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

Note: this is an early draft. It’s known to be incomplet and incorrekt, and it has lots of bad formatting.
# Contents

Contents ii

List of Tables iv

List of Figures v

## Revision history

0.1 Revision 8 .......................... 1
0.2 Revision 7 .......................... 2
0.3 Revision 6 .......................... 3

## Scope

4

## Normative references

5

## Terms and definitions

6

## Error reporting

11

## Header `<experimental/io2d> synopsis`

12

## Colors

6.1 Introduction to color ......................... 19
6.2 Color usage requirements ..................... 19
6.3 Class `rgba_color` .......................... 19

## Graphics math

7.1 General .................................. 29
7.2 Requirements ................................ 29

## Graphics surfaces

8.1 General .................................. 37
8.2 Requirements ................................ 37

## Linear algebra

9.1 Class `basic_point_2d` ...................... 72
9.2 Class `basic_matrix_2d` ..................... 75

## Geometry

10.1 Class template `basic_display_point` ........ 80
10.2 Class `basic_bounding_box` ................. 81
10.3 Class `basic_circle` ....................... 83

## Text rendering and display

11 Text rendering and display .................... 86

## Paths

12 Paths ..................................... 87
12.1 Overview of paths ......................... 87
12.2 Path examples (Informative) ................ 87
12.3 Class template `basic_figure_items` ........ 91
12.4 Class `basic_interpreted_path` ............. 114
12.5 Class `basic_path_builder` ................. 115
List of Tables

1  Class identifiers modified since R6 .................................................. 2
2  rgba_color static members values ..................................................... 25
3  Graphics math summary ................................................................. 29
4  X typedef-names ........................................................................ 29
5  GraphicsMath type member data .................................................... 29
6  GraphicsMath requirements .......................................................... 30
7  X::brushes requirements .............................................................. 38
8  X::paths typedef-names ............................................................... 41
9  X::paths type member data ........................................................... 42
10 X::paths requirements ................................................................. 42
11 X::surface_stateProps typedef-names ........................................... 47
12 X::surface_stateProps type member data ....................................... 47
13 X::surface_stateProps requirements ............................................. 48
14 X::surfaces typedef-names .......................................................... 53
15 X::surfaces type member data ...................................................... 53
16 Graphics surfaces requirements .................................................... 54
17 Path interpretation state data ......................................................... 113
18 Figure item interpretation effects .................................................. 113
19 wrap_mode enumerator meanings .................................................. 126
20 filter enumerator meanings ......................................................... 127
21 brush_type enumerator meanings ................................................ 127
22 antialias enumerator meanings ..................................................... 132
23 fill_rule enumerator meanings ..................................................... 133
24 line_cap enumerator meanings ..................................................... 134
25 line_join enumerator meanings .................................................... 134
26 compositing_op basic enumerator meanings .................................. 136
27 compositing_op blend enumerator meanings .................................. 137
28 compositing_op hsl enumerator meanings ...................................... 139
29 format enumerator meanings ....................................................... 140
30 scaling enumerator meanings ....................................................... 141
31 refresh_style value meanings ....................................................... 155
32 imagefileformat enumerator meanings .......................................... 156
33 surface rendering and composing operations .................................. 157
34 surface rendering and composing common state data ...................... 158
35 surface rendering and composing specific state data ....................... 158
36 Point transformations ................................................................. 159
37 Output surface observable state ................................................... 161
# List of Figures

1. Example 1 result ................................................................. 88
2. Example 2 result ................................................................. 89
3. Path example 3 ................................................................. 90
4. Path example 4 ................................................................. 91
0 Revision history

0.1 Revision 8

1 Modified the revision 7 notes (0.2) to denote trademarks where applicable, and to use the correct capitalization for the cairo graphics library. The contents of those notes is otherwise unchanged.

2 Changed the Revision history Clause 0 to be Clause 0.

3 Added a new Clause Graphics math (Clause 7), which defines the requirements of a type that conforms to the GraphicsMath template parameter used by various classes.

