CS193j, Stanford Handout #32
Winter, 2001-02 Nick Parlante

Java Advanced

Today: How does repaint really work? Why does our animation look smooth? How is java used currently, and how can it be deployed to regular users? How might it be used in the future?

Inside Repaint v1

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 does this automatically
  JComponents draw to an offscreen buffer -- that's what the "Graphics g" passed in points to
  That's why shape move/resize looks smooth despite our simple drawing strategy

Inside Repaint v2

Smart Repaint -- smaller region

component.repaint(rect) -- added **rect** argument in the co-ord system of the component.
Do not request the whole bounds of the component, just a sub-rect of it
e.g. Moving a shape -- just redraw the old+new rects around the shape.

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.

Conclusion

Using smart repaint(rect) to redraw just an area of the component can be much faster.
The smaller region requires less drawing -- some speedup
The smaller region means that the offscreen bitmap can be smaller and there are fewer pixels to copy -- this is a large part of the speedup.
The client components can be unaware of the sophisticated drawing that's going on under the hood. They just respond to paintComponent().

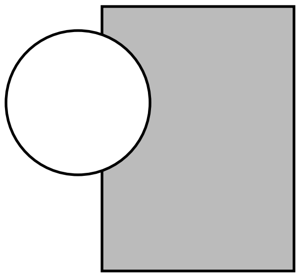
Repaint Examples

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.

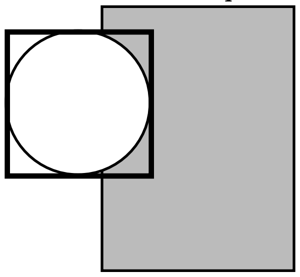
Example 1

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



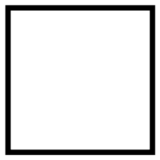
1. State change -> Repaint -> Update region

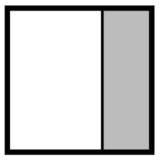
Change the state of the circle to pattern=true.
Repaint just around the circle.
This adds the square shown to the update region

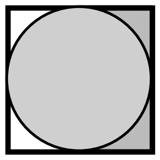


2. Offscreen Drawing

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.

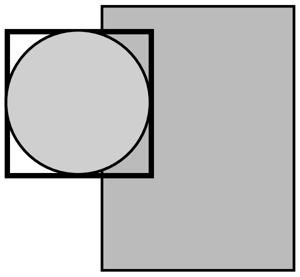






3. Copy Bits

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

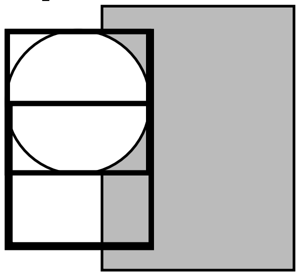


Example #2

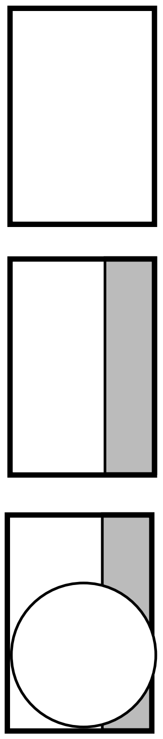
Basically the same thing, but moving the circle instead of filling it with a pattern.
Moving something requires two repaints -- one for its old rectangle and one for its new rectangle.
The update region is smart about "unioning" the two rectangles together to make one enclosing rectangle.

1. Repaint x 2

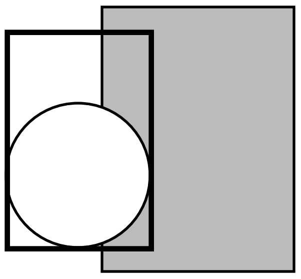
Move the circle down
Repaint its old and new rectangles



2. Offscreen Drawing



3. Copybits



Java Niches

Java Niches -- Server vs. Client

Niche: server-side internet apps -- Java is very popular here already -- portable, secure, programmer efficient -- show well in this niche

- "Business logic" applications using Java and its JDBC library to connect to the database and fiddle around with the data. Note: possibly no GUI, just strings, ints, dates, etc.

