Document number: N4021
Project: JTC1/SC22/WG21 Programming Language C++/SG13 - Graphics

Date: 2014-05-21

Authors: Michael B. McLaughlin <mikebmcl@gmail.com>
Herb Sutter <hsutter@microsoft.com>
Jason Zink <jzink_1@yahoo.com>

A Proposal to Add 2D Graphics Rendering and Display to C++

Revision 1

I. Table of Contents

II. Introduction

The goal of this proposal is to define a 2D drawing API for the C++ programming language. This proposal is a revision of N3888, which was presented at the meeting of SG13 at the Issaquah meeting in February 2014. This proposal aims to provide a clean, modern C++ API that builds on the N3888 API and incorporates the feedback on it. [*Note*: The N3888 API was a mechanical transformation of the cairo graphics library. Cairo is a comprehensive, cross-platform, widely-used, mature 2D graphics library written in C with an object-oriented style. – *end note*]

III. Motivation and Scope

Computer graphics first appeared in the 1950s. The first recognized video game, *Spacewar*, was created in 1961. Today, computer graphics are pervasive in modern life, and are even replacing console-style I/O for basic user interaction on many platforms. For example, a simple `cout << "Hello, world!"` statement doesn't do anything useful on many tablets and smartphones. We feel that C++ programmers should have a simple, standard way of displaying 2D graphics to users.

Application programmers write programs that often need to render or display basic 2D graphics. Some need to display images. Some need to create simple charts and graphs. Some need to display text beyond the capabilities offered by the C++ console. Many application types call for all three of these capabilities along with the ability to interact with them using keyboard, mouse, and (more and more) touch. Displaying 2D graphics can also make for a compelling educational

experience, since the beginning programmer can have rich visual feedback as a reward for their efforts.

Currently C++ programmers must either use platform-dependent technologies (such as Microsoft's Direct2D with Win32 API input, Apple's SpriteKit, or the X Window System's Xlib library), or else to take a dependency on a third-party cross-platform library (such as Qt, Cairo, GTK+, or SDL). While these are all fine technologies with their own merits (and drawbacks), modern developers should not need to look outside of standard C++ to perform 2D graphics operations which have been pervasive in computing since the mid-1980s.

The audience for this technology consists of developers who need to render or display 2D graphics, regardless of what type (games, business applications, or other). Though the implementation of the proposal require domain expertise, its proposed API aims to be easily understood (and thus easily used) by all levels of C++ programmers, from beginners through experts.

The scope of this proposal is limited to a 2D drawing API. User input to graphical surfaces is not included. The authors believe a standardized way of receiving such user input is important, but felt it would be better to focus on the 2D drawing API first. There are plans to draw up a separate proposal which will propose additional features such as the ability to capture and process keyboard, mouse, and touch input. When unified with this proposal, the result would finally offer C++ programmers a standard way to create basic interactive modern applications, thus vastly expanding the types of programs which can be written with standard C++.

IV. Impact on the Standard

[*Note:* At present this proposal is contemplated only as a technical specification. The discussion which follows assumes that the proposal could eventually be incorporated into the Standard. – *end note*]

The proposal does not require changes to the core language. It makes use of some existing standard library functionality. It does not require any changes to the library functionality it uses. It can be implemented with C++11.

The technical specification presented below depends on the following standard library components:

- function
- string
- vector
- exception
- shared_ptr

V. Design Decisions

A. Leveraging the Cairo API

1. Reasoning

The use of cairo as a starting point allows several goals to be met. Cairo is a C graphics library that provides cross-platform support for advanced 2D drawing and is used in many well-known, widely-used software projects (e.g. Mozilla Firefox, Mono, GTK+). As such, it has demonstrated that it is portable, implementable, and useful. It has a mature API which, though written in C, is nonetheless object-oriented in its design and is even (mostly) const-correct. This allows cairo's C API to be transformed into a C++ API by application of a set of well-defined rules. This mechanical transformation is accomplished mostly through general purpose rules, with only a small number of narrowly tailored rules required to avoid situations where the general purpose rules alone would have rendered ill-formed C++ code.

Starting from a C API rather than a C++ API also provides the benefit of easier potential malleability. While the cairo API is designed in an object-oriented fashion, it accomplishes this through the C idiom of passing an explicit this pointer as the first argument to the relevant stand-alone functions along with indicative naming conventions. The fact that it is a C API makes it conceptually easier to consider and approve changes that reshape the API since such proposals would not involve changing the design of existing C++ classes, which would almost certainly need to be done if the starting point was a C++ library. Indeed, this proposal reshapes the original N3888 API in several critical ways, which are discussed later in this paper. [*Note*: The changes affect form, not function. They do not step outside the bounds of cairo's functionality and as such they let us continue to rely on the cross-platform portability of cairo's functionality while letting us create a cleaner, more modern feeling C++ API. – *end note*]

Taken as a whole, starting from cairo allows for the creation of a 2D C++ drawing library that is already known to be portable, implementable, and useful without the need to spend years drafting, implementing, and testing a library to make sure that it meets those criteria.

2. Alternative: Synthesize a New API

An alternative design would be to create a new API via a synthesis of existing 2D APIs. This has the benefit of being able to avoid any perceived design flaws that existing APIs suffer from. Unfortunately this would not have implementation and usage experience. Further, doing so would not provide any guarantee that design flaws would not creep in.

This could be mitigated some by spending a long time designing the API and using it in real projects. But this is, itself, an argument for starting from an existing API which has already gone through that process. Any design flaws an existing API might have are likely to be known and can be corrected or at least mitigated.

If there were no suitable existing APIs or if other reasons existed that weighed against the propriety of using one, then synthesizing a new API would be a reasonable design decision. But in this case there are indeed existing APIs which are suitable, so a designing new API would be redundant at best.

3. Alternative: Derive an API from the HTML5 canvas API and SVG Specification

An alternative design would be to create an API based upon the HTML5 canvas API that incorporates key concepts from the SVG specification. This has the benefit of being very familiar to programmers who are used to working with JavaScript. The downside of this alternative is that because SVG is designed primarily as a web-based technology, it has a number of complexities (e.g. CSS styling, HTML DOM integration, and its declarative, retained mode design) which go beyond the simple drawing API that is desired here. Additionally, there are indications that there are patents covering some aspects of the canvas API which could complicate licensing issues. [*Note:* See http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2007-March/010129.html for details of the potential patents issue. – *end note*]

Because of the existence of good C and C++ drawing APIs which provide similar functionality without the difficulties involved in creating an API from the canvas API and SVG specification, this alternative was not fully explored. There may be functionality in the SVG specification which does not exist in cairo that would be a desirable addition. This would be a good area for further exploration in future revisions.