4 Updated the relevant class member functions in this proposal to define their effects to include calls to the appropriate GraphicsMath functions. This completes the work begun in P0267R7, of abstracting the implementation of the linear algebra and geometry classes, thereby allowing users to specify a preferred implementation of the mathematical functionality used in this proposal.

5 Added a new Clause Graphics surfaces (Clause 8), which defines the requirements of a type that conforms to the GraphicsSurfaces template parameter used by various classes.

6 Updated the relevant class member functions in this proposal to define their effects to include calls to the appropriate GraphicsSurfaces functions. This completes the work begun in P0267R7, of abstracting the implementation of the brush, paths, surface state, and surface classes, thereby allowing users to specify a preferred implementation of the functionality specified in this proposal.

7 Added a new Clause Surface state props (Clause 14) and moved the relevant enum class types and the basic_render_props, basic_brush_probs, basic_clip_probs, basic_stroke_probs, and basic_mask_probs class templates to it.

8Added Michael Kazakov as a co-author. He has written an implementation of this proposal using the Core Graphics framework of Cocoa®, thus providing a native implementation for iOS® and OS X®. It is available as part of the reference implementation (See 0.2).

9 He has also written a series of tests for compliance. This has drawn attention to several issues that require some revision.

10 Eliminated format::rgb16_565 and format::rgb30.

11 Eliminated compositing_op::dest since it is a no-op.

12 Significant cleanup of terms and definitions.

13 Added overload of copy_surface for basic_output_surface.

14 Removed format_stride_for_width: it has had no use since mapping functionality was removed.

15 Added functions degrees_to_radians and radians_to_degrees.

16 Added equality comparison operators for a number of classes.

17 Removed the copyright notice that stated that the proposal was copyrighted by ISO/IEC. Neither organization, jointly or severally, made any contribution to this document and no assignment of interests by the authors to either organization, jointly or severally, has ever been executed. The notice was there unintentionally and its presence in all revisions of P0267 was a mistake.

18 Added basic_dashes which was added in R7 but had its description omitted accidentally.

19 Removed the mandate of underlying layout of pixel formats in enum class format and made it and, the interpretation of the data (i.e. what each bit value in each channel means), and whether data is in a premultiplied format implementation defined.

20 Added GraphicsSurfaces::additional_formats This allows implementations to support additional visual data formats.

21 Eliminated all flush and mark_dirty member functions. These only existed to allow users to modify surfaces externally. Implementations that wish to allow users to modify surfaces externally should provide and document their own functionality for how to do that. The errors, etc., are all implementation dependent anyway so a uniform calling interface provides no benefit at all in the current templated-design.
Renamed `enum class refresh_rate` to `refresh_style` to more accurately reflect its meaning. This was already done in parts of the R7; it is now complete.

Changed the order of items in the `basic_figure_items::figure_item` type alias from alphabetical to grouping by function (e.g., `abs_new_figure`, `rel_new_figure`, and `close_figure` are grouped together and `abs_line` and `rel_line` are grouped together). While it’s not expected that any new figure item types will be added, there is no chance that the existing ones will be augmented with additional types. So if new figure items are added, grouping by type will simply add them to the end, thus preserving the validity of the existing index values without having the existing entries be alphabetized and new entries not being alphabetized.

Moved the class template definitions for the nested classes within `basic_figure_items<GraphicsSurfaces>`, to the descriptions of each of those types from the synopsis of `basic_figure_items<GraphicsSurfaces>` itself.

Added `format::xrgb16`. The number of bits per channel is left to the implementation since, e.g., Windows® is 565 whereas OS X® and iOS® are 555 with an unused bit. This is useful for platforms with limited memory where supported so having it as an official enumerator will help.

Users can now request a different output device format when calling the overloads of the `basic_output_surface ctor` and the `basic_unmanaged_output_surface ctor` that take separate output device width and height preferences.

