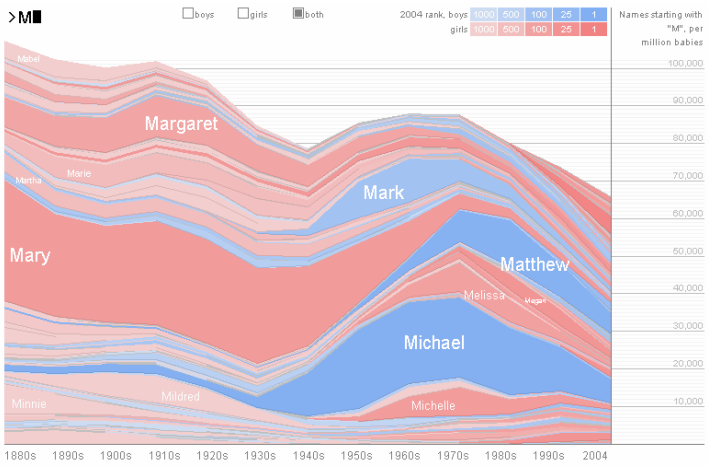
Lecture 13: Animation

UI Hall of Fame or Shame?



Suggested by Casey Dugan

This web site is the Baby Name Wizard's Name Voyager (www.babynamewizard.com).

It's a great example of direct manipulation. The baby names have a continuous visual representation, which I can control by physical actions (clicking on a colored line). The text entry field is an incremental search, so it has rapid, incremental, and reversible effects.

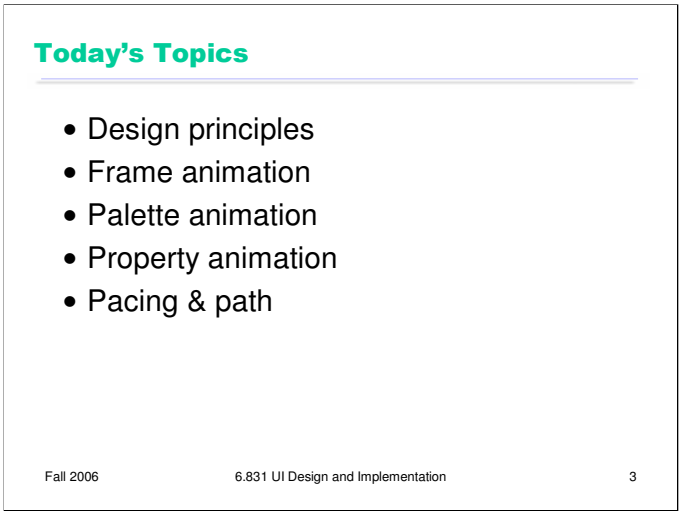
Alas, some actions do *not* have easily reversible effects. Clicking on a name, like Michael, can't be reversed by another click or single keypress; instead, I have to backspace to erase the whole word "Michael" in order to get back to where I was before. That makes it much harder to explore, and makes me (as a user) unwilling to click on something, because the cost of reversing the click isn't worth the benefit I get from zooming in on one name.

Another unfortunate limitation is that I can't zoom or filter the time line. Knowing how popular Mary was in the 1880's is historically interesting, but if I'm trying to choose a name for my baby, the last 10 or even 5 years may be much more important to me, so I'd want to use as much more length of the y axis position for that range.

Another issue is that you can't easily compare names that don't share a common prefix – but comparing names is very important to new parents. This may be a failure of task analysis.

What visual variables are used here?

- **hue** for gender (a nominal variable)
- **saturation** for popularity rank (an ordinal variable)
- **size** (vertically) for frequency (a quantitative variable)
- **position** (horizontally) for time (a quantitative variable)

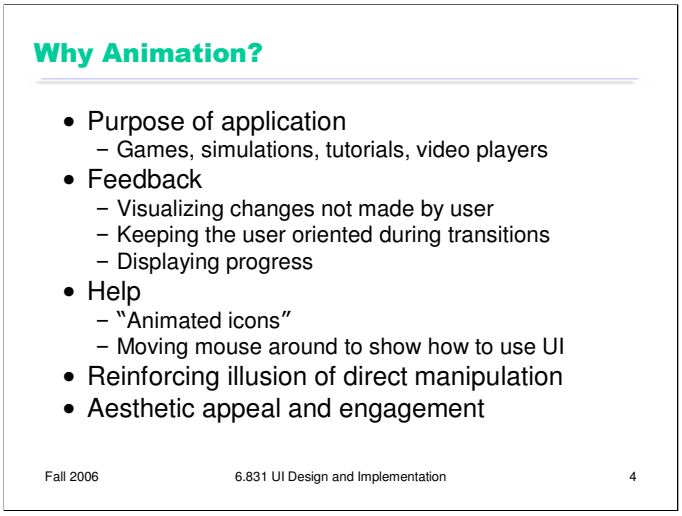


Today we're going to talk about using animation in graphical user interfaces.

