Graphics and Skia

Chrome uses Skia for nearly all graphics operations, including text rendering. GDI is for the most part only used for native theme rendering; new code should use Skia.

Also see the section RenderText and Chrome UI text drawing for a more specific discussion of Chrome UI text rendering.

Why Skia?

- GDI is not full-featured enough for things like SVG, Canvas, and some of the complex UI things we have in mind.
- Skia is in-house and can be modified to suit our needs.
- There was already a high-quality Webkit/Canvas port to Skia.
- GDI+ is no longer in development by Microsoft and is slower than Skia on most operations, at least on XP. On Vista, it seems that at least some GDI+ operations are hardware accelerated, but software Skia is generally fast enough for our needs.
- GDI+ text rendering is unacceptably different from GDI text rendering (see GDI+ Text, Resolution Independence, and Rendering Methods), and slow to boot.

Background: What's with all these Canvii?

The lowest levels

There are a confusing number of Canvas classes in use. At the lowest level is SkCanvas (in third_party/skia/include/SkCanvas.h, it is pretty well documented and you will want to refer to this file a lot in normal usage). This object contains all the methods that are used to draw. Canvases draw into an internalSkDevice, which is a simple wrapper around an SkBitmap object that actually holds the bits.

In addition, we have a platform-specific wrapper around the native Skia canvas called SkPlatformCanvas, which allows using platform-specific APIs to draw things such as themes. Its backing store is a SkPlatformDevice, which includes memory that is shared by Skia (who writes into it directly) and Windows, where we create an HBITMAP for the bytes and draw into it using an HDC wrapping the bitmap. In this way, we can use Skia for normal rendering and Windows for theme drawing.

The SkPlatformDevice has two functions BeginPlatformPaint (which returns an HDC you can draw into) and EndPlatformPaint that are used to wrap sections of native drawing. They manage the necessary flushing that allow Skia and GDI to see a consistent bitmap. Be very careful when rendering to the returnedHDC, see comments below on this.

The application layer

For the Webkit layer, SkCanvas and SkPlatformCanvas are the only objects that are used. Callers must manually set up for text drawing, which is OK because all graphics operation go through one graphics layer, GraphicsContext that we have ported to use this scheme.

If you are writing code that will run in the browser process, for example, that draws the tab strip or one of the Destinations views, there is a higher-level option. At this layer, we also support gfx::Canvas (in ui/gfx/canvas.h) which wraps a SkCanvas. It provides text-drawing routines that provide richer functionality than the ones on SkCanvas and also wrappers around some of the common graphics primitive routines that take integer values instead of SkScalar coordinates which are more handy when integrating with most of our graphics code (see "Scalar values" below). These integer wrappers are named with "Int" appended, and a capital first letter (we use Google coding styles, Skia uses a different one for historical reasons) for example, DrawRectInt.

One thing to note is that SkPlatformCanvas is heavier weight than SkCanvas because it has a Windows HDC and an HBITMAP associated with it. If you are doing rendering that does not need to be copied to the screen, you should use an SkCanvas. An example would be a temporary buffer used to render a bitmap into, which is then resampled and composited into the final SkPlatformCanvas before being copied to the screen.

Which canvas should you use?

If you do not need to call GDI operations and only copy the canvas to another canvas instead of the screen, use SkCanvas directly. It will be the fastest.
If you are in the renderer, you must use SkPlatformCanvas if SkCanvas is not enough for you. Inside Webkit, though, you should always use its interface, GraphicsContext, instead.
All browser-level code that needs more functionality than SkCanvas should use gfx::Canvas. The Views system uses only gfx::Canvas except for intermediate results as described above.

Scalar values

Skia uses a custom type, SkScalar, for most of its routines to represent coordinates. Depending on the value of a compiler flag, this is either typedefed to a float for floating-point mode, or an int for fixed-point mode. In fixed-point mode, the high 16-bits represent the whole part of the number, and the low 16-bits represent the fractional part. Although we currently compile with floating-point mode, we want to leave open the option to move to fixed point mode. You should therefore be very careful and not pass integer or floating point values directly to these functions.

