Document number: N3888
Project: JTC1/SC22/WG21 Programming Language C++/SG13 - Graphics

Date: 2014-01-18

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++

I. Table of Contents

II. Introduction

The goal of this proposal is to define a 2D drawing API based on 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.

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 mainstream platform targets. For example, writing a simple `cout << "Hello, world!"` doesn't do anything useful on many tablets and smartphones.

Application programmers write programs that often need to render or display basic 2D graphics, including for introductory examples. 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.

Their current options are to 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 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), 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 or other). Though the design and implementation of the proposal require domain expertise, its design aims to be easily understood (and thus easily usable) by all levels of C++ programmers, from beginners through experts.

The scope of this proposal is limited to 2D drawing. It is expected that there will be one or more additional proposals which will add additional features such as the ability to capture and process keyboard, mouse, and touch input. When unified, the resulting proposal 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 anything currently in the Standard. It makes use of some existing library functionality.

The technical specification presented below depends on:

- function
- string
- vector
- exception
- shared_ptr

This proposal can be implemented using C++11 compilers and libraries.

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 implementable and that it is 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 mechanism of an explicit this pointer as the first argument to the relevant stand-alone functions (along with indicative naming conventions). This makes it conceptually easier to consider and approve future proposals to 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.

Taken as a whole, starting from cairo allows the creation of a 2D C++ drawing library that is already known to be cross-platform implementable, useful, and object-oriented without the need to spend years drafting, implementing, and testing the 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, and open the doors to a whole new set of potential design flaws which might be missed in the design process. 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 and the design flaws of which, if any, are known and can be corrected. If there were no suitable existing APIs or if other reasons existed that negated the possibility 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 extraneous aspects (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 potential complexities 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 on an underlying technology 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 working directly with the underlying rendering technology is likely to produce better performance and on some platforms (e.g. Windows) since cairo makes use of older rendering technologies that have been superseded by newer APIs (cairo's Win32 implementation is currently GDI-based; a Direct2D-based implementation would likely be significantly faster).

B. Native Handles

1. Reasoning

The use of native handles allows for access to platform-specific resources to use functionality which was not encapsulated in the standard (whether because it is too new for clear idioms, too

dependent on the underlying platform to admit itself to standardization, or otherwise not sufficiently ripe). 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 any revision of the standard being mandatory.

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, many of which have versions 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.

This system would be extremely brittle as a result. It would potentially involve the committee in making value 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 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.

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 to form geometries which are assigned 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).

Since commoditized GPUs normally support Microsoft's Direct3D API or the Khronos Group's OpenGL API or OpenGL ES API (frequently all three, with the decision of which is used dependent on the platform which is making use of the hardware), it would be possible to divide the world up into one group for each of those three specifications. However these specifications evolve and change over time 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 completely break compatibility before the committee had a chance to modify the standard, it would still place a burden to act upon the committee which does not exist with the use of a native handle. 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 gain access to a native resource and use implementation-dependent functionality to use it. There is no expectation that an implementation will be portable to any platform other than the set of platforms that the implementers contemplated.

In an ideal world, the library would provide all functionality that users require such that they would never need to resort to the use of 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. The native handle need not be a one-to-one mapping to an underlying type; it could be a `struct` containing 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.

C. Usage of Shared Ownership Semantics for Certain Types

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

What these types have in common is that they (typically) represent 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. 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 a request to read data that the GPU may be 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).
- Typical usage patterns of GPUs for graphics applications involve the creation of and loading of GPU resources followed by their manipulation by the GPU with only minimal direction from the CPU. 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.

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.

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.

Being C++ and proud of it, this library must address the semantics that its objects should have. The two most common types are value semantics and reference semantics. But shared ownership semantics is another option. [*Note:* see 20.8.2.2 [util.smartptr.shared]/1. – *end note*]

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.

This initial proposal explores using shared ownership semantics; feedback is welcome on whether value semantics or explicit use of smart pointers would be better.

2. Alternative: Value Semantics

One alternative is to make these types be copyable value types. This is considered undesirable for performance reasons, and unworkable for patterns which are inheritance-based reference types.

Value semantics, when workable, are simple to work with:

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

For a GPU resource, however, value semantics would make it easy to create copies, which are expensive in both time (stalling) and space (easily several MB or more per copy).

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 and will still incur some copying and likely some usage of shared_ptr or something of a similar design. 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. An early need to delve

into lvalues, shared_ptrs, and templates would remove the simplicity and probably make it difficult and frustrating for most beginners.

3. Alternative: Reference Semantics

Another alternative is to make these types be move-only types intended to be held by reference. In particular, in this proposal, patterns are already inheritance-based types that are exposed in this way.

Reference semantics avoid the performance pitfall of value semantics by making deep copying something that only happens if the programmer forces it. The cost is that users are exposed to pointers.

```
shared_ptr<X> x1 = ...;
auto x2 = x1;
```

The copying is shallow, and this is naturally visible to the user. However, the user is required to know about shared_ptrs.

The first object that needs to be created to use this library is a `surface`. The current rules in this proposal specify that there must be a function named `make_surface` that returns a `surface` object. With reference semantics, the function would instead return a `unique_ptr<surface>`. Once you have a `surface` object, you can construct a `context`.