Some might say, based on bad experiences with the Web, that animation has *no* place in a usable interface. Indeed, the <blink> tag originally introduced by the Netscape browser was an abomination. And many advertisements on the web site use animation to grab your attention, which distracts you from what you're really trying to do and makes you annoyed. So animation has gotten a bad rap in UI.

Others complain that animation is just eye candy – it makes interfaces prettier, but that's all.

But neither of those viewpoints is completely fair. Used judiciously, animation can make an important contribution to the usability of an interface. And it's not as hard to implement as you might think. We'll talk about both design issues and implementation today.



Why would we want to use animation in a GUI?

First, it might be an essential part of the application itself. Games and educational simulations generally *have* to use animation to be realistic and engaging, because they're simulating a virtual world in which time passes and things move. And a video player would be pointless if the moving pictures didn't actually move.

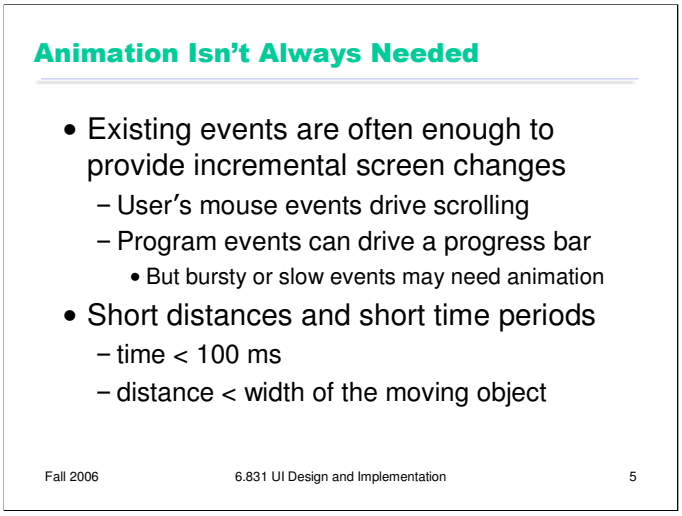
Even if the application's main purpose isn't animation, animation can enhance **feedback**, by drawing attention to and explaining changes in the display that would otherwise be hard to follow and understand. One example is a change that was not made by the user – due to actions by other users in a multi-user application, or actions by the software itself. A move made on a checkerboard by another player is an example of a change that might be animated.

Another change is a major **transition** in the viewpoint of the display, because a sudden drastic shift can disorient the user. What happened? Which direction did I go? How would I get back to where I was before? Scrolling around a large space, or zooming in or out, are examples of these transitions. Problems tend to occur not when the transition is under direct manipulation control (e.g. dragging a scrollbar thumb), but rather when it happens abruptly due to a button press (Page Up) or other action (Zoom pulldown menu, hyperlink press, another user or computer action, etc.) Animating the transition *as if* the user had done it with fast direct manipulation helps the user stay oriented.

Animation can also be used for **progress feedback**, to show that something is still happening. Here, the animation basically reassures the user that the program hasn't crashed. The "throbber" on a web browser (that spinning earth or stomping Mozilla or whatever) is an example of this kind of animation; it's probably the most trivial kind to do.

Animation is also used for **help**. There was some research back in the early 90s on animated toolbar icons, which showed a little movie of how the tool worked when you hovered over with your mouse (Baecker, Small & Mander, "Bringing Icons to Life", 1991). Documentation and tutorials can also *demonstrate* how to use the interface by actually moving the mouse pointer around. (Java has support for this in the java.awt.Robot class.)