4. Implementer Decisions

Implementers will need to decide which underlying technology or technologies they will use to implement the API. There may be a strong temptation to use cairo itself, and this may be the correct decision for some implementations. But even in these cases working directly with the underlying rendering technologies that cairo uses is likely to produce better performance. Nonetheless, since the API only uses functionality that cairo provides, the use of cairo is feasible and could be the right choice for certain platforms.

B. Native Handles

1. Reasoning

The use of native handles allows for access to platform-specific resources in order to use functionality which was not encapsulated in the standard. The concept has been used successfully in the C++ Standard Library's threading library. It avoids the need to standardize the specifics of any particular platform's implementation of the underlying concepts. It also allows for any future platforms to expose their own non-standard functionality in a standards-compliant way without requiring any revision of the standard.

2. Alternative: Codify Particular Platforms

An alternative design would be to standardize the existing functionality of each of the platforms on which C++ is used. This design suffers severe drawbacks. Even excluding freestanding implementations, there are a large number of existing platforms, all of which have multiple versions and many of which have variants that run on different CPU architectures. The effect of codifying existing functionality would be to take a snapshot in time of all of these platforms. Platforms would need to remain conformant to that snapshot unless and until the committee modified the snapshot or else break compatibility with the standard.

This system would be extremely brittle as a result. It would potentially involve the committee in making subjective decisions as to which platforms should be included. It would also make it impossible for new platforms (or even new versions of existing platforms) to be standards compliant until such time as the standard was modified to recognize the additional platform, version, or variant.

3. Alternative: Abstract Platform Groups

Another alternative design would be to attempt to create platform groups, slotting each individual platform into an artificial grouping based upon which functionality it supports. This system is not as brittle as codifying individual platforms at first glance, but it nonetheless suffers brittleness and creates dependencies on specific (groups of) platforms, which the Standard normally avoids.

To understand this alternative, it is important to know that modern consumer GPUs have been optimized to render 3D triangles extremely fast. As a result, modern 2D graphics rendering is typically a façade that hides the use of 3D triangles that form geometries representing the 2D shapes. These are assigned a color either via texture mapping or by the use of procedurally-generated colors (e.g. a solid color, where the same value is returned regardless of the inputs, or a linear gradient, where the value returned is interpolated from two values dependent on the inputs). The end result is that modern 2D graphics are really just a specialized form of 3D graphics, one which uses an orthoscopic projection matrix to create the appearance of being 2D.

Since current commoditized GPUs normally support Microsoft's Direct3D API or else the Khronos Group's OpenGL API or its OpenGL ES API (frequently all three, with the decision of which is used dependent upon the platform which is making use of the hardware), it would seem feasible divide the world up into one group for each of those three specifications. However these specifications evolve and change over time in API breaking ways (viz. Direct3D 9 vs. Direct3D 10) and new specifications appear (e.g. AMD's Mantle API) while older ones can fall out of use (e.g. 3dfx's Glide API).

While it is unlikely that any changes would occur which would require an implementer to completely break compatibility before the committee had a chance to modify the standard, it would still place a burden to act upon the committee, one which native handles do not. It also fails to account for specialty hardware or for the use of existing hardware with a freestanding implementation which may choose to implement only a subset of the functionality of one of those specifications or which may choose to create its own custom hardware interface in order to comply with engineering constraints.

The native handle concept does not have these drawbacks and provides a result that is at least as good since the only use for publicly exposing a native handle is to expose functionality that is, by definition, platform-dependent.

4. Consequences

The use of a native handle provides a clear separation of portable and non-portable code. Its use by an implementation is perfectly fine and even expected in order for the implementation to

provide access to a native resource. There is no user expectation that use of a native handle will be portable to any platform other than the set of platforms that the implementer contemplated.

In an ideal world, the library would be able to provide all functionality that users require such that they would never need a mechanism such as a native handle. Unfortunately it is impractical to attempt to contemplate every possible requirement that users have and it is possible that some platforms may have desirable functionality (either now or in the future) that cannot be reasonably simulated on other platforms.

Providing users with access to a native handle such that they can access otherwise unexposed functionality gives users the opportunity to write code that is significantly more portable than wholly platform-dependent code would be. This is a positive achievement in terms of programmer productivity and the potential value of the user's labors.

5. Implementer Decisions

The choice of type for each native handle type is left to the implementer. Implementers should note that a native handle need not be a one-to-one mapping to an underlying type; it could be a `struct` that provides access to several native objects, for example, or a `struct` containing an `enum` type identifier and a `union` of the different types if several different possible types would make sense. Further, implementers need not provide any native handles. If they feel that their users will have no need of them, they can choose not to provide them; this proposal simply allows for their existence and creates a standard typedef name for the type, *`some_class`*`::native_handle_type`, and a signature for the member function to retrieve it, *`some_class`*`::native_handle_type` *`some_class`*`::native_handle()`.

C. Merger of the `context` type's functionality into the `surface` type

1. Reasoning

The `surface` type exists as a representation of an object that will be drawn to (a "render target" in graphics parlance). This object might be tied to a display surface, it might be block of memory with no connection to a display system, or it might even be a file or some sort of output port. What defines it from the user's perspective is that drawing operations are performed upon it.

The `context` type in N3888 had no way of setting a different, existing `surface` as its target once it was created, nor could any way of doing so easily be added since cairo itself has no such functionality. [*Note*: The `push_group` / `pop_group` functionality might seem to accomplish this but `push_group` does not take a `surface` object as a parameter because cairo does not support that. As such, `push_group` cannot be modified to take an existing `surface` without abandoning cairo. That would be ill-advised since a major benefit of leveraging cairo is using it as a measure of what operations can be done in a portable way. – *end note*]

To analogize to the real world, the `surface` is like a sheet of paper, a `pattern` is like a rubber stamp, a `pattern` used for masking is like a stencil, a stroke operation is like tracing, etc. The `context` in this analogy is… the person's hands? That's a reasonable fit, but a person can switch

to another piece of paper and start drawing there; our `context` object could not do that nor could it be modified to do it without going outside the bounds of operations that cairo supports.