Eliminated `redraw_required` from `basic_unmanaged_output_surface`. Users can and should track the need to redraw in their own code when they manage the output device.

Eliminated `user_scaling_callback` functionality from `basic_output_surface` and `basic_unmanaged_output_surface` since the output device is intentionally not fully specified (same as stdout, etc.).

`begin_show` now returns void instead of int and has an error code overload in case the user tries to show more output surfaces than the system permits.

`render_props` now has a `filter` instead of an `antialias`.

`stroke_props` now has an `antialias` instead of a `filter`.

New type `basic_fill_props` for parameters specific to the fill operation.

Removed the `fill_rule` from `basic_brush_props` as it was only being used for fill operations.

### 0.2 Revision 7

The significant difference between R7 and R6 is the abstraction of the implementation into separate classes. These classes provide math and rendering support. The linear algebra and geometry classes are templated over any appropriate math support class, while the path, brush and surface classes are templated over any appropriate rendering support class.

The reference implementation of this paper provides a software implementation of the math and rendering support classes. This is based on cairo; indeed, so far the reference implementation has been based on cairo. However, it is now possible to provide an implementation more appropriate to the target platform.

For example, a Windows® implementation could provide support classes based on DirectX®, while a Linux® implementation could provide support classes based on OpenGL®. In fact, any hardware vendor could provide a support library, targeting a specific implementation and their particular silicon if they wanted to exploit particular features of their hardware.

Additionally, the surface classes have been modified: now there are simply managed and unmanaged output surfaces, the latter of which offers developers the opportunity to take finer control of the drawing surface.

The modified classes are as follows:

<table>
<thead>
<tr>
<th>R6 identifier</th>
<th>R7 identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector_2d</td>
<td>basic_point_2d</td>
</tr>
<tr>
<td>matrix_2d</td>
<td>basic_matrix_2d</td>
</tr>
<tr>
<td>rectangle</td>
<td>basic_bounding_box</td>
</tr>
<tr>
<td>circle</td>
<td>basic_circle</td>
</tr>
<tr>
<td>path_group</td>
<td>basic_interpreted_path</td>
</tr>
<tr>
<td>path_builder</td>
<td>basic_path_builder</td>
</tr>
<tr>
<td>color_stop</td>
<td>gradient_stop</td>
</tr>
</tbody>
</table>

Table 1 — Class identifiers modified since R6
<table>
<thead>
<tr>
<th>R6 identifier</th>
<th>R7 identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>brush</td>
<td>basic_brush</td>
</tr>
<tr>
<td>render.props</td>
<td>basic_render_props</td>
</tr>
<tr>
<td>brush.props</td>
<td>basic_brush_props</td>
</tr>
<tr>
<td>clip.props</td>
<td>basic_clip_props</td>
</tr>
<tr>
<td>stroke.props</td>
<td>basic_stroke_props</td>
</tr>
<tr>
<td>mask.props</td>
<td>basic_mask_props</td>
</tr>
<tr>
<td>image_surface</td>
<td>basic_image_surface</td>
</tr>
<tr>
<td>display_surface</td>
<td>basic_output_surface</td>
</tr>
</tbody>
</table>

6 The `surface` class and the `mapped_surface` class have been withdrawn, while the `basic_unmanaged_output_surface` class has been introduced.

7 The reference implementation, including a software-only implementation of math and rendering support classes, is available at https://github.com/mikebmcl/P0267_RefImpl

0.3 Revision 6

[io2d.revisionhistory.r6]

1 Presented to LEWG in Toronto, July 2017
1 Scope

This Technical Specification specifies requirements for implementations of an interface that computer programs written in the C++ programming language may use to render and display 2D computer graphics.
Normative references