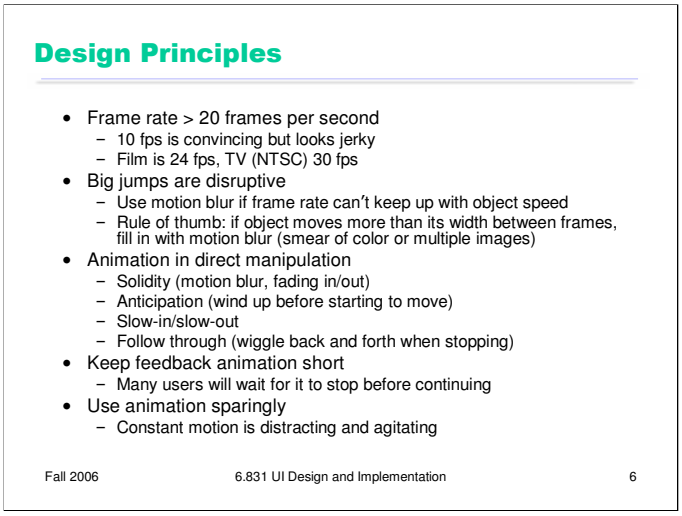
Animation also has some subjective and perceptual effects (we're getting closer to the eye candy argument now, but subjective satisfaction matters too). Animation makes interfaces more lively and engaging, more appealing. It also reinforces the illusion of the physical simulation provided by direct manipulation. If, rather than teleporting from one place to another, objects in the GUI world have to move through the intervening space, then it looks more physical.



Fortunately, it turns out that in many cases, you don't need to do anything special to obtain the benefits of animation.

Many event-driven parts of a GUI are *already* incremental. If the user is dragging the scrollbar thumb, then they're basically animating the interface themselves – you don't need to do anything special. (Although we'll discuss something called *motion blur* on the next page that may be worth adding to enhance the visual effect.) Similarly, backend events (like bytes read from a file or files copied) may be able to make progress feedback appear animated, simply by coming often enough so that the progress bar fills in smoothly. But if the backend events are bursty or have long gaps between them, you may need to supplement with animation. That's why web browsers have an animated throbber – because sometimes network connections just sit there generating no backend events to drive the progress.

If feedback is very brief, or a transition very short, then animation is likewise unnecessary, because it will be lost on the user anyway. Some good rules of thumb are shown above.



Here are some basic design principles for animation in a GUI.

First, the **frame rate** should be at least 20 frames per second – i.e., the animation should make an incremental change at least 20 times per second.

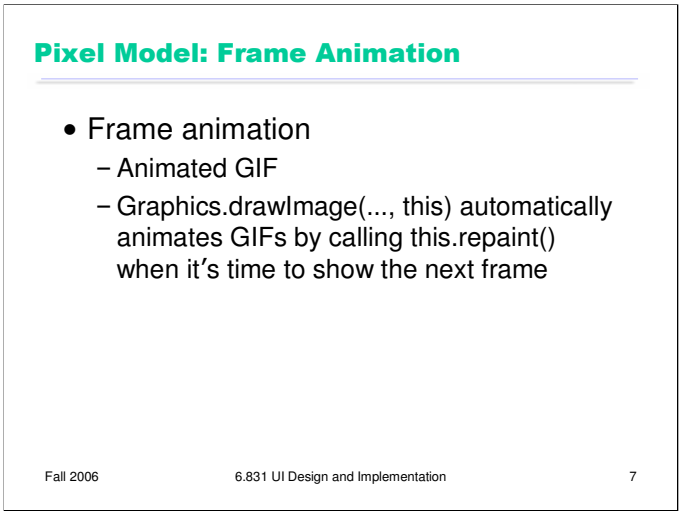
Big jumps are disruptive, so pay attention when you're using low frame rates with high object speeds. In the real world, an object doesn't disappear from one place and reappear in another – it passes through the intervening points, and even if it moves too fast for us to focus on it, it leaves an impression of its path our visual system – a *smear*. That smear is called **motion blur**. If an object is moving so fast that it moves more than its own width between frames, leaving a visible gap between subsequent images of the object, then you should consider filling in the gap with simulated motion blur. Two common ways to do it: (1) a smear of the object's color, and (2) simply drawing multiple overlapping images of the object. Another solution is to crank up the frame rate, if that's possible

There are several useful ways to use animation to enhance the illusion of direct manipulation (Chang & Ungar, "Animation: From Cartoons to the User Interface", UIST '93), which were originally drawn from the experience of Disney cartoonists. The principle of **solidity** says that the animated behavior of an object should give clues about its squishiness – so a ball, when it strikes the ground, should flatten out into an ellipse before rebounding. Most objects in GUIs are rigid, so the solidity principle is mostly about preventing high-speed GUI objects from appearing to teleport across the screen (using motion blur), and having them fade into and out of view rather than appearing and disappearing abruptly.