4. Shared Ownership Semantics

Another alternative is to use handle/body, and make these types be "counted handle" types where copying is shallow and instances share representation. In this proposal we are currently exploring this option and would welcome feedback.

The advantages of this option are that it avoids the performance pitfall of silent expensive copying (because copying is always shallow) and the need for the user to use smart pointers. A potential drawback is that in code like

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

the two objects do not silently *copy*, but instead silently *alias*. For example, the user must remember that using x1 and x2 on different threads must be synchronized. This would arguably be clearer and easier to see if x1 and x2 were shared_ptrs and the explicit indirection of using *x1 and *x2 on different threads would be a reminder of the potential aliasing.

That being said, graphics code is typically not threaded. To produce fluid animations, which requires maintaining a reasonably high frame rate (typically at least 30 frames per second), graphics developers typically devote one thread to rendering with other threads used for auxiliary tasks. The graphics thread typically only synchroni mechanisms where it will not be kept

waiting. There are advanced techniques for multi-threaded graphics rendering, but they are beyond the scope of this proposal.

Summary

This proposal contemplates that there is room for further use of shared ownership semantics and allows these classes to assume shared ownership semantics. As stated earlier, the committee's insight into these complicated issues will be most welcome.

VI. Mechanical Transformation Rules

The following is a series of mechanical rules used to transform select portions of version 1.12.16 of the cairo API into the API proposed by this library. Specifically, referring to the cairo API manual for version 1.12.16 (stable link: https://web.archive.org/web/20130831050514/http://cairographics.org/manual/), the following items are incorporated:

- 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.

0. Interpretation.
   0.1. Rules which are in the form #. [*Example:* "1." – *end example*] are informative only and have no normative value.
   0.2. A child rule is any rule which is denominated by at least two numbers separated by a period. [*Example:* Rule 1. is not a child rule. Rule 1.1 is a child rule. – *end example*]
   0.3. All child rules are normative absent an explicit statement to the contrary.
   0.4. A parent rule is a rule which has at least one child rule. [*Example:* If there exists a rule 1.1, then rule 1. is a parent rule. If there is no rule 1.1 nor any other rule in the form 1.#, then rule 1. is not a parent rule. – *end example*]
   0.5. When two or more rules are applicable to a situation, in the event of any conflict a clause in a child rule overrides a clause in its parent and a clause in a later rule overrides a clause in an earlier rule absent explicit language to the contrary.
   0.6. When reference is made to a rule by the rule itself or by another rule, by default all of the referenced rule's child rules and their children, etc., are included in that reference. [*Note:* Use of explicit language such as "... rule 1.1, excluding its children, ..." shall serve to override the default interpretation. – *end note*]
      0.6.1. When reference is made to a parent rule by one of its immediate children, by default none of the parent's child rules are included in that reference. [*Example:* a reference to rule 1.1 by rule 1.1.1 shall come under this rule as 1.1.1 is an immediate child of 1.1, but a reference to rule 1.1 by rule 1.1.1.1 shall not come under this rule as 1.1.1.1 is not an immediate child of 1.1. – *end example*] [*Note:* Use of explicit

language such as "... rule 0.2, including its children, ..." shall serve to override the default interpretation. – *end note*]

0.6.1.1. When a reference to a rule in the form "#." [*Example:* "rule 1." – *end example*] is made, by default the reference shall include its child rules [*Example:* "1.1" – *end example*] and their children, etc.

1. Transformations for types and type names.
   1.1. Convert `struct` types with the naming scheme *cairo_X_t* to class type *class X* or *struct X*. Any *cairo_X_t* type which, after the application of all other rules, would be an *aggregate* if defined as a `struct` shall be defined as a `struct`.
      1.1.1. The `cairo_t` type would, by application of rule 1.1, have no resulting type name. As such, it shall instead be transformed so that it has the name `context`. [*Note:* This name was chosen because the `cairo_t` type operates as a graphics context, as that term is used in the domain of computer graphics and it was felt that `drawing::context` alone, without a graphics_ qualifier prefixing it, would be sufficient to convey its purpose within the scope of its use. – *end note*]
   1.2. Convert `enum` types with the naming scheme *cairo_X_t* to *enum class X*.
      1.2.1. Convert enum types with the naming scheme *cairo_X_flags_t* to a *bitmask* type with name *X_flags* that shall be contained within the `namespace` *::std::experimental::drawing::X_flags*.
      1.2.2. The type `enum cairo_operator_t` would, by application of rule 1.2, have a type name of `operator`, which is a reserved keyword. As such, it shall instead be transformed so that it has the name `compositing_operator`. [*Note:* This name was chosen for its descriptive value since `cairo_operator_t` is used to describe how compositing (the mixing together of source and destination colors) should be performed and the term compositing is used in the cairo documentation that describes this `enum`. – *end note*]
      1.2.3. The types `cairo_surface_type_t` and `cairo_device_type_t` shall not have corresponding types in this library. [*Note:* These types are excluded in order to avoid including platform-specific values in this library. Implementers will need to decide the most appropriate underlying type for the `surface` class and the `device` class in their implementations and can expose the underlying types using a native handle if that is desirable. – *end note*]
   1.3. Enumerators in cairo in the form *CAIRO_ENUM_NAME_VALUE* shall be transformed by removing *CAIRO_ENUM_NAME_* and transforming the remaining *VALUE* part into lower case. In circumstances where the application of this rule would result in an enumerator that is a reserved word, the enumerator shall have an underscore followed by the name of its *enum* appended to it. [*Example:* The `enum cairo_hint_metrics_t` type has `CAIRO_HINT_METRICS_OFF` as one of its enumerators. Operation of this rule gives as the resulting enumerator `off`, contained within `enum class hint_metrics`. This same `enum` has `CAIRO_HINT_METRICS_DEFAULT` as another one of its enumerators. Application of the first clause of this rule would give `default` as the resulting enumerator and since `default` is a reserved word, application of the second clause is invoked, giving as the resulting enumerator `default_hint_metrics`, contained within `enum class hint_metrics`. – *end example*]
   1.4. Convert `union` types with the naming scheme *cairo_X_t* to *union X*.