1 The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882, Programming languages — C++
(1.2) — ISO/IEC 2382 (all parts), Information technology — Vocabulary
(1.3) — ISO/IEC 10646-1:1993, Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane
(1.4) — ISO/IEC 10918-1, Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines
(1.5) — ISO 12639, Graphic technology — Prepress digital data exchange — Tag image file format for image technology (TIFF/IT)
(1.6) — ISO/IEC 15948 Information technology — Computer graphics and image processing — Portable Network Graphics (PNG) Functional specification
(1.7) — ISO/IEC TR 19769:2004, Information technology — Programming languages, their environments and system software interfaces — Extensions for the programming language C to support new character data types
(1.8) — ISO 15076-1, Image technology colour management — Architecture, profile format and data structure — Part 1: Based on ICC.1:2004-10
(1.9) — IEC 61966-2-1, Colour Measurement and Management in Multimedia Systems and Equipment - Part 2-1: Default RGB Colour Space - sRGB
(1.11) — ISO 80000-2:2009, Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology
(1.12) — Tantek Çelik et al., CSS Color Module Level 3 — W3C Recommendation 07 June 2011, Copyright © 2011 W3C® (MIT, ERCIM, Keio)

2 The compressed image data format described in ISO/IEC 10918-1 is hereinafter called the JPEG format.
3 The tag image file format described in ISO 12639 is hereinafter called the TIFF format. The datastream and associated file format described in ISO/IEC 15948 is hereinafter called the PNG format.
5 The library described in ISO/IEC TR 19769:2004 is hereinafter called the C Unicode TR.
6 The document CSS Color Module Level 3 — W3C Recommendation 07 June 2011 is hereinafter called the CSS Colors Specification.
3 Terms and definitions

For the purposes of this document, the following terms and definitions apply. ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at http://www.iso.org/obp

1 Terms that are used only in a small portion of this document are defined where they are used and italicized where they are defined.

3.1 point
coordinate designated by a floating-point x axis value and a floating-point y axis value

3.2 origin
point with an x axis value of 0 and a y axis value of 0

3.3 standard coordinate space
Euclidean plane described by a Cartesian coordinate system where the x axis is a horizontal axis oriented from left to right, the y axis is a vertical axis oriented from top to bottom, and rotation of a point, excluding the origin, around the origin by a positive value in radians is counterclockwise

3.4 integral point
point where the x axis value and the y axis value are integers

3.5 normalize
map a closed set of evenly spaced values in the range [0, x] to an evenly spaced sequence of floating-point values in the range [0, 1] [Note: The definition of normalize given is the definition for normalizing unsigned input. Signed normalization, i.e. the mapping of a closed set of evenly spaced values in the range [−x, x) to an evenly spaced sequence of floating-point values in the range [−1, 1] is not used in this Technical Specification — end note]

3.6 aspect ratio
ratio of the width to the height of a rectangular area

3.7 visual data
data in a possibly bounded Euclidean plane consisting of one or more components representing color, transparency, or some other quality where the component values are not necessarily uniform throughout the plane

3.8 visual data element
unit of visual data at a specific point

3.9 channel
component of visual data

3.10 color channel
channel that only represents the intensity of a specific color
3.11 alpha channel
channel that only represents transparency

3.12 visual data format
specification of information necessary to transform a set of one or more channels into colors in a color model

3.13 premultiplied format
visual data format with one or more color channels and an alpha channel where each color channel is normalized and then multiplied by the normalized alpha channel value. **Example:** Given the 32-bit non-premultiplied RGBA pixel with 8 bits per channel \{255, 0, 0, 127\} (half-transparent red), when normalized it would become \{1.0f, 0.0f, 0.0f, 0.5f\}. When premultiplied it would become \{0.5f, 0.0f, 0.0f, 0.5f\} as a result of multiplying each of the three color channels by the alpha channel value. — *end example*

3.14 raster graphics data
data comprised of a rectangular array of visual data elements together with their visual data format where the top-left visual data element is located at the origin in the standard coordinate space and additional visual data elements are located at integral points of consecutive values