Anticipation means that an object winds up a bit (moving backwards to get more leverage) before starting a motion. The wind-up draws the user's attention, and resembles what animate creatures in the real world do when they move. **Slow-in, slow-out** describes how a realistic animation should be paced – rather than keeping a constant speed throughout, the object should accelerate up to a cruising speed, and then decelerate to a stop. Finally, **follow-through** says that objects should wiggle back and forth a bit when they finish a motion, to expend the remaining kinetic energy of the motion.

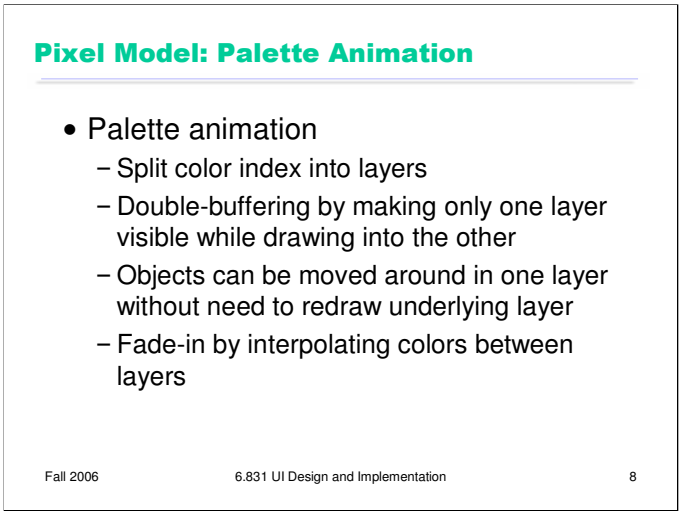
Feedback animations should be kept short. Many users will wait for it to stop before continuing, so there's a tradeoff between the duration of the animation and the efficiency of the interface. Don't block user control – allow the user to start on their action while the animation is still going on.

Finally, use animation judiciously. As we know from ads on web sites, constant motion is distracting and agitating.



Now let's focus on how to implement animation, starting with animations in the pixel model.

Frame animation is probably the easiest kind of animation. It consists of a sequence of images, each a little different from the previous, that are displayed at regular intervals. Most of the moving pictures you know -- movies, television, digital video – work this way. Animated GIFs are an easy way to put frame animation into a GUI. (One drawback is that GIFs have only 8-bit color resolution; if you need richer color, you may have to do it yourself by displaying a sequence of PNGs, or switch to a video player that supports AVI or MPEG – but video players are much harder to integrate.) Java has built-in support for animated GIFs. You just use Graphics.drawImage() to draw the animated GIF during your paint method, passing your component as the ImageObserver. Java will automatically set a timer for the GIF's frame rate and call repaint() on your component when the next frame needs to be displayed.



Palette animation (aka colormap animation) works only with palette-based pixel models. This kind of pixel model typically uses 8 bits per pixel (the GIF file format is actually a good example of a palette pixel model). Each 8-bit pixel is an index into a 256-entry palette (each entry has a 24-bit RGB value). Graphics hardware does the translation, looking up each pixel's 8-bit color index in the palette to find its true color. Any colors can be in the palette -- doesn't have to be 256 distinct colors.

The principle of palette animation is that the palette can be changed very rapidly – replacing just 256 entries, rather than thousands or millions of pixels on the screen. As a simple example: you can make the entire screen fade to black by animating the palette colors from their original values down to (0,0,0).

To animate only certain objects, not the entire screen, you split the color index into separate bitfields, each representing a different layer. For example, suppose we had two 4-bit layers, aaaa and bbbb. Then just changing the palette can make whole layers appear and disappear:

show only layer A: [aaaa bbbb] => color of aaaa

show only layer B: [aaaa bbbb] => color of bbbb

This lets you do double-buffering in place (not in separate memory): you draw in the layer that's not showing (i.e., you change only the 4 bits of each pixel corresponding to the invisible layer), then very quickly make it appear by updating the palette.

