moment and review the modeling activities we have performed in sections 2.2.4 and 2.2.5, shown schematically in Fig. 2-17.

- We have created a scene by instantiating reusable templates, using instance transforms to place, reshape, and orient the instances.
- We have defined a custom coordinate system for the entire scene, to simplify specification of geometry, and we have used transformations to perform coordinate-system mapping to convert the scene’s geometry into the native coordinate system of the WPF canvas.

2.3 **DYNAMICS IN 2D VECTOR GRAPHICS USING WPF**

A retained-mode architecture simplifies the implementation of dynamics; the application focuses on maintaining the model, and the platform carries the burden of keeping the screen image in sync with the model.
In this section we examine two types of dynamics available to WPF applications:

- Automated, non-interactive animations, in which 2D shapes are manipulated by “storyboards” that can be fully specified in XAML.
- Traditional user-interface dynamics, in which procedural code (callbacks) is activated by user manipulation of GUI controls, like buttons, list boxes, text-entry areas, etc.

### 2.3.1 Dynamics via Declarative Animation

WPF provides the ability to specify animations using XAML (i.e. without the need for procedural code) via *animation elements* that can force object properties to move from starting values to ending values in a controlled manner via time-based interpolation. We will use this feature (which owes a great deal to predecessors like SVG/SMIL) to rotate the clock hands at appropriate speeds.

An application sets up a desired dynamic by creating an animation element, connecting it to the property that it is meant to manipulate, and specifying the animation’s characteristics (e.g., the starting value, the ending value, the rate of speed of change, and whether it should start over when reaching the ending value). Finally, the application specifies what event should trigger the activation of the animation.

Once set up, the animation element does its work automatically, without the need for any intervention by the application.

Just about any property of any XAML element can be the target of an animation. Examples include:

- The origin point of a shape (e.g., the upper-left corner of a rectangle) could be manipulated by an animation element to make the shape appear to vibrate.
- The fill-color, edge-color, or edge-thickness properties of a shape primitive could be manipulated by an animation element to perform feedback animations, such as glowing or pulsing.
- The angle property of a rotation transform could be manipulated by an animation element to make the affected objects rotate.

Obviously, that last example is of interest to us as clock builders. We can use three animation elements, one for each of the clock hands, to provide for the clock’s movement.

Let’s look at the current status of our minute hand’s transform group, as designed in Section 2.2. Currently, it is simply an instance transform:

```xml
<RenderTransform>
  <!-- The instance transform (scale + rotate): -->
  <ScaleTransform ScaleY='0.9' ScaleX='0.25'/>
  <RotateTransform Angle='180'/>
</RenderTransform>
```

The instance transform already contains a `RotateTransform`, which is being used to rotate the hand from its pointing-down position (which is naturally how it appears in WPF’s Y-axis-downward orientation) into a 12 o’clock position. If we were to add procedural code to this application, to
initialize the clock to the actual local time, we would change this element: it would need to have a unique name, i.e., a value for the name attribute, to allow the procedural code to access it, and it would no longer need a hardwired initial angle value. But for now, let’s keep this element as-is to preserve the current default 12 o’clock position.

So, to achieve animation, let’s add a second `RotateTransform` to be manipulated by WPF’s animation logic:

```xml
<RenderTransform>
  <!-- The instance transform (hardwired scale + computed rotate): -->
  <ScaleTransform ScaleY='0.9' ScaleX='0.25'/>
  <RotateTransform Angle='180'/>
  <!-- The animation transform (rotate controlled by animation element): -->
  <RotateTransform Name='minuteAnimator' />
</RenderTransform>
```

Now, let’s look at the declaration of the animation element that will rotate the minute hand through a full circle each hour. WPF provides one animation element type for each data type that one might want to automate. For animation of a rotation magnitude, which is a double-precision floating-point number, you use the element type `DoubleAnimation`:

```xml
<DoubleAnimation
  TargetName='minuteRotator'
  TargetProperty='Angle'
  From='0.0' To='360.0'
  Duration='01:00:00'
  RepeatBehavior='Forever'
/>
```

The animation is connected to the minute hand through the setting of the `TargetName` and `TargetProperty` attributes. The `From` and `To` attributes determine the range and direction of the rotation. The specification of `Duration` controls how long it will take (in terms of CPU “ticks”) for the animation element to move the property’s value through the specified range of values. Finally, the `RepeatBehavior='Forever'` setting ensures the clock hand will continue moving as long as the application is running; as soon as the value reaches the “To” destination, it “wraps around” to the “From” value and continues the animation.

