Problem statement

Traditionally UIs were designed and laid out in pixels, meaning on-screen hardware pixels. So, things like button border widths, text heights, padding, etc would be different for different screen resolutions. However, the display sizes and densities were such that these sizes while not identical across different setups were at least usable. However, some modern displays have a very high density (HiDPI), one the order of double the traditional density (which is generally around 100dpi for normal monitor, although much less for e.g. a projector). At this density traditional UI layouts start to become so small that they are actually hard to use.

Traditionally we have allowed changing the text DPI (the mapping between app specified font sizes in points and on-screen pixel size) to modify the size of text in the UI, thereby changing the size of the UI to ensure that text is readable on higher density screens (or for people with worse eyesight). However, this is not really enough to handle HiDPI screens, as the non-text objects (padding, borders, icons, etc) are not handled by this and are now so small that its hard to use them.

Further complicating this is the fact that most external monitors are not HiDPI, so setups with a mix of HiDPI and non-HiDPI monitors will be very common, and we have to handle this case well.

Scaling

The solution to this is pixel scaling. For sufficiently high resolution outputs (we can autodetect this and let the user override it) we pick a scaling factor (> 1) and apply that to the window contents. This essentially separates the “window coordinates” (window positions/sizes, event coordinates, etc) from “hardware pixels” (what is drawn on the monitor) into two different coordinate spaces (by a simple scale transformation).

To the application using the higher level APIs it would appear like the window was smaller by the scale factor, and it would draw everything as if it was a regular density display. But, internally (e.g. in cairo) the scaling factor would be applied to all drawing operations, such that the rendering would be higher resolution, essentially giving you subpixel rendering precision. Also, events would have subpixel precision coordinates (they generally already have this to support e.g. wacom tablets).

While it's technically possible to scale rendering by any amount we want to limit ourselves to integer scaling factors. Non-integer scaling factors are problematic for the common case of horizontal/vertical lines of integer pixel widths, as well as pixel based line-art images If these are drawn scaled or on non-integer positions they will look very ugly. It also becomes problematic to handle widgets/windows or similar rectangular regions that are positioned next to each other, as in the case of non-integer scaling they may share a pixel row making clipping much more complicated.

In order to simplify the discussion, lets define names for the two coordinate spaces. Following the OSX HiDPI docs we use pixels to refer to the hardware pixels, and points to window/user/layout coordinates.

Each monitor has a resolution and a backing scale factor that converts from points to pixels. For instance, a MacBook pro 13” would have a resolution of 2560x1600 pixels, or 1280x800 points, and thus a backing scale factor of 2.

Additionally each window will have a size in points and a backing store factor that converts from the window coordinates to the actual pixel backing store for the window (i.e. the buffer shared between the compositor and the client). The compositor would know the scale factor of each individual window and use this when rendering that window on a particular output. For instance, a window with a factor 2 which is partly rendered on a monitor with factor 1 would need to be scaled down when rendering to that monitor.

In general, windows should use the backing store factor that matches the monitor they are “mostly” on (i.e. which overlaps the majority of the window). In the case of cloned outputs with mixed resolutions it is up to the user to select somehow. OSX lets you chose “best resolution for retina or for external (when cloned)” to handle this case. This choice should be signaled to the client somehow so it can do the right choice.

Applications should also be able to handle dynamic changes in the display resolution, so that e.g. if you drag a window to a low-dpi screen we (eventually) switch to a smaller backing store.

Wayland details

Wayland is sufficiently new that we can assume that all toolkits and thus app will respect HiDPI once we specify and implement it. So, we don’t need any kind of automatic scaling of rendering, but can assume the clients will respect the scaling factors when rendering. (Whereas on X we have to run old clients that need help scaling the UI.)

On wayland all buffers for surfaces (windows, cursors, subsurfaces) are allocated and rendered to by the client. This means we need to let the client chose the backing scaling factor for each surface and tell it to the compositor so that it knows how to render it on the outputs. However, the compositor needs to assign a scaling factor to each wl_output which the client can use when deciding the scaling for e.g. a window.

Additionally, all the window sizes, output sizes, event coordinates, opaque/input regions, pointer hotspots, etc need to be transformed for the clients. This is best done in the compositor, as this will be needed to handle the multi-monitor-different-density case (need different input device to event coord scaling depending on what output the pointer is in), as well as allow later extensions of the protocol (i.e. add new things that need to be scaled) and letting us guarantee that e.g. window positions/sizes are integer number of points (otherwise it will be problematic to render them on non-HiDPI screens, etc).

Protocol changes needed:

- Add wl_surface::set_buffer_scale()
- Add backing scale member to wl_output::geometry event. Or if that is not possible in a backwards compatible way, add a separate wl_output::backing_scale event
- Similarly add backing_scale_priority to the output geometry so that apps know what scaling factor to use in the cloned output case.