1.5. For each enumerator in `cairo_pattern_type_t`, create a `struct` or `class` that publicly derives from the pattern class created by application of rule 1.1. The name shall be formed by removing *CAIRO_PATTERN_TYPE_* from the enumerator's name, transforming the remaining part into lower case, and appending *_pattern* to it. [*Example:* `CAIRO_PATTERN_TYPE_MESH` becomes `mesh_pattern`. – *end example*]

1.5.1. Where there is at least one `cairo_pattern_XXX` or `cairo_mesh_pattern_XXX` function that accepts multiple pattern types as valid input, create a new intermediate class or struct which publicly inherits from the `pattern` class created by application of rule 1.1. Those pattern types will then publicly inherit from the new intermediate type. The name of the intermediate type shall be the name used in the cairo function's documentation to describe the pattern type, with *_pattern* appended to it. [*Note:* In the event that there is no clear name, a rule denoting the name applied to the type generated by the function shall be created and added to this list of rules. – *end note*]

1.6. The `cairo_user_data_key_t` type shall be transformed as follows:

1.6.1. It shall become a class type with the name `user_data_key`. [*Note:* The `cairo_user_data_key_t` type is a struct that contains a single integer, the value of which is disregarded. Instead, the *memory address of* the `cairo_user_data_key_t` is used as a unique value. The purpose of `user_data_key` is to provide a unique value without requiring the use of a memory address as the value. – *end note*]

1.6.2. The `user_data_key` type shall have a default constructor which stores a unique value for the `user_data_key` object. No other instance of `user_data_key` shall be assigned that value by the default constructor for the life of that object. [*Note:* The type of the stored value is implementation defined subject to the following constraints: the type must be capable of representing a number of discrete values equal to or greater than the number of discrete addresses that a native pointer can address in the target environment; and, it must be possible to compare two values of the type using the *equality operators*. – *end note*]

1.6.3. The `user_data_key` type shall overload the operators `==` and `!=` so that they compare the stored values of the `user_data_key` objects that are being compared rather than the objects themselves.

1.6.4. All other details of the `user_data_key` type shall be implementation defined.

1.6.5. [*Example:* Given an environment where the native pointer type is capable of 64-bit addressing, the following is an acceptable implementation (expositve only):

```
class user_data_key {
  static ::std::atomic<::std::uint_fast64_t> _Cnt;
  ::std::uint_fast64_t _Val;
public:
  user_data_key() : _Val(++_Cnt) {}
  ::std::uint_fast64_t _Get_value() const { return _Val; }
  bool operator<(const user_data_key& other) const {
    return _Val < other._Val;
  }
  bool operator>(const user_data_key& other) const {
    return _Val > other._Val;
  }
```

```
  bool operator==(const user_data_key& other) const {
    return _Val == other._Val;
  }
  bool operator!=(const user_data_key& other) const {
    return _Val != other._Val;
  }
};
```

– *end example*]

1.7. Create a struct or class named `image_surface`. It shall publicly inherit from the `surface` class.

1.8. Create a struct or class named `toy_font_face`. It shall publicly inherit from the `font_face` class.

1.9. The `cairo_bool_t` type shall become the C++ `bool` type.

2. Transformations for functions and member function assignments.

2.1. Convert functions with the name pattern *cairo_TYPENAME_X* to have the name *X*, where *TYPENAME* is the name of a resulting type after the application of all parts of rule 1 except rule 1.5.

2.1.1. Convert functions with the name pattern *cairo_mesh_pattern_X* to have the name *X*.

2.1.2. Application of rule 2.1 to the functions `cairo_region_xor` and `cairo_region_union` would result in functions named `xor` and `union`, which are reserved keywords. Therefore, the functions `cairo_region_xor`, `cairo_region_union`, `cairo_region_intersect`, and `cairo_region_subtract` shall be converted to have the names `xor_region`, `union_region`, `intersect_region`, and `subtract_region`, respectively. [*Note:* The functions cairo_region_intersect and cairo_region_subtract are renamed by this rule to maintain naming consistency. – *end note*]

2.2. Any function parameter that is a cairo type shall be converted to be the type it has after the application of rule 1. For the remainder of rule 0, whenever the symbol *X* is used, it shall refer to a cairo type and whenever the symbol *X′* is used, it shall refer to the type that results from the application of rule 1 to *X*.