3.15 pixel
discrete element of raster graphics data

3.16 vector graphics data
data comprised of zero or more paths together with a sequence of rendering and composing operations and graphics state data that produces continuous visual data when processed

3.17 color model
ideal, mathematical representation of color

3.18 additive color
color defined by the emissive intensity of its color channels

3.19 RGB color model
color model using additive color comprised of red, green, and blue color channels

3.20 RGBA color model
RGB color model with an alpha channel

3.21 color space
systematic mapping of values to colorimetric colors

3.22 sRGB color space
color space defined in IEC 61966-2-1 that is based on the RGB color model

3.23 start point
point that begins a segment
3.24  
**end point**  
point that ends a segment

3.25  
**control point**  
point, other than the start point and the end point, that is used in defining a curve

3.26  
**Bézier curve**  
(quadratic) curve defined by the equation \( f(t) = (1-t)^2 \times P_0 + 2 \times t \times (1-t) \times P_1 + t^2 \times t \times P_2 \) where \( t \) is in the range \([0, 1]\), \( P_0 \) is the start point, \( P_1 \) is the control point, and \( P_2 \) is end point

3.27  
**Bézier curve**  
(cubic) curve defined by the equation \( f(t) = (1-t)^3 \times P_0 + 3 \times t \times (1-t)^2 \times P_1 + 3 \times t^2 \times (1-t) \times P_2 + t^3 \times t \times P_3 \) where \( t \) is in the range \([0, 1]\), \( P_0 \) is the start point, \( P_1 \) is the first control point, \( P_2 \) is the second control point, and \( P_3 \) is the end point

3.28  
**segment**  
line, Bézier curve, or arc

3.29  
**initial segment**  
segment in a figure whose start point is not defined as being the end point of another segment in the figure  
[Note: It is possible for the initial segment and final segment to be the same segment. — end note]

3.30  
**new figure point**  
point that is the start point of the initial segment

3.31  
**final segment**  
segment in a figure whose end point does not define the start point of any other segment  
[Note: It is possible for the initial segment and final segment to be the same segment. — end note]

3.32  
**current point**  
point used as the start point of a segment

3.33  
**open figure**  
figure with one or more segments where the new figure point is not used to define the end point of the figure’s final segment  
[Note: Even if the start point of the initial segment and the end point of the final segment are assigned the same coordinates, the figure is still an open figure. This is because the final segment’s end point is not defined as being the new figure point but instead merely happens to have the same value as that point. — end note]

3.34  
**closed figure**  
figure with one or more segments where the new figure point is used to define the end point of the figure’s final segment

3.35  
**degenerate segment**  
segment that has the same values for its start point, end point, and, if any, control points
3.36 command
(close figure command) instruction that creates a line segment with a start point of current point and an end point of new figure point

3.37 command
(new figure command) an instruction that creates a new path

3.38 figure item
segment, new figure command, close figure command, or path command

3.39 figure
collection of figure items where the end point of each segment in the collection, except the final segment, defines the start point of exactly one other segment in the collection

3.40 path
collection of figures

3.41 path transformation matrix
affine transformation matrix used to apply affine transformations to the points in a path

3.42 path command
instruction that modifies the path transformation matrix

3.43 degenerate figure
figure containing a new figure command, zero or more degenerate segments, zero or more path commands, and, optionally, a close figure command

3.44 graphics state data
data which specify how some part of the process of rendering or composing is performed in part or in whole

3.45 render
transform a path into visual data

3.46 composition algorithm
algorithm that combines source visual data and destination visual data producing visual data that has the same visual data format as the destination visual data

3.47 compose
apply a composition algorithm

3.48 rendering and composing operation
operation that is either a composing operation or a rendering operation followed by a composing operation that uses the data produced by the rendering operation

3.49 filter
algorithm that determines a color value from a raster graphics data source for a non-integral point

§ 3.49
3.50 sample
apply a filter

3.51 aliasing
errors in the appearance of the results of rendering where the resulting visual data is raster graphics data because of inaccuracies in transforming continuous data into discrete data

3.52 anti-aliasing
application of an algorithm while rendering to reduce aliasing
4 Error reporting

1 2D graphics library functions that can produce errors occasionally provide two overloads: one that throws an exception to report errors and another that reports errors using an `error_code` object. This provides for situations where errors are not truly exceptional.

