Document Number: N3825
Date: 2013-11-21
Author: Jason Zink

SG13 Graphics Discussion
Redmond WA
Meeting Minutes

Attendees:

- Herb Sutter
- Michael McLaughlin
- Jason Zink

Opening discussions

We started with a brief review of Gunnar Sletta's discussion of Qt's API's, based on QPainter and then the scene graph based system.

Andrew Bell's response (via email to Herb) was that we could potentially utilize the SVG specification as a good starting point, and then build the API to fit into that realm. SVG+canvas is a good candidate. This also fits with the advice from Beman Dawes at the Chicago meeting of SG13 to use a known starting point, but perhaps a spec instead of specifically a library.

There is some worry about using an existing standard (such as SVG) since it could be an evolving/moving target. However, this is a common practice in ISO based specifications, where a normative reference is often chosen and then the new specification can choose to stick with the existing normative reference or continue to track it directly.

It seems that SVG could easily be built into an API once the C++ 'style's' are chosen, but Canvas is actually already an actual API. This could be an interesting starting point for the API, allowing familiarity from lots of developers, and in general could provide a good starting point.

What about Cairo (or CairoMM)? It is closer to C++, and it seems to be the intersection between Cinder and OpenFrameworks. It is already very cross platform also, and has been endorsed by several other library developers (from Cinder and OpenFrameworks).

Two approaches seem to be repeating throughout the discussion:
- Use a spec. as the starting point (SVG+Canvas)
- Use a library (Cairo or Cinder or whatever) as a starting point

A general comment is that we are trying to provide access to low level hardware, with an appropriate abstraction level. For example, composition should be a trivially accessible operation with our basic objects for a canvas.

Discussion of current state of Mike's proposal

Description of the implementation concept, which currently uses templates to parameterize the impl. Herb mentioned possibly using a regular polymorphic parameter passed into a constructor instead of using template parameters to define the implementation (this keeps the implementation details from

bubbling up into the type and therefore into the API surface). Or is it even necessary - why not stick to a single implementation at a time?

The proposal also discusses the intention to separate the concerns of drawing and displaying. We can encapsulate the operations of drawing to a surface/canvas, and then all of the different 'display' technologies can consume the result of the drawing. For example, the drawing *targets* a surface, and then it can be *consumed* by a PDFDisplay, a WindowDisplay, PNGFileDisplay, or whatever else. The distinction between render and display seems very useful.

If we were to follow the Canvas example, font would probably be intended to follow the CSS concept. There are some questions about the complexity of implementing all of the options for CSS, but we could always take a subset in the initial stages.

Discussion of Cairo as a starting point

After more discussion about potentially using Cairo as a starting point, we checked the current status of CairoMM (which is a C++ wrapper for Cairo). The project appears to be orphaned, and not actively tracking Cairo anymore, with the last substantive commit in 2010.

Because of this, we inspected the Cairo API itself to see if it would be a suitable starting point. In fact, Cairo seems to be an object oriented (with handles in C) and also const correct API, and the API documentation is high quality and close to standardese completeness. This seems to be a very good potential starting point for a specification. It was also noted again that Cairo was probably the most frequently suggested starting point in group discussion to date, and endorsed by influential participants like Bell.

After discovering these aspects of the Cairo project, we took a first pass through what type of 'mechanical' transformation we would perform to C++-ify the API. The structure of the API didn't present too many challenges, and this went well enough that we actually were able to review the entire API and create a first-cut basic plan for creating a C++ wrapper.

Action Items:

1. Check with the Numerics SG about fixed width int and float, and IEEE754 and IEEE754R status. These may be used in the SG13 specification later on.
2. Find out if the CairoMM project is still active or not (assumed to be DOA - hasn't been updated since Dec.2010!).
3. Contact Cairo to discuss using the documentation in the standard (i.e. copyright restrictions).
4. Create a basic API mechanically transformed from the Cairo implementation.

The following are the (note: very preliminary!) list of mechanical rules discovered so far to transform the Cairo C API to a C++ API.

```
// Use a namespace to hold our definitions
namespace experimental::drawing

// Object conversions
Convert structs named cairo_XXX_t to class XXX or struct XXX for aggregates (e.g.,
struct cairo_surface_t to class surface, structs cairo_rectangle_t to struct
rectangle), and change struct cairo_t itself to class context.
```

```
// Internal ref counting functions
_reference and _destroy functions -> just drop these, as we have shared_ptr

// Constructors
_create function becomes a constructor
anything that returns a new instance also becomes a ctor

// Copying functions
_copy -> copy constructor

// Member functions
f( cairo_t* o, --- ) -> o.f( --- ) // all obvious / simple member functions
f( cairo_1_t*, cairo_2_t*, --- ) -> f( 1&, 2&, --- ) // cases where the there is no
obvious 'this' parameter keep it as a free function.

// Any time you see an item pointer and a length (i.e. an array of items) then it
should become a vector<item> or later on possibly an array_view
cairo_rectangle_list_t -> vector<rectangle>
f( X*, int len ) -> f( vector<X>& )

// User data: Where Cairo allows storing custom user data, instead of allocating an
empty object to use its address as a key/cookie, use a value type and its unique
value instead

// Example user data key implementation:
class user_data_key {
  static atomic<int_fast_64> cnt = 0; // exposition only
  val;
public:
  user_data_key() : val(++cnt) {}
};

x_key_t, x_value -> map<x_key,x_value>

// Error handling: Cairo uses cairo_status_t for error returns, where status can be
Success or some failure value, some of which are precondition violations and some
are run-time errors. As a first cut:

cairo_status_t -> drawing_exception(int == status value) // just wrap a status value
and throw an exception within it. Get the .what() messages from
cairo_status_to_string which already documents the exact message strings

cairo_status_t f( --- ); -> void f( --- ); // any function that returns a status
instead returns void, and throws if status != success

// Debug support: Can probably remove this one memory debug function as not relevant
to our API proposal
cairo_debug_reset_static_data -> (remove)

// Some types are polymorphic -- mechanically changes these to Envelope/Letter.
// Example: A monomorphic “envelope" pattern representation:

class pattern {
```

```
  shared_ptr<cairo_pattern_t,destroy> p; // This shared_ptr's deleter that calls
_destroy on the cairo pattern
public:
  void get_radial_circles( double&... );
  pattern( rgb );
  pattern( rbga );
  pattern( surface );
  .
  .
  .
};

// Basic types (bool)
bool_t -> bool

// Things to look for:
// Conversion of multiple parameters (x,y) to a point type (xy).
```