2.3. Any function which takes a pointer to *X* as its first parameter shall become a member function of *X′*. That pointer to *X* shall be removed from the function's explicit parameters as it becomes the implicit `this` pointer.

2.3.1. Any function which takes a pointer to *X* as its first parameter but which the cairo naming conventions make clear should nonetheless not become a member function of *X′* shall not become a member function of *X′*.

2.3.2. Any function with a name containing *_create* that returns *X* shall become a constructor for *X′* and shall not become a member function of any other type. [*Example:* The function `cairo_create` shall become a constructor for the `context` type, and shall not be a member function of the `surface` type, despite taking a pointer to it as its first parameter. – *end example*] Any constructor generated by this rule which has only one parameter shall be marked `explicit`.

2.3.2.1. Any function with a name containing *_create* that returns a `cairo_pattern_t` shall become a constructor for a type that is derived from `pattern`. To determine the type that the function should become a constructor

for, take the `cairo_pattern_t` that the function returns and determine which `cairo_pattern_type_t` value would be returned by calling `cairo_pattern_get_type` on it. The correct type is the type that was generated for that value by application of rule 1.5. [*Example:* Calling `cairo_pattern_create_linear` returns a `cairo_pattern_t` for which `cairo_pattern_get_type` returns `CAIRO_PATTERN_TYPE_LINEAR`. Application of rule 1.5 for that value ultimately results in the generation of the type `linear_pattern`. As such, `cairo_pattern_create_linear` becomes a constructor for `linear_pattern`. – *end example*]

2.3.2.2. Any function with a name containing *image_surface_create* shall become a constructor for the `image_surface` class. It shall not be a constructor or member of any other class.

2.3.2.3. Any function with a name containing *toy_font_face_create* shall become a constructor for the `toy_font_face` class. It shall not be a constructor or member of any other class.

2.3.3. Any function with a name ending in *_reference* or *_destroy* shall not have corresponding functions in this library. Any class type *X′* which would have had reference or destroy member functions but for the application of this rule shall be storable in a `shared_ptr<X′>`. [*Note:* These functions are for intrusive reference counting, which is not needed in the presence of standard reference counting. – *end note*]

2.3.4. Any function which is not a member function of another type after application of rule 0, excluding this rule, and which does not have a clear `this` pointer to *X* but which creates or manipulates a class type object of type *X′* shall become a static member function of *X′*.

2.4. Any function which would become a non-constructor member function of the `pattern` class by application of rule 2.3 shall be examined to see if it can either return `CAIRO_STATUS_PATTERN_TYPE_MISMATCH` or cause the `cairo_pattern_t` that is passed to it as its `this` pointer to enter into an error state where its status is `CAIRO_STATUS_PATTERN_TYPE_MISMATCH`. If either of those two events could occur, the function shall not become a member function of the `pattern` class but instead shall become a member function of the least-derived child type of `pattern` for which the `CAIRO_STATUS_PATTERN_TYPE_MISMATCH` error condition would not occur. [*Example:* `cairo_pattern_get_rgba` returns `CAIRO_STATUS_PATTERN_TYPE_MISMATCH` if the cairo pattern it is called on is not a solid color pattern. As such, the correct least-derived type for `get_rgba`, and thus the type that it becomes a member function of, is the `solid_color_pattern` type. Similarly, `cairo_pattern_add_color_stop_rgb` causes the `cairo_pattern_t` passed in as its first argument to enter the error state `CAIRO_STATUS_PATTERN_TYPE_MISMATCH` if it is not a gradient pattern. As such, the correct least-derived type for `add_color_stop_rgb` is the `gradient_pattern` type. – *end example*] [*Note:* This expresses the constraints in the type system, and so eliminates classes of runtime errors by replacing dynamic checks with static typing. – *end note*]

2.5. Any function which would become a non-constructor member function of the `surface` class by application of rule 2.3 which has a name containing *image_surface* shall instead become a member function of the `image_surface` class.

2.6. Any function which would become a non-constructor member function of the `surface` class by application of rule 2.3 which has a name containing *toy_font_face* shall instead become a member function of the `toy_font_face` class.

2.7. Any function parameter that is a pointer to cairo type *X* shall be converted so that it becomes an *lvalue reference* to type *X'*. It shall retain any *cv-qualifiers* that it possessed.

2.8. Any function which returns a pointer to *X* which has not been converted into a constructor of *X'* by application of rule 2.3 shall have its return type modified so that it returns an object of type *X'* by value.

2.8.1. Any function which returns a possibly *cv-qualified*, null-terminated pointer to `char` which is used to return a C-style string shall be modified to return a `::std::string`.