2 report errors as follows, unless otherwise specified:

3 When an error prevents the function from meeting its specifications:

   (3.1) Functions that do not take argument of type `error_code&` throw an exception of type `system_error` or of an implementation-defined type that derives from `system_error`. The exception object shall include the enumerator specified by the function as part of its observable state.

   (3.2) Functions that take an argument of type `error_code&` assigns the specified enumerator to the provided `error_code` object and then returns.

4 Failure to allocate storage is reported by throwing an exception as described in [res.on.exception.handling] in C++ 2017.

5 Destructor operations defined in this Technical Specification shall not throw exceptions. Every destructor in this Technical Specification shall behave as-if it had a non-throwing exception specification.

6 If no error occurs in a function that takes an argument of type `error_code&`, `error_code::clear` shall be called on the `error_code` object immediately before the function returns.
namespace std {
    namespace experimental {
        namespace io2d {
            // ... (rest of the code snippet) ...
        }
    }
}

5 Header <experimental/io2d> synopsis [io2d.syn]

namespace std {
    namespace experimental {
        namespace io2d {
            inline namespace v1 {
                template <class T>
                constexpr T pi = T(3.14159265358979323846264338327950288L);
                template <class T>
                constexpr T two_pi = T(6.28318530717958647692528676655900577L);
                template <class T>
                constexpr T half_pi = T(1.57079632679489661923132169163975144L);
                template <class T>
                constexpr T three_pi_over_two = T(4.71238898038468985769396507491925432L);
                template <class T>
                constexpr T tau = T(6.28318530717958647692528676655900577L);
                template <class T>
                constexpr T three_quarters_tau = T(4.71238898038468985769396507491925432L);
                template <class T>
                constexpr T half_tau = T(3.14159265358979323846264338327950288L);
                template <class T>
                constexpr T quarter_tau = T(1.57079632679489661923132169163975144L);

                template <class T>
                constexpr T degrees_to_radians(T deg) noexcept;
                template <class T>
                constexpr T radians_to_degrees(T rad) noexcept;

                class rgba_color;
                constexpr bool operator==(const rgba_color& lhs, const rgba_color& rhs) noexcept;
                constexpr bool operator!=(const rgba_color& lhs, const rgba_color& rhs) noexcept;
                template <class T>
                constexpr rgba_color operator*(const rgba_color& lhs, T rhs) noexcept;
                template <class U>
                constexpr rgba_color operator*(const rgba_color& lhs, U rhs) noexcept;
                template <class T>
                constexpr rgba_color operator*(T lhs, const rgba_color& rhs) noexcept;
                template <class U>
                constexpr rgba_color operator*(U lhs, const rgba_color& rhs) noexcept;

                class gradient_stop;
                constexpr bool operator==(const gradient_stop& lhs, const gradient_stop& rhs) noexcept;
                constexpr bool operator!=(const gradient_stop& lhs, const gradient_stop& rhs) noexcept;

                template <class GraphicsMath>
                class basic_bounding_box;
                template <class GraphicsMath>
                bool operator==(const basic_bounding_box<GraphicsMath>& lhs, const basic_bounding_box<GraphicsMath>& rhs) noexcept;
                template <class GraphicsMath>
                bool operator!=(const basic_bounding_box<GraphicsMath>& lhs, const basic_bounding_box<GraphicsMath>& rhs) noexcept;
            }
        }
    }
}