So the `context` type was just a collection of functionality designed to operate on a specific `surface`, to store the necessary state to do so, and in some cases to create that state. It makes more sense to move those drawing operations and that state data to the `surface` type itself, since it will ultimately need to be applied there anyway. [*Note*: Microsoft's Direct2D uses this interface, though recent revision have added an interface that mirrors the Direct3D interface, presumably to make it more familiar to Direct3D users. – *end note*] The state data that a `context` created, objects of type `path`, are now immutable objects that are generated by a factory class (see below).

For these reasons, along with the negatives of the following contemplated alternatives, the `context` type was eliminated and its functionality was merged into the `surface` type.

2. Alternative: Retain Separate `context` and `surface` Types

The initial design adopted cairo's separate context and surface model. It did so because it was a mechanical transformation of the cairo API into a C++ API. We now have a chance to examine this interface decision *de novo* and see if it makes sense.

Historically, graphics APIs have adopted the model of using a drawing context. In early graphical computing, this was undoubtedly a memory saving mechanism since it allowed for sharing and reuse of resources by the graphical subsystem without user involvement and without significant backend overhead. But we should ask ourselves whether having a separate context is worthwhile for modern computing devices.

As mentioned above, in cairo a context can only ever draw to one non-memory surface. Further, it can only draw to memory surfaces through the push_group / pop_group interface. This interface does not allow the user to provide an existing memory surface but instead creates one on its own and delivers it to the user (or renders it onto the underlying surface) when the pop_group functionality is invoked. As such it is a one-way mechanism. Each surface must have its own context if you wish to draw to it.

Given that, the context is effectively bound to one surface for its entire life. It is concerned only with that surface and exists only to serve that surface. Having it as a separate type, when in reality it exists only to operate on a specific surface, seems to be an overzealous decoupling of functionality. Drawing functionality operates on a surface and the state data for those operations must be associated with the surface when the drawing operation occurs. As such, the functionality and state data more properly belong with the surface, rather than with a seemingly free, but in reality bound, context object.

Using cairo as a guide for cross-platform capabilities makes sense, but we are not obligated to retain its structure where the structure is orthogonal to portability (as it is here). A surface object exists primarily to be drawn on and a context object existed exclusively to draw on a specific surface. Since we are only standardizing the interface (and not the implementation), it seems

more logical to eliminate the context object and divide up its functionality in ways that better reflect what that functionality is doing.

3. Alternative: Have `context` be a Nested Child of `surface`

Rather than eliminating the `context` type, it could have been made a nested child object of its surface. This would allow for the separation of surface functionality and drawing functionality.

It turns out, though, that there really isn't much in the way of `surface`-specific functionality. Most of the limited number of `surface` functions concern themselves with ways of accessing and manipulating a surface's data outside of the normal drawing functionality. There are a few functions that deal with sharing the surface with other APIs, a couple to allow for data manipulation and recording (e.g. saving the surface's contents to an image file), and a few to provide the user with the (minimal) information that a surface discloses about itself.

Because the `surface` type has very little of its own functionality and because drawing functionality is so intimately bound with the surface it operates on, it seems reasonable to merge the two rather than create a separate child object type that must then be accessed in some way (probably via an accessor function that returns a shared_ptr) so as to prevent hidden aliasing and other insidious problems that come from separating an object's functionality (i.e. drawing to a surface) from the object itself.

4. Consequences

The surface type is now the central type in the library. In the event of a successful merger of an input mechanism proposal, the surface type would likely require additional modifications to accommodate user input. This should not present a problem, but it may be worth revisiting the composition of the surface type at that time if it seems that the type might become unwieldy as a result of the new functionality.

5. Implementer Decisions

There are no implementer decisions related to this design decision that go beyond the ordinary scope of implementing any API. For rendering technologies that do use a separate context object, implementers should not face any insoluble problems related to having the API present functionality from the context and the render target as a single interface.

D. Use of move-only semantics and immutability

1. Reasoning

Background: GPU Resources and Usage Patterns

The following cairo types use internal reference counting managed via explicit _reference/_destroy/_get_reference_count APIs:

- cairo_t
- cairo_font_face_t

- cairo_scaled_font_t
- cairo_device_t
- cairo_surface_t
- cairo_pattern_t
- cairo_region_t [*Note*: Not retained in the proposed API as it was unused. – *end note*]

What these types have in common is that they (typically) own GPU resources.

A key point is that **GPU resources are expensive to copy**.

- Moving data between the CPU and GPU (especially from the GPU to the CPU) is often expensive in terms of processing time due to synchronization requirements. Modern GPUs are highly parallel devices; a CPU can stall for quite a bit of time waiting for a GPU to respond to a request (especially if the request is to read data that the GPU is actively using). Additionally, GPU resources such as surfaces, patterns, and fonts typically take up large amounts of memory (a 32-bit RGBA bitmap image that is 1920px x 1080px takes up over 8000 KB) such that even when the data transfer operation commences, many more cycles must be spent moving all of that data.
- Typical usage patterns of GPUs for graphics applications involve the creation of and loading of GPU resources followed by their use and manipulation by the GPU with only minimal direction from the CPU. Indeed, many resources are never manipulated such that making them immutable is practical (which then allows for optimizations that otherwise would be impossible).
- GPU hardware and drivers are optimized to move data from the CPU to certain types of GPU buffer objects on a regular basis. This facilitates useful computation by the GPU and the rendering of dynamic graphical scenes. If too much data is moving from the CPU to the GPU, though, the GPU can wind up stalling while waiting for data it needs to complete computation or rendering operations that it has been instructed to perform. If data needs to move in the other direction, stalls are much more likely since moving data from the GPU to the CPU is far less likely to be an optimized transaction.

For these reasons, it is uncommon for GPU resources to be deep copied. Applications typically reuse the same resources again and again for improved performance and minimization of memory consumption and rarely manipulate GPU resources except for render targets, which is usually a heavily optimized use case.

GPU Resource Management in Various Technologies

Many existing APIs use reference counting to manage the lifetime of GPU resources.

- In Microsoft's DirectX graphics, Microsoft's COM technology is used to provide lifetime management of GPU resources. Normal usage involves reusing the same COM objects and increasing and decreasing the reference count of those COM objects as interfaces to them are obtained and released. There are special interface methods for creating deep copies of these resources (i.e. copies that result in two separate, identical objects on the GPU). Usage of these methods is not common in graphics rendering.