2.9. Any function which has as parameters a pointer to *X* followed by an integer parameter that denotes the number of elements of type *X* that the pointer points to (i.e. a C-style array with its length) shall be modified to replace the two parameters with an lvalue reference to *vector<X'>*, retaining any *cv-qualifiers* that applied to the pointer to *X* parameter. [*Example:* The cairo function `void cairo_glyph_path(cairo_t *cr, const cairo_glyph_t *glyphs, int num_glyphs);` becomes a member function of `context` with the signature `void glyph_path(const vector<glyph>& glyphs);` after the application of these rules. - *end example*]

2.9.1. Any function which has as a parameter a pointer to *X* which is documented as a pointer to a C-style array but which does not have a corresponding integer length parameter shall be modified to replace the pointer to *X* with an lvalue reference to *vector<X'>*, retaining any cv-qualifiers that applied to the pointer to *X* parameter. [*Example:* The cairo function `void cairo_get_dash(cairo_t *cr, double *dashes, double *offset);` becomes a member function of `context` with the signature `void get_dash(vector<double>& dashes, double& offset);` after the application of these rules. – *end example*]

2.9.2. Any function which has as a parameter a pointer to `char`, possibly followed by an integer parameter that denotes length, where the pointer to `char` is used to pass a C-style string shall be modified to replace the pointer to `char` with a reference to `::std::string`, retaining any *cv-qualifiers*, and to remove the integer length parameter, if any.

2.9.3. [*Note:* For any function which, after application of these rules, returns data to the caller via a non-`const` lvalue reference to *vector<X'>* parameter, the implementation shall ensure that the *vector<X'>* which is passed in as the argument for that parameter is empty or that any data contained in the *vector<X'>* is overwritten or otherwise removed. – *end note*]

2.10. Any function with the name pattern *cairo_X* which has not become a member function by operation of the previous rules shall become a standalone function with the name *X*.

2.11. The function `cairo_debug_reset_static_data` shall not have a corresponding function in this library.

2.12. The function `cairo_surface_get_type` shall not have a corresponding function in this library.

2.13. Any non-aggregate class for which the application of the previous rules did not produce an explicitly-defined default constructor shall not have a public default constructor.

2.14. Any class which is a base class as a result of the application of the previous rules shall have a public, virtual destructor. [*Example:* The `pattern` class and gradient_pattern class would both have public, virtual destructors as a result of this rule. – *end example*]

2.15. Every non-aggregate class shall be copyable and movable. These classes shall have shared ownership semantics. [*Note:* See 20.8.2.2 [util.smartptr.shared]/1. – *end note*] This rule does not apply to the `drawing_exception` class nor to the `user_data_key` class.

3. Native handles.

3.1. All classes described in these rules may have a member typedef `native_handle_type` and a member variable `native_handle` of that type. The presence of these members and their semantics is implementation-defined. [*Note:* These members allow implementations to provide access to implementation details. Their names are specified to facilitate portable compile-time detection. Actual use of these members is inherently non-portable. – *end note*]

3.2. A conforming implementation which implements rule 3.1 for a type may provide overloads of functions which have native handles of that type as parameters. Native handle parameters shall be placed before any non-native handle parameters in the function definition. If such a function has the same non-native handle parameters as an existing overload function (excluding any parameters of type *X'* which shall be replaced by an *X'* native handle), then any *X'* native handle that takes the place of an *X'* parameter of type *X'* shall appear in the same place as the *X'* parameter would have appeared and all non-native handle parameters shall appear in the same order as in the existing function. [*Example:* If function `void foo(const some_class& bar, some_class& baz, double x, double y);` exists and the implementer wishes to create an overload which takes a native handle for the *baz* parameter instead, the correct signature would be `void foo(const some_class& bar, some_class::native_handle_type baz, double x, double y);`. – *end example*] [*Note:* The intention of this rule and of rule 3.2 generally is to facilitate portability. When two (or more) implementers introduce functions which act upon or use native handles in a similar way and for a similar purpose as functions which do not use native handles, the intended result is that the native handle functions will have the same signature as each other provided that they each overload the same non-native handle function and have the same non-native handle parameters. Actual use of such functions is inherently non-portable and implementers are not required to guarantee that the observable result of any such function is the same as that of a non-native handle equivalent nor that of any other implementation. – *end note*]

4. Function pointers.

4.1. Any function pointers shall be converted to be a `::std::function` with the same return type and arguments. Wherever a function pointer appears as a parameter to a function, it shall be replaced with the equivalent `::std::function`. [*Note:* Wherever a `typedef` of a function pointer appears as a parameter of a cairo function, it shall be

treated as though the `typedef` did not exist and the function pointer appeared directly as the parameter type. – *end note*] No cairo function pointer `typedef` shall have a corresponding `typedef` in this library.

4.2. All parameter type and return type transformations in rule 2 shall be applied to function pointers. [*Example:* `typedef cairo_status_t (*cairo_write_func_t)(void *closure, const unsigned char *data, unsigned int length);` becomes `::std::function<void(void* closure, const ::std::vector<unsigned char>& data)>`. By application of rule 4.1, that `::std::function` appears directly as the parameter type of a transformed function wherever `cairo_write_func_t` appeared in the corresponding cairo function. – *end example*]

5. Error handling.

5.1. A new class type named `drawing_exception` shall be created in the `::std::experimental::drawing` namespace.

5.1.1. It shall publicly derive from `exception`.

5.1.2. It shall have a member function with the signature `status status() const noexcept;`.

5.1.3. It shall have a default constructor which shall create a `drawing_exception` object whose `status` member function returns the value `status::success`.

5.1.4. It shall have a constructor with the signature `explicit drawing_exception(status s) noexcept;` which shall create a `drawing_exception` object whose `status` member function returns the value of the `status s` parameter.