Niche: "custom" applications

- A custom GUI application that is part of a larger custom system -- e.g. the "View Order Status" application used by the foo.com customer service people

Possible niche: Client side java

Possible niche: Small devices -- palm pilots, TVs, ...

HTML Forms Are A Hack

Currently,.almost all net services, (e.g. Amazon, yahoo email, ...) are presented through HTML forms.

This has the huge advantage of compatibility -- it works with most any client OS, and such platform-independent compatibility has been the key ingredient in the growth of the internet.

- Note that the Internet did not develop exchanging proprietary .doc files, even though 90% of users have MS Word -- the Internet explosion really kicked in with 100% portable formats such as HTTP and HTML.

However, HTML forms do not present a great interface -- the user sees a state, they can click a button, there is a 2 second delay, and they see the next state.

Contrast this to a real GUI program -- you move the mouse or scroll a list, and 1/100th a of a second later you get the visual feedback.

We are so used to HTML forms, we have grown blind to how lame they are for constructing a good UI.

Future: Real Client GUI

Imagine Amazon client program

Runs on the client side

Communicates back to the server as needed

Presents a responsive GUI to the client -- lists, text fields, selections etc.

Still limited by networking speed, but can be far better than the HTML form

Applets

Run in a security "sandbox" in the browser -- prevent the applet from touching the local file system, etc.

Applets have not caught on too much

Performance problems

Running inside the browser created inevitable reliability problems

Microsoft is not, shall we say, enthusiastic about making applets work correctly in the browser.
Original applets used AWT
With the latest Java 1.2 or later installed on a machine, the Swing JApplet may be used -- the browser must be set up to support Java.
Sun's "java plugin" is a browser plugin that provides applet support.

Jar files

.jar file is an archive file that contains directories of .class files + misc images, sounds, and other support files.
Double click on the .jar runs the application (works on windows, solaris, and MacOSX)
Users need to install Java first -- the Java Runtime Environment from Sun (JRE)
Code does not run in a "sandbox"
It's easy to package your java application into a .jar file -- then you can distribute it as simply as a PDF. Users just download the file and double click it.

Java Web Start

Replacement for applets and jar files
http://java.sun.com/products/javawebstart/
Client installs the JWS loader on their machine once (like installing Acrobat)
Package app in a .jar
Write a .jnlp file that describes the package -- where's one I made for a DiceMachine class in a DiceMachine.jar file (made by tweaking the Sun sample)...

```
<?xml version="1.0" encoding="utf-8"?>
     <!-- JNLP File for SwingSet2 Demo Application -->
     <jnlp
       spec="1.0+"
       codebase="http://www.stanford.edu/~nick/dice/"
       href="dice.jnlp">
       <information>
         <title>Dice Application</title>
         <vendor>Sun Microsystems, Inc.</vendor>
         <homepage href="docs/help.html"/>
         <description>SwingSet2 Demo Application</description>
         <description kind="short">A demo of the capabilities of the Swing Graphical
User Interface.</description>

         <!-- <icon href="images/swingset2.jpg"/>   -->
         <offline-allowed/>
       </information>
       <security>
<!--  we are not signed, so we don't request all perms  <all-permissions/>   -->
       </security>
       <resources>
         <j2se version="1.2"/>
         <jar href="DiceMachine.jar"/>
       </resources>
       <application-desc main-class="DiceMachine"/>
     </jnlp>
```

Unsigned code runs in a sandbox
The client just downloads the .jnlp file which points to enough info for the client to download and run the java code.
Can run with or without a net connection once downloaded.
Can check for updates automatically
The point: You send someone just a URL, and they can just click it to run the program on their machine. Updates can happen automatically.

Will JWS Catch On?

Like Flash catching on -- chicken-and-egg problem that works best if many clients have it pre-installed.
This will be hard since Microsoft controls the dominant OS and browser, and Microsoft hates Java
Enterprises love it internally -- easy way to distribute and update little custom apps -- just send out the URL