Header <experimental/io2d> synopsis
template <class GraphicsSurfaces>
class basic_brush;
template <class GraphicsSurfaces>
bool operator==(const basic_brush<GraphicsSurfaces>& lhs,
    const basic_brush<GraphicsSurfaces>& rhs) noexcept;
template <class GraphicsSurfaces>
bool operator!=(const basic_brush<GraphicsSurfaces>& lhs,
    const basic_brush<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsSurfaces>
class basic_brush_props;
template <class GraphicsSurfaces>
bool operator==(const basic_brush_props<GraphicsSurfaces>& lhs,
    const basic_brush_props<GraphicsSurfaces>& rhs) noexcept;
template <class GraphicsSurfaces>
bool operator!=(const basic_brush_props<GraphicsSurfaces>& lhs,
    const basic_brush_props<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsMath>
class basic_circle;
template <class GraphicsMath>
bool operator==(const basic_circle<GraphicsMath>& lhs,
    const basic_circle<GraphicsMath>& rhs) noexcept;
template <class GraphicsMath>
bool operator!=(const basic_circle<GraphicsMath>& lhs,
    const basic_circle<GraphicsMath>& rhs) noexcept;

template <class GraphicsSurfaces>
class basic_clip_props;
template <class GraphicsSurfaces>
bool operator==(const basic_clip_props<GraphicsSurfaces>& lhs,
    const basic_clip_props<GraphicsSurfaces>& rhs) noexcept;
template <class GraphicsSurfaces>
bool operator!=(const basic_clip_props<GraphicsSurfaces>& lhs,
    const basic_clip_props<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsSurfaces>
class basic_dashes;
template <class GraphicsSurfaces>
bool operator==(const basic_dashes<GraphicsSurfaces>& lhs,
    const basic_dashes<GraphicsSurfaces>& rhs) noexcept;
template <class GraphicsSurfaces>
bool operator!=(const basic_dashes<GraphicsSurfaces>& lhs,
    const basic_dashes<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsSurfaces>
class basic_display_point;
template <class GraphicsSurfaces>
class basic_figure_items;
template <class GraphicsSurfaces>
bool operator==(
    const typename basic_figure_items<GraphicsSurfaces>::abs_new_figure& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::abs_new_figure& rhs) noexcept;
template <class GraphicsSurfaces>
bool operator!=(
    const typename basic_figure_items<GraphicsSurfaces>::abs_new_figure& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::abs_new_figure& rhs) noexcept;
template <class GraphicsSurfaces>
bool operator==(
    const typename basic_figure_items<GraphicsSurfaces>::rel_new_figure& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::rel_new_figure& rhs) noexcept;
template <class GraphicsSurfaces>
bool operator!=(
    const typename basic_figure_items<GraphicsSurfaces>::rel_new_figure& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::rel_new_figure& rhs) noexcept;
template <class GraphicsSurfaces>
bool operator!=(
    const typename basic_figure_items<GraphicsSurfaces>::rel_new_figure& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::rel_new_figure& rhs)
noexcept;

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::close_figure& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::close_figure& rhs)
noexcept;

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::close_figure& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::close_figure& rhs)
noexcept;

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::abs_matrix& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::abs_matrix& rhs)
noexcept;

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::abs_matrix& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::abs_matrix& rhs)
noexcept;

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::rel_matrix& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::rel_matrix& rhs)
noexcept;

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::rel_matrix& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::rel_matrix& rhs)
noexcept;

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::revert_matrix& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::revert_matrix& rhs)
noexcept;

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::revert_matrix& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::revert_matrix& rhs)
noexcept;

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::abs_line& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::abs_line& rhs)
noexcept;

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::abs_line& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::abs_line& rhs)
noexcept;

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::rel_line& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::rel_line& rhs)
noexcept;

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::rel_line& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::rel_line& rhs)
noexcept;