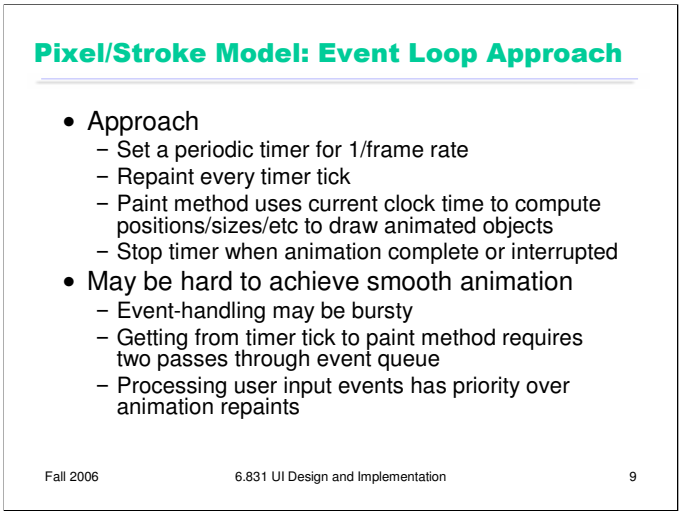
You can also stack the layers on top of each other, and reserve a color (say 0) in each layer to mean *transparent*, so that the layer below shows through:

[aaaa 0000] => color of aaaa

[aaaa bbbb] => color of bbbb (if bbbb != 000)

This lets you move objects around in the upper layer and simply *erase* pixels when the object moves – e.g., set the upper 4 bits to the transparent color. You don't have to redraw the damaged region in the lower layer, saving a lot of repainting.

Palette animation comes with a significant cost in available colors for drawing (i.e., # of distinct colors on the screen at the same time). It only makes sense if the palette is small relative to the screen size, and after you split it into layers and introduce a transparent color, you're left with only a handful of colors for your actual drawing. Palette animation isn't used much today – processors are much faster, and the sacrifice in color depth isn't worth the benefit.



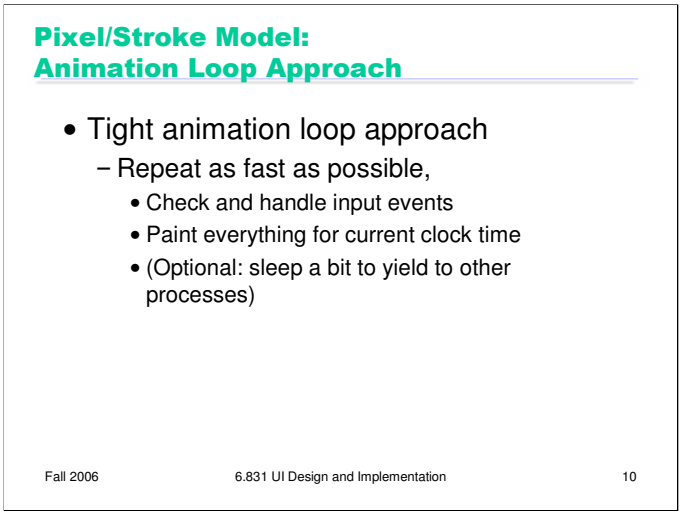
Suppose you can't use animated GIFs. How do you animate changes to a pixel model or stroke model output?

When animation is merely used a feedback or help effect in an application that otherwise isn't concerned with animation, the best solution is the **event loop approach**. The basic idea is to use a timer object (such as javax.swing.Timer), which delivers periodic events to the GUI event queue. Set the timer interval to the desired frame rate (e.g. 50 msec for 20 fps). Every time the timer fires, use the current clock time to determine where to redraw the objects. Stop the timer when the animation is done.

This technique integrates very well with an existing GUI application, because automatic-redraw and input-handling support of the GUI toolkit continue to be supported even while the

But this is also the main drawback of the event-loop approach. Event handling is often bursty, and user input events generally get priority over repaints, so the animation may be jerky if the user is (e.g.) typing or waving the mouse around. Also, getting from the timer tick (when the timer fires) to actually updating the screen (painting the next step of the animation) requires **two** passes through the event queue: first the timer event is put on the queue, and then it's handled merely by putting a repaint event on the queue.