Sidenote:

You may well wonder about the accuracy of the actual animation, considering how precise the specifications are. Will the animation operate if the CPU is under heavy load or is insufficiently powered?

Although the smoothness of the animation may suffer (become “jumpy”) when the CPU is stressed, the image will keep up with where it needs to be at any given time. The animation engine works in an absolute way – newly calculating where the property values should be at the present time – instead of in a relative way based on accumulation of deltas. Thus, even if the WPF application has been denied adequate CPU time for a long period of time – seconds or even minutes – the image will

---

7 A WPF tick is equivalent to 100 nanoseconds.

8 Other available behavior types include reverse motion (i.e., “bouncing back”) and simply stopping (for “one-shot” motions).
jump to the correct state when the application next receives sufficient CPU cycles for image refreshing.

The final step is to “install” the animation in its proper place in the XAML code. We want the animation to start as soon as the clock face is made visible. Thus, we create an EventTrigger and use it to set the Triggers property of the canvas. A trigger must specify what type of event launches it (in this case: as soon as the XAML is fully loaded) and what action it performs (in this case: a “storyboard” composed of three animation elements):

```xml
<Canvas ...

    ... The specification of the entire clock model goes here. ...

<Triggers>
    <EventTrigger RoutedEvent='FrameworkElement.Loaded'>
    <Actions>
        <BeginStoryboard>
        <Storyboard>
            <DoubleAnimation
                TargetName='minuteRotator'
                TargetProperty='Angle'
                From='0.0' To='360.0'
                Duration='01:00:00' RepeatBehavior='Forever' />
            ...
        </Storyboard>
        </BeginStoryboard>
        </Actions>
    </EventTrigger>
</Triggers>
</Canvas>
```

Revision V.08 of the online lab application shows this animated clock, but its XAML has been modified to make the minute hand move unnaturally fast to make it easier to notice and test the animation’s movement. We recommend the following exercises to those who want to test their understanding of this section:

Study the XAML code of revision V.08, and then perform these exercises:

1. Make the minute hand’s animation speed accurate (one rotation per hour).
2. Set up the animation for the second hand by adding another DoubleAnimation element to the storyboard.
3. Finally, add the animation for the hour hand by creating a third DoubleAnimation element.
4. If you’d like to demo the clock to a friend, “manually” change the instance transformations in the XAML code in order to initialize the hands to better approximate the actual time at your location.
2.3.2 Graphical User Interfaces in WPF

This introduction to 2D WPF has addressed only its support for modeling and animating a scene containing behavior-less 2D shapes. Sitting atop WPF’s shape classes (e.g. Polygon and Ellipse) and modeling support (templates and transformations) is a UI-framework layer that adds interactivity and feedback behaviors, and above that layer sits a large palette of ready-to-use UI control classes (like buttons, sliders, dropdown lists, etc.). Moreover, WPF supports automated geometric layout functionality (via its set of Panel classes) that simplifies the task of creating the appearance of a complex user interface.

Chapter [2DTESTHARNESS] provides some examples of XAML’s UI framework; for a more complete survey of the gamut of XAML UI controls, you are invited to browse the online materials located at:

http://www.sklardevelopment.com/graftext/ChapWPFD2D/section2.3.2

2.3.3 Dynamics via Procedural Code

Obviously, there is a limit to the richness of an application built using XAML alone. Procedural code is a necessary part of any sophisticated interactive application, for the performance of processing, logic, database access, and more sophisticated interactivity. WPF developers use XAML for what it’s best suited (scene initialization, resource repository, simple animations, etc.), and use procedural code to complete the application’s functionality.

XAML and procedural code are integrated in a variety of ways in a WPF application:

- A data-binding feature allows a XAML object’s properties to be linked to data members/variables in the procedural code, facilitating auto-synchronization (one-way or bi-directional). This same feature is what allows an Animation element to manipulate properties of other XAML elements, as we saw in section 2.3.1.

- Each element type in the XAML (e.g. Ellipse) corresponds to a class on the procedural side, and each attribute (e.g. Width) to a data member thereof. Thus, an instantiated XAML object is available to the procedural code as a full-fledged object, just as though the object had been instantiated “the usual way” by the new operator in procedural code.

- The XAML UI control classes support event-handling, allowing a XAML element to directly reference a callback method on an object in the procedural code (e.g. an instance of the XAML Button element type can specify the procedural code to be called when the user interacts with it).

If you are interested in learning more about how applications can be constructed via the integration of procedural code and XAML, browse the online materials located at:

http://www.sklardevelopment.com/graftext/ChapWPFD2D/section2.3.3