The easiest way is to use the *Int functions provided in gfx::Canvas if you are using one of the commonly-used functions that have wrappers. If this is not an option, you must use either SkIntToScalar or SkFloatToScalar to convert your value (an int or float, respectively), to the appropriate SkScalar type. To convert from scalars back to ints, use SkScalarRound.

The epic battle of RGBA and BGRA

Both Windows and Skia use RGBA color in each 32-bit word. When written to memory on a little-endian machine like a Pentium, this becomes BGRA. While we are fortunate they agree, this still causes some problems when getting data in and out of the system. Many graphics libraries such as libpng and libjpeg expect the data to be in RGB(A) order in memory regardless of machine endianness.

The result is that we have to be careful to swizzle the bytes when interfacing with these systems. Our application-level graphics codecs such as

PNGEncoder,PNGDecoder, and JPEGCodec (these are not used by Webkit), support this.

Opacity

Using both Windows and Skia to draw into a single buffer is great because it gives us the best of both. Windows, however, doesn't know about the alpha channel. Most GDI operations treat the high 8 bits of each 32-bit word as padding.

Premulitplied alpha channels

It is important to know that Skia uses premultiplied colors. This means that the color components have already been multiplied by the alpha channel. Most people are accustomed to having a color and then thinking about a separate opacity value. Premultiplied colors have already been "composited on black," allowing layers to be composited by addition of the color values rather than requiring an additional multiplication by alpha. Therefore, if the alpha is 50%, no color channels can have values greater than 50%.

Since most graphics file formats assume postmultiplied alphas (color values plus a separate opacity of that color), we have to be careful to convert properly between the two formats when reading and writing to these systems.

Use SkPreMultiplyColor to convert from an SkColor to a premultiplied color value (SkPMColor).

The case of the disappearing alpha

Windows causes some problems by not knowing about the alpha channel. When drawing with GDI, it inconveniently sets the alpha channel to 0 (transparent). We have two approaches to dealing with this, one for the Webkit port, and the other for the UI. We may merge or enhance these in the future.

Transparency in the Webkit port

Web pages do not require arbitrary transparency to render properly. The base of the web page is fully opaque. Some image formats can support alpha channels, but these images are composited on top of the opaque background and require no special handling. The only way that a web page can create transparency is by setting a global transparency on an element. Therefore, we don't need to support arbitrary transparency, only compositing layers with a 1-bit mask to separate out the totally transparent regions from the (possibly partially) opaque regions.

New SkPlatformDevices with transparency will have their values filled with a color that is impossible using premultiplied alphas. Webkit draws like normal over the top of this using opaque colors, potentially clearing the alpha channel using text drawing routines. When the layer is being composited, we do color replacements in fixupAlphaBeforeCompositing to reconstruct the correct alpha channel and composite. The thing to realize here is that this operation is destructive and the device will be invalid if used again.

Transparency in the UI

In the UI, we often want to have more complex alpha channels. We give finer-grained control using prepareForGDI and postProcessGDI which are called for a rectangular region before and after GDI calls that may mangle the alpha. It does a similar replacement of transparent values to impossible colors, and then back. It assumes that all pixels written by GDI are supposed to be opaque. If you want more complex operations, you will have to do the rendering to an intermediate buffer and do separate compositing to get what you want.

Coordinate transforms

Currently, SkDevice syncs Windows' coordinate transform to the current Skia transform whenever you call BeginPlatformPaint. This means that you can call GDI operations without worrying about the current transform (unless you have set up a perspective transform, which Windows doesn't support).

However, this will eventually change. The problem is that Windows does a surprisingly bad job handling transformed graphics operations. For example, in text drawing, character spacing is expressed in terms of integer values in the untransformed coordinate space. If your transform has a positive scaling, the characters will become larger, but there will be visible rounding errors in the character spacing and the characters will appear uneven. Drawing themed controls is even worse, as the nearest-neighbor resampling Windows does will cause parts to expand or contract unevenly.