- In the Khronos Group's OpenGL graphics technology, GPU resources are identified by "names", which are integral values that are obtained from the OpenGL context and bound to resources of the appropriate type. Normal usage involves passing an OpenGL name around to various parts of the application which need to use the resource associated with the name. Applications must track usage carefully and only instruct the OpenGL context to delete the name when all parts of the application are finished with it (otherwise unexpected behavior will result). OpenGL contains functions that allow the user to create GPU resources and other functions that will copy data from other GPU resources to them.

Cairo supports a wide variety of back-ends. Some are hardware accelerated while others are entirely software-based. As previously mentioned, cairo uses explicit reference counting for resources that would typically be owned by the GPU in a hardware accelerated back-end. Cairo provides functions for accessing some GPU data, such as surfaces, and other functions for creating resources from that data, thus providing the ability to create deep copies of certain types of GPU resources. Other resources such as the cairo context and patterns have no such support. The user would need to create a new resource of the same type and proceed to set the appropriate state for the resource until it matched the original resource.

The Problem

Because cairo is a C API which uses explicit reference counting and pass-by-pointer parameters, its users do not have direct access to its objects. Indeed, because C does not have class types, the question of what type of semantics cairo's objects should have is inapplicable.

The cairo aggregate types have value semantics, so preserving that is straightforward. But for non-aggregate types, *what is the correct/best way to represent an explicitly reference counted C type in C++?*

The decision to remove the explicit and manual _reference/_destroy reference counting functions is not expected to be contentious (and did not prove to be so in the discussions of N3888).

N3888 contemplated an expanded role of shared ownership semantics. The general consensus (even, ultimately, among the authors) was that shared ownership semantics was unneeded.

In this proposal, most of the types that own GPU resources are immutable; those which must be mutable have move-only semantics. The reasons why are discussed below.

2. Alternative: Value Semantics

One alternative is to make these types be copyable value types. This is considered undesirable for performance reasons.

Value semantics are simple to work with:

```
X x1;
auto x2 = x1;
```

For types that own a GPU resource, however, value semantics would require making copies of the GPU resource, which is expensive in both time (stalling) and space (easily several MB or more per copy).

As discussed above, the great detriment of value semantics with graphics programming is the high cost of copying GPU resources both in terms of memory and in CPU and GPU cycles. It is not per se fatal, but it requires very careful design to avoid needless copying. A design goal for this library is that it be simple and useful, something that can enchant beginners in introductory courses and encourage them to continue studying C++ and computer programming (while also being useful for existing C++ programmers without requiring them to become expert in the inner workings of graphics hardware). As such, a design that would require expert knowledge to obtain efficient use of computing resources should be avoided when there is a simpler design that frees users from the need to have expert level knowledge.

3. Immutability and Move-Only Types

[*Note*: This is a continuation of the Reasoning section. – *end note*]

Further thought was given to this problem and a solution arose. Many of the types that own GPU resources never need to be changed once they are created. As such, it was relatively simple to make these types immutable. Once they became immutable, their inner workings were covered by the "as-if" rule such that they can have value semantics without the requirement that a deep copy to be created.

Users may wish (or need) to construct these immutable objects in multiple steps. They may also find it convenient to be able to construct an object that is similar to one they have already created. This was the motive behind the `*_builder` types. These types are mutable and copyable. They are not meant to own GPU resources and are not capable of being used in the rendering pipeline. They are factory classes that generate the immutable objects that are intended to own GPU resources. They do so based on their state at the time the immutable object is created and any subsequent state change to the builder object will not reflect back into the previously created immutable objects (if any). This approach was taken with both the `path` and `pattern` types, reflecting the likelihood that the former would probably be the owner of vertex and index buffer resources while the latter would likely be the owner of texture resources. The text rendering types that are likely to own GPU resources (e.g. font_face, font_options, scaled_font) are also immutable.

Because of how they are used, certain GPU resource owning types cannot be immutable. Specifically surfaces must be mutable in order to accomplish any drawing. It also helps to have image surfaces be mutable so that they can serve as render targets. As such, the `surface` type and the `image_surface` type are move-only types. This makes it impossible to unintentionally create a deep copy of one of these resources while keeping open options such as putting one of these objects into a shared_ptr if that fits the user's needs. The `context` type from N3888 was merged into the `surface` type, but would have also been a mutable, move-only type if it had stayed separate. The `device` type is also a move-only type, reflecting its role as a high-level interface to the graphics subsystem.

4. Consequences

There currently is no direct way of building a `pattern` that draws using the contents of a `surface`. This has been mitigated with the addition of drawing methods that take a const reference to a `surface` and execute the drawing operation while preserving drawing state (i.e. retaining and restoring any existing pattern that was in use). This should bypass the possibility that a user might accidentally have a surface bound as a source while the surface is the target of another rendering operation; a surface is move-only and can only be bound as a source for the duration of one of these synchronous drawing functions.

5. Implementer Decisions

Implementers will need to decide what rendering technologies to use. They will need to decide whether they will make use of hardware acceleration and whether to provide software fallbacks in the event that there is no suitable GPU. They will also need to decide how to handle an attempt to modify a resource that is in use as a source object in a rendering operation. The library API tries to avoid these circumstances, but a sufficiently determined user will likely be able to construct a situation in which this sort of illegal instruction could occur. Given differences between current rendering technologies and likely differences in future technologies, it seemed best to make such behavior implementation-defined rather than demanding any specific result.

VI. Mechanical Transformation Rules

The original mechanical transformation rules are documented in N3888. They were used to transform the cairo graphics library's C API into a C++ API. This met the design objective of starting from "an existing successful library already in wide use" as set forth in N3791 – Lightweight Drawing Library. The rules may be updated at some point to reflect changes made to the proposed API, but the authors feel that the rules have substantially served their purpose such that updates to them are a lower priority than developing and refining the proposal.

See the cairo API manual for version 1.12.16 (stable link: https://web.archive.org/web/20130831050514/http://cairographics.org/manual/) for the semantics of the library's functionality. Note that differences now exist between the behavior of cairo 1.12.16 and the behavior of the proposal (to whit, the function `font_options surface::get_font_options()` returns a `font_options` object that represents the actual font options that would be used to render text to the surface given the current state of the surface, which differs from the results given by `cairo_get_font_options` and by `cairo_surface_get_font_options`). The authors recognize the importance of properly documenting the proposed library's semantics and consider it a very high priority.