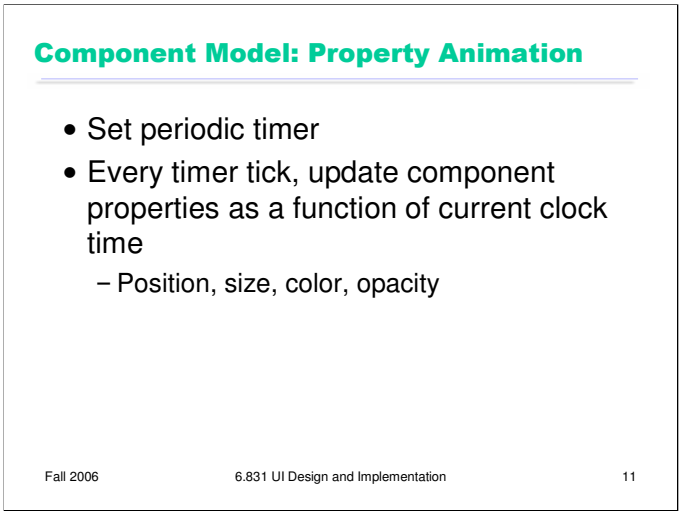
So animation has low priority in an implementation like this – but that's appropriate if it's just used for feedback.



Applications that demand smooth, high frame-rate animations often forgo the standard GUI event loop and write their own tight animation loop. This custom loop can give greater priority to the animation, and it can avoid overhead like managing damage rectangles by just assuming that most of the screen will have to be repainted every frame. It can also maximize the frame rate for the user's processor, simply by running at top speed, without having to configure a timer with a fixed frame rate.

A tight animation loop like this often used for games and simulations, but it doesn't mesh well with the component model of a standard UI toolkit, which expects you to use their event loop and automatic redraw. So programs that use tight custom animation loops often do *not* use standard widgets from a toolkit.

Don't try to mix these two approaches. In particular, don't try to implement animated feedback by putting a tight animation loop inside an event handler. (Why not? What will happen?)



The component model enables a more modular approach to animation. Rather than putting animation code into the paint method, we can simply update the changing properties of the component (position, size, etc.) on every timer tick. So the animation effect can be entirely decoupled from the component itself (as long as the component exposes properties for all the visual effects we might need to animate). This technique is called **property animation**.

Pacing and Path

- Pacing function maps time t to parameter s [0,1]
  - Linear: s = t / duration
  - Slow-in/slow-out: s ~ atan(t)
- Path function maps s to property value v
  - Linear: (x,y) = (1-s)*(x0,y0) + s*(x1,y1)
  - Quadratic Bezier curve: (x,y) = (1-s)^2*(x0,y0) + 2s(1-s)(x1,y1) + s^2(x2,y2)
  - Color: HSV vs. RGB

Pacing and path are relevant to animating both stroke and component models.

Pacing concerns how the animation evolves over time. It's a function that maps clock time to an abstract parameter, usually 0 to 1, representing the completeness of the animation (0% to 100%). Pacing is how you can implement slow-in/slow-out. Slow-in/slow-out is done with a sigmoid (S-shaped) function. The arctangent is a good, easy sigmoid; so is 1/(1+e^-x). Note that you have to tweak the domain and range of these functions so that the desired time domain (0 to the duration of the animation) maps to the desired s-parameter range (typically 0..1). Normally arctangent maps the domain [-inf, +inf] to the range [–PI/2, PI/2].

Path describes how the animated property moves through its value space. For position, this is easy – it's the curve of points that the position traces out. If you want to add some visual appeal to a moving object, make it move through an **arc**. A quadratic Bezier curve has this effect and is trivial to implement – you just need a control point between the start point and end point. (The control point pulls the curve away from the straight line between the endpoints – to be precise, the curve is tangent at each endpoint to the line that connects the endpoint with the control point.)

For 1-dimensional properties, like size or opacity (alpha value), it's just a starting point and an ending point. For multidimensional properties like color, you'll want to think about what color space the path should go through, and whether you want the path to be simply a line in that color space or something more complicated. For color animation, it may be more effective to animate in HSV space.

Declarative Animation in XAML

```
<Rectangle Fill="Black"
           Height="100px" Width="100px"
           Canvas.Bottom="5px"
           Canvas.Right="5px">
  <Rectangle.Height>
    <LengthAnimationCollection>
      <LengthAnimation From="100" To="50"
                       Duration="3"
                RepeatDuration="Indefinite" />
    </LengthAnimationCollection>
  </Rectangle.Height>
</Rectangle>
```


Windows Presentation Framework (a new GUI toolkit that will be included with Windows Vista) supports declarative specification of property animations like these. Here's a bit of XAML code (WPF's declarative user interface language) that describes a rectangle whose height will animate between 100 and 50 over the course of 3 seconds.

For more about WPF, see

http://msdn.microsoft.com/windowsvista/default.aspx?pull=/library/en-us/dnlong/html/introwpf.asp