5.1.5. It shall be copy constructible and copy assignable.

5.1.6. Calling the `what` member function of an object of type `drawing_exception` shall return the same string that calling `cairo_status_to_string` with the `cairo_status_t` enumerator that is equivalent to the `drawing_exception` object's internal `status` value.

5.2. Any function or function pointer which returns cairo type `cairo_status_t` shall be converted so that it returns `void`. When executed, if the cairo function would have returned `CAIRO_STATUS_SUCCESS` when called with cairo equivalents of the transformed function's arguments and with any necessary cairo objects having a state that is equivalent to the state of their counterparts in this library, the transformed function shall return normally. If the cairo function would have returned a different `cairo_status_t` value, the transformed function shall throw an object of type `drawing_exception`. The return value of the status member function of the `drawing_exception` shall be the `status` enumerator value that is equivalent to the `cairo_status_t` value that the cairo function would have returned.

6. Miscellany.

6.1. A conforming implementation shall provide a function named `make_surface` in the `::std::experimental::drawing` namespace. This function shall return an object of type `surface`. Its parameter list is implementation defined.

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.

A. <drawing> header synopsis

```
namespace std {
  namespace experimental {
    namespace drawing {
      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 region_overlap {
        in,
        out,
```

```
  part
};

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,
  close_path
};

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

enum class filter {
  fast,
  good,
  best,
  nearest,
  bilinear,
  gaussian
};

enum class pattern_type {
  solid,
  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
};

struct text_cluster_flags {
  enum text_cluster_flags_t : int {
    backward = 0x1
  };
};

struct rectangle_int {
  int x;
  int y;
  int width;
  int height;
};

struct rectangle {
  double x;
  double y;
  double width;
  double height;
};
```

```
struct rectangle_list {
  status status;
  ::std::vector<rectangle> rectangles;
};

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

struct path {
  ::std::vector<path_data> data;
};

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;

  void init(double xx, double yx, double xy, double yy,
    double x0, double y0);
  void init_identity();
  void init_translate(double tx, double ty);
  void init_scale(double sx, double sy);
  void init_rotate(double radians);

  void translate(double tx, double ty);
  void scale(double sx, double sy);
  void rotate(double radians);
  void invert();
  static matrix multiply(const matrix& a, const matrix& b);
  void transform_distance(double& dx, double& dy);
  void transform_point(double& x, double& y);
};

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

  virtual ~drawing_exception();

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

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

struct region {
  region();
  region(const region& other);
  region& operator=(const region& other);
  region(region&& other);
  region& operator=(region&& other);
  explicit region(const rectangle_int& rectangle);
  explicit region(const ::std::vector<rectangle_int>& rectangles);

  status status();

  void get_extents(rectangle_int& extents);
  int num_rectangles();
  void get_rectangle(int nth, rectangle_int& rectangle);
  bool is_empty();
  bool contains_point(int x, int y);
  region_overlap contains_rectangle(const rectangle_int& rectangle);

  bool equal(const region& other);
  void translate(int dx, int dy);
```

```
  void intersect_region(const region& other);
  void intersect_rectangle(const rectangle_int& rectangle);
  void subtract_region(const region& other);
  void subtract_rectangle(const rectangle_int& rectangle);
  void union_region(const region& other);
  void union_rectangle(const rectangle_int& rectangle);
  void xor_region(const region& other);
  void xor_rectangle(const rectangle_int& rectangle);
};

struct user_data_key {
  user_data_key();

  bool operator==(const user_data_key& other) const;
  bool operator!=(const user_data_key& other) const;
};

struct device {
  device(const device& other);
  device& operator=(const device& other);
  device(device&& other);
  device& operator=(device&& other);
  status status();
  void finish();
  void flush();
  void set_user_data(const user_data_key& key,
    ::std::shared_ptr<void>& value);
  ::std::shared_ptr<void>& get_user_data(const user_data_key& key);
  void acquire();
  void release();
private:
  device(); // Expositive only.
};

struct font_options {
  font_options(const font_options& other);
  font_options& operator=(const font_options& other);
  font_options(font_options&& other);
  font_options& operator=(font_options&& other);
  status status();

  void merge(const font_options& other);
  unsigned long hash();
  bool equal(const font_options& other);
  void set_antialias(antialias antialias);
  antialias get_antialias();
  void set_subpixel_order(subpixel_order subpixel_order);
  subpixel_order get_subpixel_order();
  void set_hint_style(hint_style hint_style);
  hint_style get_hint_style();
  void  set_hint_metrics(hint_metrics hint_metrics);
  hint_metrics get_hint_metrics();
private:
  font_options(); // Expositive only.
```

```
      };

      struct font_face {
        font_face(const font_face& other);
        font_face& operator=(const font_face& other);
        font_face(font_face&& other);
        font_face& operator=(font_face&& other);
      protected:
        font_face(); // Expositive only.
      };

      struct scaled_font {
        scaled_font(const scaled_font& other);