Note that the following items are incorporated from the cairo 1.12.16 API (as documented in the cairo API manual link above):

- All items under the Drawing topic (e.g. cairo_t, Paths, cairo_pattern_t, etc.);
- cairo_font_face_t and cairo_scaled_font_t from the Fonts topic;
- cairo_device_t, cairo_surface_t, Image Surfaces, PNG Support from the Surfaces topic; and

- All items under the Utilities topic.

VII. Technical Specifications

The semantics of each type and function below are as specified in The cairo API Documentation - Version 1.12.16 for the pre-transformed original. Note that `font_options surface::get_font_options()` has the semantics documented in the Mechanical Transformation Rules section above rather than those found in the cairo documentation. [*Note*: It is the authors' intention to provide proper API documentation with the next major revision of this proposal. – *end note*]

A. <drawing> header synopsis

```
namespace std {
  namespace experimental {
    namespace drawing {
      inline namespace v1 {
        enum class status {
          success,
          no_memory,
          invalid_restore,
          invalid_pop_group,
          no_current_point,
          invalid_matrix,
          invalid_status,
          null_pointer,
          invalid_string,
          invalid_path_data,
          read_error,
          write_error,
          surface_finished,
          surface_type_mismatch,
          pattern_type_mismatch,
          invalid_content,
          invalid_format,
          invalid_visual,
          file_not_found,
          invalid_dash,
          invalid_dsc_comment,
          invalid_index,
          clip_not_representable,
          temp_file_error,
          invalid_stride,
          font_type_mismatch,
          user_font_immutable,
          user_font_error,
          negative_count,
          invalid_clusters,
          invalid_slant,
          invalid_weight,
          invalid_size,
          user_font_not_implemented,
```

```
  device_type_mismatch,
  device_error,
  invalid_mesh_construction,
  device_finished,
  last_status
};

enum class antialias {
  default_antialias,
  none,
  gray,
  subpixel,
  fast,
  good,
  best
};

enum class content {
  color,
  alpha,
  color_alpha
};

enum class fill_rule {
  winding,
  even_odd
};

enum class line_cap {
  butt,
  round,
  square
};

enum class line_join {
  miter,
  round,
  bevel
};

enum class compositing_operator {
  clear,
  source,
  over,
  in,
  out,
  atop,
  dest,
  dest_over,
  dest_in,
  dest_out,
  dest_atop,
  xor_compositing_operator,
  add,
```

```
  saturate,
  multiply,
  screen,
  overlay,
  darken,
  lighten,
  color_dodge,
  color_burn,
  hard_light,
  soft_light,
  difference,
  exclusion,
  hsl_hue,
  hsl_saturation,
  hsl_color,
  hsl_luminosity
};

enum class format {
  invalid,
  argb32,
  rgb24,
  a8,
  a1,
  rgb16_565,
  rgb30
};

enum class path_data_type {
  move_to,
  line_to,
  curve_to,
  arc,
  arc_negative,
  new_sub_path,
  close_path
};

enum class extend {
  none,
  repeat,
  reflect,
  pad,
  default_extend = none
};

enum class filter {
  fast,
  good,
  best,
  nearest,
  bilinear,
  gaussian,
  default_filter = good
```

```
};

enum class pattern_type {
  solid_color,
  surface,
  linear,
  radial,
  mesh,
  raster_source
};

enum class font_slant {
  normal,
  italic,
  oblique
};

enum class font_weight {
  normal,
  bold
};

enum class subpixel_order {
  default_subpixel_order,
  rgb,
  bgr,
  vrgb,
  vbgr
};

enum class hint_style {
  default_hint_style,
  none,
  slight,
  medium,
  full
};

enum class hint_metrics {
  default_hint_metrics,
  off,
  on
};

namespace text_cluster_flags {
  enum text_cluster_flags : int {
    none = 0x0,
    backward = 0x1
  };
};

struct rectangle {
  double x;
  double y;
```

```
  double width;
  double height;
};

struct rgba_color {
  double r;
  double g;
  double b;
  double a;
};

struct point {
  double x;
  double y;

  point operator+=(const point& rhs);
  point operator+=(double rhs);
  point operator-=(const point& rhs);
  point operator-=(double rhs);
  point operator*=(const point& rhs);
  point operator*=(double rhs);
  point operator/=(const point& rhs);
  point operator/=(double rhs);
};

point operator+(const point& lhs);
point operator+(const point& lhs, const point& rhs);
point operator+(const point& lhs, double rhs);
point operator-(const point& lhs);
point operator-(const point& lhs, const point& rhs);
point operator-(const point& lhs, double rhs);
point operator*(const point& lhs, const point& rhs);
point operator*(const point& lhs, double rhs);
point operator/(const point& lhs, const point& rhs);
point operator/(const point& lhs, double rhs);

struct glyph {
  unsigned long index;
  double x;
  double y;
};

struct text_cluster {
  int num_bytes;
  int num_glyphs;
};

struct font_extents {
  double ascent;
  double descent;
  double height;
  double max_x_advance;
  double max_y_advance;
};
```

```
struct text_extents {
  double x_bearing;
  double y_bearing;
  double width;
  double height;
  double x_advance;
  double y_advance;
};

struct matrix {
  double xx;
  double yx;
  double xy;
  double yy;
  double x0;
  double y0;

  static matrix init_identity();
  static matrix init_translate(const point& value);
  static matrix init_scale(const point& value);
  static matrix init_rotate(double radians);

  void translate(const point& value);
  void scale(const point& value);
  void rotate(double radians);
  void invert();
  void transform_distance(point& dist);
  void transform_point(point& pt);

  matrix operator*=(const matrix& rhs);
};

matrix operator*(const matrix& lhs, const matrix& rhs);

union path_data
{
  struct {
    path_data_type type;
    int length;
  } header;
  point point;
  double value;
};

class path {
public:
  typedef implementation-defined native_handle_type; // Exposition

  path() = delete;
  path(const path_builder& pb);
  path(const path& other);
  path& operator=(const path& other);
  path(path&& other);
```

```
  path& operator=(path&& other);

  ::std::vector<path_data> get_data() const;
  const ::std::vector<path_data>& get_data_ref() const;
  void get_path_extents(point& pt0, point& pt1) const;
};

class path_builder {
public:
  path_builder();
  path_builder(const path_builder& other);
  path_builder& operator=(const path_builder& other);
  path_builder(path_builder&& other);
  path_builder& operator=(path_builder&& other);

  path get_path() const;
  path get_path_flat() const;

  void append_path(const path& p);
  void append_path(const path_builder& p);
  bool has_current_point();
  point get_current_point();
  void new_sub_path();
  void close_path();
  void arc(const point& center, double radius, double angle1,
    double angle2);
  void arc_negative(const point& center, double radius,
    double angle1, double angle2);
  void curve_to(const point& pt0, const point& pt1,
    const point& pt2);
  void line_to(const point& pt);
  void move_to(const point& pt);
  void rectangle(const rectangle& rect);
  void rel_curve_to(const point& dpt0, const point& dpt1,
    const point& dpt2);
  void rel_line_to(const point& dpt);
  void rel_move_to(const point& dpt);

  ::std::vector<path_data> get_data() const;
  const ::std::vector<path_data>& get_data_ref() const;
  ::std::vector<path_data>& get_data_ref();
  void get_path_extents(point& pt0, point& pt1) const;
};

class drawing_exception : public exception {
public:
  drawing_exception() noexcept;
  explicit drawing_exception(status s) noexcept;

  virtual ~drawing_exception() noexcept;

  drawing_exception(const drawing_exception& rhs) noexcept;
  drawing_exception& operator=(