Additionally, on Wayland we could allow per-output buffers. The clients are notifiied when a surface moves in and out of an output, and can render buffers according to that outputs DPI and other properties, such as sub-pixel orientation or scanout orientation.

X11

X11 support is done via XWayland, which is a rootless Xserver that the wayland compositor starts. Apps connect to this and get their windows redirected as if it was a traditional Xserver with a compositing manager running. The buffers from the redirected windows are handled (mostly) like a normal surface in the wayland compositor. The Xserver is somewhat limited, so It probably doesn’t handle XRandR, but it gets told the monitor resolution and layout by the wayland compositor and apps can read that out via e.g. Xinerama.

There is two ways we can support X11 apps:

- Backwards compatible with automatic upscaling on HiDPI screens
- Expose full resolution and tell clients about monitor scalings

We don’t want to be in the business of upscaling core X drawing operation, so fundamentally this will require a compositing manager to work, but this is guaranteed with XWayland. This makes the first method is really simple. We make the resolution and coordinates XWayland reports be in points, and have always allocate surfaces for the XWindows using scaling factor 1.

The second model is a bit harder. Its unlikely we could support a system with dynamically changing backing store scale factor for already mapped window with old clients. So, we need a bit more static version.

We use the scale factor of the monitor with the largest scale factor and multiply all point sizes with that. Then we use that scale factor for the backing store of all windows. For instance, in the case with a 2000x2000 HiDPI monitor on the left and an non-HiDPI 1000x1000 monitor on the right we would report both monitors as 2000x2000, but automatically downscale when drawing on the right monitor. We then add an extension that lets you query the actual scaling factor of each xinerama monitor and window.

In the (uncommon) case of a non-HiDPI monitor with existing clients and then a HiDPI monitor is plugged in we will send resize event of both old monitor, but not change the size of existing windows, as that would confuse clients a log. Modern clients could read the new scaling factors and re-realize the windows to properly handle this, but this situation is uncommon enough that not handling it is probably not a problem in practice.

One issue with mode 2 is that certain window positions and sizes are invalid (not a multiple of the scaling factor), at least for toplevel windows (and for foreign windows in clients using a different mode). Hopefully we can just round these up and apps would still work (X WMs can mangle your size request however they want anyway).

So, how do we select what mode to use? I think by default we want to use mode 1, but allow applications to opt-in to mode 2 either by the application using some new extension during initialization, or by the user setting some environment variable to force full resolution for the application.

XWayland (and xlib) changes needed are:

- Extension to enable mode 2 (scale all screen coordinates by max-scale-factor)
- Extension to query monitor/window scaling factor

Cairo

In order to implement scaling of drawing in a way that is invisible to applications we need to set a device scaling factor on cairo_surface_t. This way any cairo_t will get an initial scaling factor that the application cannot accidentally remove. This is already implemented in cairo in the internal function _cairo_surface_set_device_scale. We just need to export this and ensure it works properly.

Also, we probably want cairo_surface_create_similar to create new surfaces with the same device_scale (and large size).

Gdk

Gdk new APIs to report the scaliing factor for each monitor (gdk_screen_get_monitor_backing_scaler()), and each window (gdk_window_get_backing_scale()).

The wayland backend needs code to track the monitor a window is on and pick the scale factor depending on that.

The X11 backend could use the APIs to enable full resolution and do the coordinate mapping itself to make it HiDPI supporting, or we could just recommend using wayland for this.

Gtk

We want a highlevel api for the scaling, like gtk_widget_get_backing_scale(). This is nor generally needed when just drawing a widget, as cairo will scale the drawing for you, but it lets you pick e.g. alternative high-resolution images for pixel data.

GtkImage may need an api to set an alternative high-dpi pixel source. And it needs to rasterize svgs at a higher resolution.

GtkIconTheme needs to have support for high-res versions of icons. We can't just pick the larger icons, i.e the 64x64 instead of the 32x32, because the 64x64 icon will have more detail than the 32x32 one which will look crowded when rendering at the same physical size as the 32x32 icon.

Text layout may be affected by HiDPI. We need to consider how scaling affects this. Behdad wrote an article about this.

Gnome-shell (and Weston)

Obviously this depends on the wayland work happening in gnome-shell. But apart from that we need support for:

- Per-output surfaces, so that you can sanely have a HiDPI monitor next to a low DPI monitor. Its possible to handle this with a shared scanout buffer, but you waste memory and need to play with strides in weird ways.
- Support scaling window buffers when drawing them, depending on target/window scaling factor
- Support scaling cursors, depending on the cursor surface scaling factor and the current monitor output scale
- Push scaling factors to XWayland