        scaled_font& operator=(const scaled_font& other);
        scaled_font(scaled_font&& other);
        scaled_font& operator=(scaled_font&& other);
      private:
        scaled_font(); // Expositive only.
      };

      struct toy_font_face : public font_face {
        toy_font_face(const toy_font_face& other);
        toy_font_face& operator=(const toy_font_face& other);
        toy_font_face(toy_font_face&& other);
        toy_font_face& operator=(toy_font_face&& other);
        toy_font_face(const ::std::string& family, font_slant slant,
          font_weight weight);
        ::std::string get_family();
        font_slant get_slant();
        font_weight get_weight();
      private:
        toy_font_face(); // Expositive only.
      };

      struct surface {
        surface(const surface& other);
        surface& operator=(const surface& other);
        surface(surface&& other);
        surface& operator=(surface&& other);
        surface(surface& other, content content, int width, int height);
        surface(surface& target, double x, double y, double width, double
height);
        status status();
        void finish();
        void flush();
        device get_device();
        void get_font_options(font_options& options);
        content get_content();
        void mark_dirty();
        void mark_dirty_rectangle(int x, int y, int width, int height);
        void set_device_offset(double x_offset, double y_offset);
        void set_fallback_resolution(double x_pixels_per_inch,
          double y_pixels_per_inch);
```

```
  void get_fallback_resolution(double& x_pixels_per_inch,
    double& y_pixels_per_inch);
  void write_to_png(const ::std::string& filename);
  virtual void write_to_png_stream(
    ::std::function<void(void* closure,
    const ::std::vector<unsigned char>& data)> write_fn,
    void* closure);
  void set_user_data(const user_data_key& key,
    ::std::shared_ptr<void>& value);
  ::std::shared_ptr<void>& get_user_data(const user_data_key& key);
protected:
  surface(); // Expositive only.
};

struct image_surface : public surface {
  image_surface(const image_surface& other);
  image_surface& operator=(const image_surface& other);
  image_surface(image_surface&& other);
  image_surface& operator=(image_surface&& other);
  image_surface(format format, int width, int height);
  image_surface(::std::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);
  image_surface(
    ::std::function<void(void* closure,
    ::std::vector<unsigned char>& data)> read_fn, void* closure);
  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();
private:
  image_surface(); // Expositive only.
};

struct pattern {
  pattern(const pattern& other);
  pattern& operator=(const pattern& other);
  pattern(pattern&& other);
  pattern& operator=(pattern&& other);
  status status();
  void set_extend(extend extend);
  extend get_extend();
  void set_filter(filter filter);
  filter get_filter();
  void set_matrix(const matrix& matrix);
  void get_matrix(matrix& matrix);
  pattern_type get_type();
  void set_user_data(const user_data_key& key,
    ::std::shared_ptr<void>& value);
  ::std::shared_ptr<void>& get_user_data(const user_data_key& key);
```

```
protected:
  pattern(); // Expositive only.
};

struct solid_color_pattern : public pattern {
  solid_color_pattern(const solid_color_pattern& other);
  solid_color_pattern& operator=(const solid_color_pattern& other);
  solid_color_pattern(solid_color_pattern&& other);
  solid_color_pattern& operator=(solid_color_pattern&& other);
  solid_color_pattern(double red, double green, double blue);
  solid_color_pattern(double red, double green, double blue,
    double alpha);
  void get_rgba(double& red, double& green, double& blue,
    double& alpha);
private:
  solid_color_pattern(); // Expositive only.
};

struct gradient_pattern : public pattern {

  gradient_pattern(const gradient_pattern& other);
  gradient_pattern& operator=(const gradient_pattern& other);
  gradient_pattern(gradient_pattern&& other);
  gradient_pattern& operator=(gradient_pattern&& other);
  void add_color_stop_rgb(double offset, double red, double green,
    double blue);
  void add_color_stop_rgba(double offset, double red, double green,
    double blue, double alpha);
  void get_color_stop_count(int& count);
  void get_color_stop_rgba(int index, double& offset, double& red,
    double& green, double& blue, double& alpha);
protected:
  gradient_pattern(); // Expositive only.
};

struct linear_pattern : public gradient_pattern {
  linear_pattern(const linear_pattern& other);
  linear_pattern& operator=(const linear_pattern& other);
  linear_pattern(linear_pattern&& other);
  linear_pattern& operator=(linear_pattern&& other);
  linear_pattern(double x0, double y0, double x1, double y1);
  void get_linear_points(double& x0, double& y0, double& x1,
    double& y1);
private:
  linear_pattern(); // Expositive only.
};

struct radial_pattern : public gradient_pattern {
  radial_pattern(const radial_pattern& other);
  radial_pattern& operator=(const radial_pattern& other);
  radial_pattern(radial_pattern&& other);
  radial_pattern& operator=(radial_pattern&& other);
  radial_pattern(double cx0, double cy0, double radius0, double cx1,
    double cy1, double radius1);
```

```
  void get_radial_circles(double& x0, double& y0, double& radius0,
    double& x1, double& y1, double& radius1);
private:
  radial_pattern(); // Expositive only.
};

struct surface_pattern : public pattern {
  surface_pattern(const surface_pattern& other);
  surface_pattern& operator=(const surface_pattern& other);
  surface_pattern(surface_pattern&& other);
  surface_pattern& operator=(surface_pattern&& other);
  explicit surface_pattern(surface& surface);
  void get_surface(surface& s);
private:
  surface_pattern(); // Expositive only.
};

struct mesh_pattern : public pattern {
  mesh_pattern();
  mesh_pattern(const mesh_pattern& other);