    const drawing_exception& rhs) noexcept;
```

```
  virtual const char* what() const noexcept;
  status get_status() const noexcept;
};

class device {
public:
  typedef implementation-defined native_handle_type; // Exposition
  native_handle_type native_handle() const; // Exposition

  device() = delete;
  device(const device&) = delete;
  device& operator=(const device&) = delete;
  device(device&& other);
  device& operator=(device&& other);
  explicit device(native_handle_type nh); // Exposition
  void flush();
  void acquire();
  void release();
};

// Forward declaration.
class font_options;

class font_options_builder {
public:
  font_options_builder();
  font_options_builder(const font_options_builder&);
  font_options_builder& operator=(const font_options_builder&);
  font_options_builder(font_options_builder&& other);
  font_options_builder& operator=(font_options_builder&& other);

  font_options get_font_options() const;
  void set_antialias(antialias a);
  antialias get_antialias() const;
  void set_subpixel_order(subpixel_order so);
  subpixel_order get_subpixel_order() const;
  void set_hint_style(hint_style hs);
  hint_style get_hint_style() const;
  void set_hint_metrics(hint_metrics hm);
  hint_metrics get_hint_metrics() const;
};

class font_options {
public:
  typedef implementation-defined native_handle_type; // Exposition
  native_handle_type native_handle() const; // Exposition

  font_options(const font_options&);
  font_options& operator=(const font_options&);
  font_options(font_options&& other);
  font_options& operator=(font_options&& other);
  font_options(antialias a, subpixel_order so, hint_style hs,
    hint_metrics hm);
```

```
  explicit font_options(native_handle_type nh); // Exposition

  antialias get_antialias() const;
  subpixel_order get_subpixel_order() const;
  hint_style get_hint_style() const;
  hint_metrics get_hint_metrics() const;
};

class font_face {
public:
  typedef implementation-defined native_handle_type; // Exposition
  native_handle_type native_handle() const; // Exposition

  font_face() = delete;
  font_face(const font_face&);
  font_face& operator=(const font_face&);
  font_face(font_face&& other);
  font_face& operator=(font_face&& other);

  explicit font_face(native_handle_type nh); // Exposition

  virtual ~font_face();
};

class scaled_font {
public:
  typedef implementation-defined native_handle_type; // Exposition
  native_handle_type native_handle() const; // Exposition

  scaled_font() = delete;
  scaled_font(const scaled_font&);
  scaled_font& operator=(const scaled_font&);
  scaled_font(scaled_font&& other);
  scaled_font& operator=(scaled_font&& other);

  explicit scaled_font(native_handle_type nh); // Exposition
  scaled_font(const font_face& ff, const matrix& fm,
    const matrix& ctm, const font_options& fo);

  font_extents get_extents() const;
  text_extents get_text_extents(const ::std::string& utf8) const;
  text_extents get_glyph_extents(
    const ::std::vector<glyph>& glyphs) const;
  ::std::vector<glyph> text_to_glyphs(double x, double y,
    const ::std::string& utf8) const;
  ::std::vector<glyph> text_to_glyphs(double x, double y,
    const ::std::string& utf8,
    ::std::vector<text_cluster>& clusters,
    text_cluster_flags::text_cluster_flags& clFlags) const;
};

class toy_font_face : public font_face {
public:
  toy_font_face() = delete;
```

```
  toy_font_face(const toy_font_face&);
  toy_font_face& operator=(const toy_font_face&);
  toy_font_face(const ::std::string& family, font_slant slant,
    font_weight weight);
  toy_font_face(toy_font_face&& other);
  toy_font_face& operator=(toy_font_face&& other);

  ::std::string get_family() const;
  font_slant get_slant() const;
  font_weight get_weight() const;
};

class pattern {
public:
  typedef implementation-defined native_handle_type; // Exposition
  native_handle_type native_handle() const; // Exposition

  pattern() = delete;
  pattern(const pattern&);
  pattern& operator=(const pattern&);
  pattern(pattern&& other);
  pattern& operator=(pattern&& other);

  ~pattern();

  pattern_type get_type() const;
};

class solid_color_pattern_builder {
  pattern_type _Pattern_type;
  extend _Extend;
  filter _Filter;
  matrix _Matrix;
  rgba_color _Color;

public:
  solid_color_pattern_builder(const solid_color_pattern_builder&);
  solid_color_pattern_builder& operator=(
    const solid_color_pattern_builder&);
  solid_color_pattern_builder(
    solid_color_pattern_builder&& other);
  solid_color_pattern_builder& operator=(
    solid_color_pattern_builder&& other);
  solid_color_pattern_builder(const rgba_color& color);

  pattern get_pattern();
  void set_extend(extend e);
  extend get_extend();
  void set_filter(filter f);
  filter get_filter();
  void set_matrix(const matrix& m);
  matrix get_matrix();

  rgba_color get_rgba();
```

```
  void set_rgba(const rgba_color& color);
  double get_red();
  void set_red(double red);
  double get_green();
  void set_green(double green);
  double get_blue();
  void set_blue(double blue);
  double get_alpha();
  void set_alpha(double alpha);
};

class linear_pattern_builder {
public:
  linear_pattern_builder() = delete;
  linear_pattern_builder(const linear_pattern_builder&);
  linear_pattern_builder& operator=(
    const linear_pattern_builder&);
  linear_pattern_builder(linear_pattern_builder&& other);
  linear_pattern_builder& operator=(
    linear_pattern_builder&& other);
  linear_pattern_builder(const point& pt0, const point& pt1);

  pattern get_pattern();
  void set_extend(extend extend);
  extend get_extend();
  void set_filter(filter filter);
  filter get_filter();
  void set_matrix(const matrix& matrix);
  matrix get_matrix();

  void add_color_stop_rgba(double offset,
    const rgba_color& color);
  int get_color_stop_count();

  void get_color_stop_rgba(int index, double& offset,
    rgba_color& color);
  void set_color_stop_rgba(int index, double offset,
    const rgba_color& color);
  void get_linear_points(point& pt0, point& pt1);
  void set_linear_points(const point& pt0, const point& pt1);
};

class radial_pattern_builder {
public:
  radial_pattern_builder() = delete;
  radial_pattern_builder(const radial_pattern_builder&);
  radial_pattern_builder& operator=(
    const radial_pattern_builder&);
  radial_pattern_builder(radial_pattern_builder&& other);
  radial_pattern_builder& operator=(
    radial_pattern_builder&& other);
  radial_pattern_builder(const point& center0, double radius0,
    const point& center1, double radius1);
```

```
  pattern get_pattern();
  void set_extend(extend extend);
  extend get_extend();
  void set_filter(filter filter);
  filter get_filter();
  void set_matrix(const matrix& matrix);
  matrix get_matrix();

  void add_color_stop_rgba(double offset,
    const rgba_color& color);
  int get_color_stop_count();

  void get_color_stop_rgba(int index, double& offset,
    rgba_color& color);
  void set_color_stop_rgba(int index, double offset,
    const rgba_color& color);

  void get_radial_circles(point& center0, double& radius0,
    point& center1, double& radius1);
  void set_radial_circles(const point& center0, double radius0,
    const point& center1, double radius1);
};

class mesh_pattern_builder {
public:
  mesh_pattern_builder();
  mesh_pattern_builder(const mesh_pattern_builder&);
  mesh_pattern_builder& operator=(const mesh_pattern_builder&);
  mesh_pattern_builder(mesh_pattern_builder&& other);
  mesh_pattern_builder& operator=(mesh_pattern_builder&& other);

  pattern get_pattern();
  void set_extend(extend extend);
  extend get_extend();
  void set_filter(filter filter);
  filter get_filter();
  void set_matrix(const matrix& matrix);
