Advanced Drawing

JComponent vs. JPanel

Simple component that just draws itself
- Subclass off JComponent, don't need to call super.paintComponent().

Component that draws every pixel in its bounds (is opaque)
- Subclass off JPanel
- setBackground() as needed to get an automatic background color
- setOpaque(true) -- tells the system to not bother drawing what's behind us, since we are drawing every pixel
- Call super.paintComponent() from paintComponent()
- The Graphics will be erased to the background color

Clipping

System sets a "clipping region" on the Graphics object before sending paintComponent().

The clipping region affects all drawing operations on the graphics objects -- drawing operations that fall outside the clipping region do not set any pixels.

By default, the clipping region is the bounds of the component, so the component does not draw outside its bounds.

For basic drawing, a component does not need to pay an explicit attention to the clipping region. The component just draws what it wants, and any drawing commands that fall outside the clipping region will be clipped automatically.

The system can use the clipping region to optimize the drawing in cases we will see later.

component.getGraphics() -- NO

Almost always incorrect to use this

Only getGraphics() if the assignment handout specifically says to

getGraphics() goes against the system / paintComponent paradigm

Repaint Details

1. Repaint -- region to draw

Repaint() tells the system that an area on screen needs to be redrawn

Repaint() is sent to a component, but the command to draw is translated to a region -- typically the bounds of that component.

component.repaint() -- specifies the entire bounds of that component

component.repaint(<rectangle>) -- specifies a sub rectangle inside the component

2. Repaint -> Update Region

The system maintains a global "update region" -- a 2-d representation of areas that need to be redrawn.
Repaint -> adds a region to the update region

3. System paint thread

1. Notices non-empty update region
2. Compute intersection of that region vs. components
3. Initiates draw recursion down the component nesting hierarchy. Composites the pixels together back-to-front.

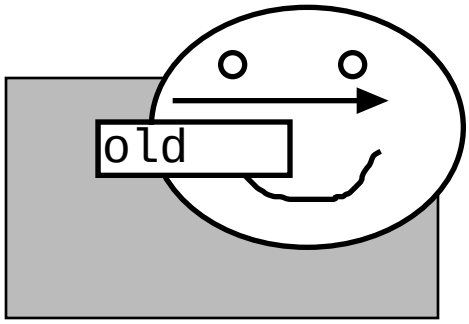
Region Based Drawing

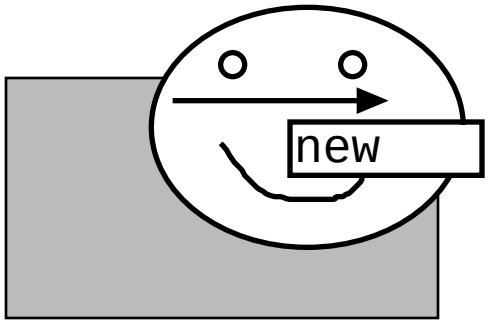
The need to draw something is always expressed in terms of **regions** of pixels, not just components.
This scheme deals with intersection and z-order correctly

Overlap

Draw all the components that intersect the pixel region that needs to be redrawn.
Draw the components from back to front.

Move Component -> old+new





Suppose we have a component in front of a smiley face. The component moves to a new position to the right-- What needs to be redrawn?

Both the old region and the new region of the component -- the old region needs to be drawn with the component not there.

Smart Repaint

Smart repaint = repaint just the rectangle of the component that needs to be redrawn, not the entire component or window bounds.

This makes the following draw cycle faster, so we get faster, smoother drawing. This can be quite a speedup, if the smart repaint rectangle is significantly smaller.

There's a version of the repaint() method that takes a rectangle argument, and just repaints that rectangle (rather than the component bounds) -- component.repaint(x, y, width, height)

e.g. -- just repaint the old+new rectangles when a component moves.

The system gets this right automatically when moving components around with, say, a JPanel. See the setBounds() source code -- repaints just the old+new regions.

Coalescing

Using repaint() to make redraw requests gives us the advantage of "coalescing" -- intelligently combining multiple repaint() requests into a single draw operation.

There is not a 1-1 correspondence between repaint() and paintComponent() -- mulitple repaints are coalesced by the system and handled by a single paintComponent().

Time: Multiple repaint requests for a region in quick succession are "coalesced" into one draw operation. You can repaint() 3 times in succession, but it just draws once.

Space: repaint regions can overlap, but the area of intersection is just drawn once.

Coalescing Example - JSlider

Consider the JSlider / MyComponent example

When the JSlider moves, it sends a setCount() to the widget, which does a repaint()

Suppose we move the slider quickly -- generating three setCounts(), 10, 11, 12, 13 in quick succession, resulting in 4 repaint() calls.

This does not mean we need to draw the MyComponent 4 times. If we did, all but the last would just be overwritten anyway -- a complete waste.

The 4 repaints() can be coalesced into a single draw, if they are close enough together in realtime.

Blinking Animation

Animation steps

The drawing process will have to go through something like the following three states...

1. Old appearance on screen
2. Erase back to background
3. Draw new appearance
   - May be several steps here as we composite together several components back to front.

Problem: Blinking

Blinking
- Get blinking if "erased" state is shown on screen

Shimmering
- If the redraw is really fast, it may look more like a "shimmer" of vibration, but it's still not good.