The eventual solution will be to never transform the GDI coordinates and manually compute the new positions and sizes of text and themed controls that we draw. In some cases, we may need to render to a temporary bitmap and do the resampling ourselves.

Useful tips

The SkBitmap class is internally reference counted. To prevent memory errors, it is generally best to copy the objects rather than hold pointers to them. These copies are lightweight so you should not be concerned about performance.

Sometimes you want to temporarily override the transform, clipping, or other canvas parameters for a function. Use the SkCanvas.save() function, which will save the current state on an internal stack. Use SkCanvas.restore() to pop the stack and go back where you were.

When you want to have a partially transparent layer of stuff, you can use the SkCanvas.saveLayerAlpha() function. This will create a new "layer," which will look like an empty, fully transparent canvas that you can draw into. In addition, this saves the same state as save(). Calling restore() will composite the layer back onto the previous layer with the transparency applied that you specified when saving it. If you do not need a global opacity, use thesaveLayer() function (it will still obey the layer's alpha channel).

Clipping regions have one-bit precision. This means that you will not get antialiased edges if you have curves in your clipping region. If you want to draw something with antialiased clipping, and you can get your contents into a bitmap, you can use a bitmap shader to fill a path instead. Filled paths have nice antialiased edges.

Using Skia

The basics

The model is that you use an SkCanvas to draw into an SkBitmap with an SkPaint. The SkBitmap (wrapped by an SkDevice) holds the bits, the SkCanvas manages the device and contains the drawing primitives, and the SkPaint holds most of the graphics state such as colors and antialiasing options. Most other graphics libraries combine the graphics state inside the canvas. SkPaint is lightweight and you should not generally worry about creating and destroying them as needed.

To draw a round rect:

```
// Create the canvas. Note that this is a SkCanvas and not a
PlatformCanvas so
// we won't be able to draw text on it. The canvas first
requires a backing store
// (SkPlatformCanvas doesn't need this since it creates its
own backing store).
SkBitmap backing_store;
backing_store.setConfig(SkBitmap::kARGB_8888_Config,
canvas_width, canvas_height);
backing_store.setIsOpaque(true); // This will make some
things faster if it's opaque.
backing_store.allocPixels(); // Don't forget to call this or
it will be invalid!


SkCanvas canvas(backing_store);
canvas.drawARGB(255, 255, 255, 255); // Fill with white.


// Make a rect from (0,0) to (100,75).
SkRect rect;
rect.set(SkIntToScalar(0), SkIntToScalar(0),
SkIntToScalar(100), SkIntToScalar(75));


// Create a path with that rect and rounded corners of
radius 10.
SkPath path;
path.addRoundRect(rect, SkIntToScalar(10),
SkIntToScalar(10));


// Fill the path (with antialiasing) in 50% transparent
green.
SkPaint paint;
paint.setStyle(SkPaint::kFill_Style);
paint.setAntiAlias(true);
paint.setARGB(128, 0, 255, 0);
canvas->drawPath(path, paint);
```

Using the platform canvas

There is a handy class called SkPlatformCanvasPaint to help you handle WM_PAINT messages. It will automatically do BeginPaint and EndPaint on init/destruction, and will create the bitmap and canvas with the correct size and transform for the dirty rect. The bitmap will be automatically painted to the screen on destruction.

```
case WM_PAINT: {
  SkPlatformCanvasPaint canvas(hwnd);
  if (!canvas.isEmpty()) {
    ... paint to the canvas ...
  }
  return 0;
}
```

Working with a writable Skia tree

Add this to the custom_deps the .gclient file:

```
      "src/third_party/skia":
"https://skia.googlecode.com/svn/trunk",
      "src/third_party/skia/gyp": None,
      "src/third_party/skia/src": None,
      "src/third_party/skia/include": None,
```

This gets you a writable tip-of-tree skia inside chrome. For a specific revision put @<rev#> after trunk in the URL.

Comments

You do not have permission to add comments.