  matrix get_matrix();

  void begin_patch();
  void begin_edit_patch(unsigned int patch_num);
  void end_patch();
  void move_to(const point& pt);
  void line_to(const point& pt);
  void curve_to(const point& pt0, const point& pt1,
    const point& pt2);
  void set_control_point(unsigned int point_num, const point& pt);
  void set_corner_color_rgba(unsigned int corner_num,
    const rgba_color& color);
  void get_patch_count(unsigned int& count);
  path get_path(unsigned int patch_num);
  path_builder get_path_builder(unsigned int patch_num);
  point get_control_point(unsigned int patch_num,
    unsigned int point_num);
```

```
  rgba_color get_corner_color_rgba(unsigned int patch_num,
    unsigned int corner_num);
};

class raster_source_pattern_builder {
public:
  raster_source_pattern_builder() = delete;
  raster_source_pattern_builder(
    const raster_source_pattern_builder&);
  raster_source_pattern_builder& operator=(
    const raster_source_pattern_builder&);
  raster_source_pattern_builder(
    raster_source_pattern_builder&& other);
  raster_source_pattern_builder& operator=(
    raster_source_pattern_builder&& other);
  raster_source_pattern_builder(void* user_data, content content,
    int width, int height);

  pattern get_pattern();
  void set_extend(extend extend);
  extend get_extend();
  void set_filter(filter filter);
  filter get_filter();
  void set_matrix(const matrix& matrix);
  matrix get_matrix();

  void set_callback_data(void* data);
  void* get_callback_data();
  void set_acquire(
    ::std::function<surface(void* callback_data, surface& target,
      const rectangle& extents)> acquire_fn,
    ::std::function<void(void* callback_data,
      surface& surface)> release_fn
    );
  void get_acquire(
    ::std::function<surface(void* callback_data, surface& target,
      const rectangle& extents)>& acquire_fn,
    ::std::function<void(void* callback_data,
      surface& surface)>& release_fn
    );
};

class surface {
public:
  surface() = delete;

  typedef implementation-defined native_handle_type; // Exposition
  native_handle_type native_handle() const; // Exposition

  surface(const surface&) = delete;
  surface& operator=(const surface&) = delete;

  surface(surface&& other);
  surface& operator=(surface&& other);
```

```
            explicit surface(native_handle_type nh); // Exposition

            surface(format fmt, int width, int height);
            // create_similar
            surface(const surface& other, content content, int width, int
height);
            // create_for_rectangle
            surface(const surface& target, const rectangle& rect);

            virtual ~surface();

            void finish();
            void flush();

            ::std::shared_ptr<device> get_device();

            content get_content();
            void mark_dirty();
            void mark_dirty_rectangle(const rectangle& rect);

            void set_device_offset(const point& offset);
            void get_device_offset(point& offset);
            void write_to_png(const ::std::string& filename);
            image_surface map_to_image(const rectangle& extents);
            void unmap_image(image_surface& image);
            bool has_surface_resource() const;

            void save();
            void restore();
            void push_group();
            void push_group_with_content(content c);
            surface pop_group();
            void pop_group_to_source();

            void set_pattern();
            void set_pattern(const pattern& source);
            pattern get_pattern();

            void set_antialias(antialias a);
            antialias get_antialias();

            void set_dash();
            void set_dash(const ::std::vector<double>& dashes,
              double offset);
            int get_dash_count();
            void get_dash(::std::vector<double>& dashes, double& offset);

            void set_fill_rule(fill_rule fr);
            fill_rule get_fill_rule();

            void set_line_cap(line_cap lc);
            line_cap get_line_cap();
```

```
void set_line_join(line_join lj);
line_join get_line_join();

void set_line_width(double width);
double get_line_width();

void set_miter_limit(double limit);
double get_miter_limit();

void set_compositing_operator(compositing_operator co);
compositing_operator get_compositing_operator();

void set_tolerance(double tolerance);
double get_tolerance();

void clip();
void clip_extents(point& pt0, point& pt1);
bool in_clip(const point& pt);
void reset_clip();

::std::vector<rectangle> copy_clip_rectangle_list();

void fill();
void fill(const surface& s);
void fill_extents(point& pt0, point& pt1);
bool in_fill(const point& pt);

void mask(const pattern& pttn);
void mask(const surface& surface);
void mask(const surface& surface, const point& origin);

void paint();
void paint(const surface& s);
void paint_with_alpha(double alpha);
void paint_with_alpha(const surface& s, double alpha);

void stroke();
void stroke(const surface& s);
void stroke_extents(point& pt0, point& pt1);
bool in_stroke(const point& pt);

void set_path();
void set_path(const path& p);

// Transformations
void translate(const point& value);
void scale(const point& value);
void rotate(double angle);
void transform(const matrix& matrix);
void set_matrix(const matrix& matrix);
void get_matrix(matrix& matrix);
void identity_matrix();
void user_to_device(point& pt);
void user_to_device_distance(point& dpt);
```

```
  void device_to_user(point& pt);
  void device_to_user_distance(point& dpt);

  // Text
  void select_font_face(const ::std::string& family,
    font_slant slant, font_weight weight);
  void set_font_size(double size);
  void set_font_matrix(const matrix& matrix);
  void get_font_matrix(matrix& matrix);
  void set_font_options(const font_options& options);
  font_options get_font_options();
  void set_font_face(font_face& font_face);
  font_face get_font_face();
  void set_scaled_font(const scaled_font& scaled_font);
  scaled_font get_scaled_font();
  void show_text(const ::std::string& utf8);
  void show_glyphs(const ::std::vector<glyph>& glyphs);
  void show_text_glyphs(const ::std::string& utf8,
    const ::std::vector<glyph>& glyphs,
    const ::std::vector<text_cluster>& clusters,
    text_cluster_flags::text_cluster_flags cluster_flags);
  void font_extents(font_extents& extents);
  void text_extents(const ::std::string& utf8,
    text_extents& extents);
  void glyph_extents(const ::std::vector<glyph>& glyphs,
    ::std::experimental::drawing::text_extents& extents);
};

class image_surface : public surface {
public:
  image_surface() = delete;
  image_surface(const image_surface&) = delete;
  image_surface& operator=(const image_surface&) = delete;
  image_surface(image_surface&& other);
  image_surface& operator=(image_surface&& other);
  image_surface(surface::native_handle_type nh,
    surface::native_handle_type map_of); // Exposition
  image_surface(format format, int width, int height);
  image_surface(vector<unsigned char>& data, format format,
    int width, int height, int stride);
  image_surface(surface& other, format format, int width,
    int height);
  image_surface(const ::std::string& filename);

  void set_data(::std::vector<unsigned char>& data);
  ::std::vector<unsigned char> get_data();
  format get_format();
  int get_width();
  int get_height();
  int get_stride();
};

int format_stride_for_width(format format, int width);
```

```
        // Parameters are exposition
        surface make_surface(surface::native_handle_type nh);
        // Parameters are exposition
        surface make_surface(format format, int width, int height);
      }
    }
  }
}
```