  mesh_pattern& operator=(const mesh_pattern& other);
  mesh_pattern(mesh_pattern&& other);
  mesh_pattern& operator=(mesh_pattern&& other);
  void begin_patch();
  void end_patch();
  void move_to(double x, double y);
  void line_to(double x, double y);
  void curve_to(double x1, double y1, double x2, double y2,
    double x3, double y3);
  void set_control_point(unsigned int point_num, double x,
    double y);
  void set_corner_color_rgb(unsigned int corner_num, double red,
    double green, double blue);
  void set_corner_color_rgba(unsigned int corner_num, double red,
    double green, double blue, double alpha);
  void get_patch_count(unsigned int& count);
  path get_path(unsigned int patch_num);
  void get_control_point(unsigned int patch_num,
    unsigned int point_num, double& x, double& y);
  void get_corner_color_rgba(unsigned int patch_num,
    unsigned int corner_num, double& red, double& green,
    double& blue, double& alpha);
};

struct raster_source_pattern : public pattern {
  raster_source_pattern(const raster_source_pattern& other);
  raster_source_pattern& operator=(
    const raster_source_pattern& other);
  raster_source_pattern(raster_source_pattern&& other);
  raster_source_pattern& operator=(raster_source_pattern&& other);
  raster_source_pattern(void* user_data, content content, int width,
    int height);
  void set_callback_data(void* data);
  void* get_callback_data();
```

```
  void set_acquire(
    ::std::function<surface(void* callback_data, surface& target,
    const rectangle_int& 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_int& extents)>& acquire_fn,
    ::std::function<void(void* callback_data,
    surface& surface)>& release_fn
    );
  void set_snapshot(
    ::std::function<experimental::drawing::status(
    void* callback_data)> snapshot_fn
    );
  void get_snapshot(
    ::std::function<experimental::drawing::status(
    void* callback_data)>& snapshot_fn
    );
  void set_copy(
    ::std::function<experimental::drawing::status(
    void* callback_data, const pattern& other)> copy_fn
    );
  void get_copy(
    ::std::function<experimental::drawing::status(
    void* callback_data, const pattern& other)>& copy_fn
    );
  void set_finish(::std::function<void(
    void* callback_data)> finish_fn);
  void get_finish(::std::function<void(
    void* callback_data)>& finish_fn);
private:
  raster_source_pattern(); // Expositive only.
};

struct context {
  explicit context(surface& s);
  context(const context& other);
  context& operator=(const context& other);
  context(context&& other);
  context& operator=(context&& other);

  status status();
  void save();
  void restore();
  surface get_target();
  void push_group();
  void push_group_with_content(content c);
  pattern pop_group();
  void pop_group_to_source();
  surface get_group_target();

  void set_source_rgb(double red, double green, double blue);
```

```
void set_source_rgba(double red, double green, double blue,
  double alpha);
void set_source(const pattern& source);
void set_source_surface(const surface& s, double x, double y);
pattern get_source();

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

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_preserve();
void clip_extents(double& x1, double& y1, double& x2, double& y2);
bool in_clip(double x, double y);
void reset_clip();

rectangle_list copy_clip_rectangle_list();

void fill();
void fill_preserve();
void fill_extents(double& x1, double& y1, double& x2, double& y2);
bool in_fill(double x, double y);

void mask(pattern& pattern);
void mask_surface(surface& surface, double surface_x,
  double surface_y);

void paint();
void paint_with_alpha(double alpha);
```

```
void stroke();
void stroke_preserve();
void stroke_extents(double& x1, double& y1, double& x2,
  double& y2);
bool in_stroke(double x, double y);

void copy_page();
void show_page();

void set_user_data(const user_data_key& key,
  ::std::shared_ptr<void>& value);
::std::shared_ptr<void>& get_user_data(const user_data_key& key);

// Paths
path copy_path();
path copy_path_flat();
void append_path(const path& p);
bool has_current_point();
void get_current_point(double& x, double& y);
void new_path();
void new_sub_path();
void close_path();
void arc(double xc, double yc, double radius, double angle1,
  double angle2);
void arc_negative(double xc, double yc, double radius,
  double angle1, double angle2);
void curve_to(double x1, double y1, double x2, double y2,
  double x3, double y3);
void line_to(double x, double y);
void move_to(double x, double y);
void rectangle(double x, double y, double width, double height);
void glyph_path(const ::std::vector<glyph>& glyphs);
void text_path(const ::std::string& utf8);
void rel_curve_to(double dx1, double dy1, double dx2, double dy2,
  double dx3, double dy3);
void rel_line_to(double dx, double dy);
void rel_move_to(double dx, double dy);
void path_extents(double& x1, double& y1, double& x2, double& y2);

// Transformations
void translate(double tx, double ty);
void scale(double sx, double sy);
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(double& x, double& y);
void user_to_device_distance(double& dx, double& dy);
void device_to_user(double& x, double& y);
void device_to_user_distance(double& dx, double& dy);
```

```
        // 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);
        void get_font_options(font_options& 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 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);

      private:
        context(); // Expositive only.
      };

      surface make_surface(); // Parameter list is expositive only.
      int format_stride_for_width(format format, int width);
    }
  }
}
```

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. References

N3791 – Lightweight Drawing Library

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

The cairo API Documentation - Version 1.12.16