Solution : Double Buffering

Offscreen Buffer
- Build a pixel buffer offscreen

1. Old appearance (as before)
2. Erase offscreen buffer
3. Draw new appearance to offscreen buffer
   - As before, there may be several steps here as we composite together several components back to front.
4. CopyBits (aka "Blit")
   - Copy the pixels for the new appearance from the buffer onto the screen.
   - It looks smooth because the "erased" state is never on screen -- on screen we move right from old to new.
   - Also, the CopyBits primitive is highly optimized.

Swing
- Swing double buffers automatically -- all JComponent drawing goes through an offscreen buffer.
- JComponents draw to an offscreen buffer -- that's what the "g" passed in points to
- That's why tetris looks smooth even with our simple drawing strategy (draw the whole board as the piece falls)

Inside Repaint 2

Smart Repaint -- smaller region

Smart repaint: component.repaint(rect) -- just redraw the given sub-rect of the component.

e.g. Tetris falling piece. Don't redraw the whole board, just request the region where the falling piece is.

Smart repaint implementation

1. Start with a region to redraw, but now its smaller
2. Find intersection components (as before)
3. Allocate an offscreen bitmap -- but exactly the size of the (small) update region.
4. Set up the origin and clip of Graphics g, to point to the small buffer.
5. Call the drawing recursion as usual, the components draw into the small offscreen buffer. Drawing outside the buffer is clipped, but the components don't need to be aware of that.
6. CopyBits the small buffer to the screen when done -- this can be much faster if the buffer is much smaller than the whole bounds. The number of bytes to move is just a lot smaller.

Tetris Example

The OPTIMIZE constant in JTetris controls this feature in the code I gave you. When moving a piece, does it repaint the whole board, or just the small area around the piece. A huge speedup.

Conclusion

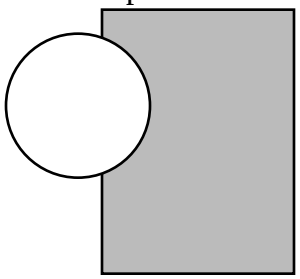
repaint(rect) to redraw just an area of the component can be a lot faster.
The client components can be unaware of the sophisticated drawing that's going on. They just obey the paintComponent() / repaint() contract.

repaint(x, y, width, height)

Call to redraw just a sub-rectangle of a component
The system is smart about using an offscreen buffer of just the size needed -- great potential speedup.
Theme: with a little cooperation from the client side, the JComponent system can do quite sophisticated drawing.

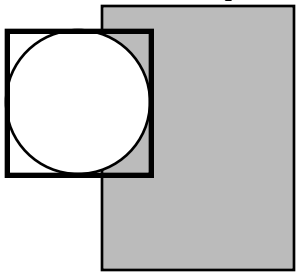
Example 1

Suppose there's a circle and a rectangle. We're changing the circle so it is filled with a pattern.



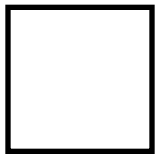
1. State change -> Repaint -> Update region

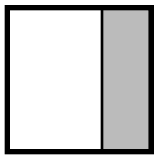
Change the state of the circle to pattern=true.
Repaint just around the circle.
This adds the square shown to the update region

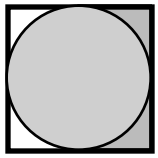


2. Offscreen Drawing

A little later, the draw thread notices the non-empty update region
Draw thread creates an offscreen buffer just the size of the update region.
Notice, for example, how many fewer pixels need to be erased compared to redrawing the whole component -- potentially huge speedup.
Clipping is set around the buffer, so drawing outside of there has no effect (and is fast).
The draw thread sends paintComponent() to the components to draw themselves back to front. Only the parts that intersect the update region actually draw.

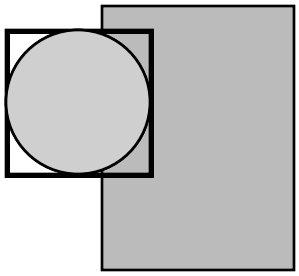






3. Copy Bits

Once everything has drawn the buffer, the draw thread copies the buffer back onscreen with one fast copybits operation ("blit") and deletes the buffer.

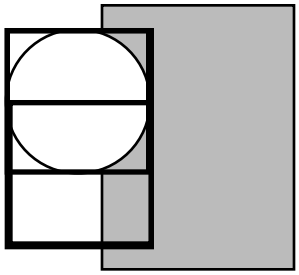


Example #2

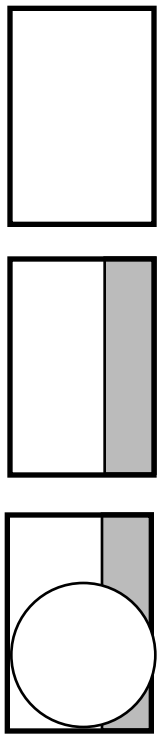
Basically the same thing, but moving the circle instead of filling it with a pattern.
Moving something requires two repaints -- one for its old rectangle and one for its new rectangle.
The update region is smart about "unioning" the two rectangles together to make one enclosing rectangle.

1. Repaint x 2

Move the circle down
Repaint its old and new rectangles



2. Offscreen Drawing



3. Copybits