VIII. Acknowledgments

The authors of this paper would like to acknowledge and thank the following individuals and groups:

- The *cairo* graphics library community
- Andrew Bell
- Beman Dawes
- Gunnar Sletta
- Behdad Esfahbod

IX. Revision History

Revision 1:

- Removed path creation from the context type. The `path` type is now an immutable class type with a `path_builder` type used for creating `path` objects.
- Removed transformation rules; they are available in N3888 for reference.
- Eliminated shared ownership semantics and replaced it with a combination of immutable types and mutable, move-only types. Most immutable types have corresponding mutable factory types. The reasoning is set forth under Design Decisions.
- Noted difference in semantics for `surface::get_font_options` from cairo's `cairo_get_font_options` and `cairo_surface_get_font_options` functions.
- The `context` type has been eliminated and its functionality has been merged into the `surface` type. The reasoning is set forth under Design Decisions.
- The `pattern` type has become an immutable type with no derived types. Its former derived types are now pattern builder types without any inheritance (e.g. `solid_color_pattern` became `solid_color_pattern_builder` and the gradient pattern type was eliminated with its functionality being put directly into the `linear_pattern_builder` and `radial_pattern_builder` types).
- Eliminated `rectangle_int` type and replaced usages of it with use of the `rectangle` type; replaced `int` parameters with `double` parameters where it made since. The reasoning is set forth under Design Decisions.

- Added a `point` POD type that represents a 2D coordinate value. Replaced instances of `double x, double y` parameters (and similar combinations) with appropriate usage of the `point` type.
- Eliminated the `matrix` type's `multiply` member function and replaced it with suitable overloads of the `*` and `*=` operators.
- Eliminated the `matrix::init` function as needless duplication of existing language functionality (e.g. braced initialization). Converted the remaining `init_*` member functions into static factory member functions to make them useful when a temporary `matrix` value is needed (e.g. as an argument to a function that takes a `const matrix&` parameter).
- Added missing functionality to `scaled_font` type.
- Simplified `surface` type by removing PNG stream writing functionality and mime functionality.
- Simplified `image_surface` type by removing create from PNG stream functionality.
- Removed `user_data_key` type and eliminated user data functionality from types that had it for simplicity and because it didn't seem to serve a valuable purpose (especially in the form it was in).
- Removed `finish` and `status` member functions from the `device` type. The `finish` function did not seem to have any good use cases and eliminating it made the `status` function unnecessary.
- Removed `region_overlap` and `region` types; they were not used by any current functionality.
- Eliminated type `rectangle_list` in favor of `vector<rectangle>`.
- Converted `surface::get_device` to return a `shared_ptr<device>` object rather than a `device` object. `device` is a move-only type and thus multiple copies of the same `device` object should not be possible (but would have been without this change).
- Placed API within `inline namespace v1` to allow for proper versioning.
- Added `rgba_color` type to represent unsigned normal RGBA color values.

X. References

[N3791 – Lightweight Drawing Library]

[N3825 – SG13 Graphics Discussion Minutes, 2013-11-21]

[The cairo API Documentation - Version 1.12.16]

[N3888 - A Proposal to Add 2D Graphics Rendering and Display to C++]