Header <experimental/io2d> synopsis
template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve& lhs,
        const typename basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve& lhs,
        const typename basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve& lhs,
        const typename basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve& lhs,
        const typename basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::abs_cubic_curve& lhs,
        const typename basic_figure_items<GraphicsSurfaces>::abs_cubic_curve& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::abs_cubic_curve& lhs,
        const typename basic_figure_items<GraphicsSurfaces>::abs_cubic_curve& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::rel_cubic_curve& lhs,
        const typename basic_figure_items<GraphicsSurfaces>::rel_cubic_curve& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::rel_cubic_curve& lhs,
        const typename basic_figure_items<GraphicsSurfaces>::rel_cubic_curve& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::arc& lhs, const typename basic_figure_items<GraphicsSurfaces>::arc& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::arc& lhs, const typename basic_figure_items<GraphicsSurfaces>::arc& rhs) noexcept;

template <class GraphicsSurfaces>
class basic_fill_props;

template <class GraphicsSurfaces>
bool operator==(const basic_fill_props<GraphicsSurfaces>& lhs,
        const basic_fill_props<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator!=(const basic_fill_props<GraphicsSurfaces>& lhs,
        const basic_fill_props<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsSurfaces>
class basic_image_surface;

template <class GraphicsSurfaces>
class basic_interpreted_path;

Header <experimental/io2d> synopsis
template <class GraphicsSurfaces>
bool operator==(const basic_interpreted_path<GraphicsSurfaces>& lhs,
    const basic_interpreted_path<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator!=(const basic_interpreted_path<GraphicsSurfaces>& lhs,
    const basic_interpreted_path<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsSurfaces>
class basic_mask_props;
template <class GraphicsSurfaces>
bool operator==(const basic_mask_props<GraphicsSurfaces>& lhs,
    const basic_mask_props<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator!=(const basic_mask_props<GraphicsSurfaces>& lhs,
    const basic_mask_props<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsMath>
class basic_matrix_2d;
template <class GraphicsMath>
basic_matrix_2d<GraphicsMath> operator*(
    const basic_matrix_2d<GraphicsMath>& lhs,
    const basic_matrix_2d<GraphicsMath>& rhs) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator*(
    const basic_point_2d<GraphicsMath>& lhs,
    const basic_matrix_2d<GraphicsMath>& rhs) noexcept;

template <class GraphicsMath>
bool operator==(const basic_matrix_2d<GraphicsMath>& lhs,
    const basic_matrix_2d<GraphicsMath>& rhs) noexcept;

template <class GraphicsMath>
bool operator!=(const basic_matrix_2d<GraphicsMath>& lhs,
    const basic_matrix_2d<GraphicsMath>& rhs) noexcept;

template <class GraphicsSurfaces>
class basic_output_surface;

template <typename Allocator = typename allocator<typename basic_output_surface::figure_item>>
class basic_path_builder;
template <class GraphicsSurfaces, class Allocator = typename allocator<typename basic_output_surface::figure_item>>
bool operator==(const basic_path_builder<GraphicsSurfaces, Allocator>& lhs,
    const basic_path_builder<GraphicsSurfaces, Allocator>& rhs) noexcept;

template <class GraphicsSurfaces, class Allocator = typename allocator<typename basic_output_surface::figure_item>>
bool operator!=(const basic_path_builder<GraphicsSurfaces, Allocator>& lhs,
    const basic_path_builder<GraphicsSurfaces, Allocator>& rhs) noexcept;

template <class GraphicsSurfaces, class Allocator = typename allocator<typename basic_output_surface::figure_item>>
void swap(basic_path_builder<GraphicsSurfaces, Allocator>& lhs,
    basic_path_builder<GraphicsSurfaces, Allocator>& rhs) noexcept(noexcept(lhs.swap(rhs)));
const basic_point_2d<GraphicsMath>& rhs) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator-(
    const basic_point_2d<GraphicsMath>& val) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator-(
    const basic_point_2d<GraphicsMath>& lhs,
    const basic_point_2d<GraphicsMath>& rhs) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator*(
    const basic_point_2d<GraphicsMath>& lhs,
    float rhs) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator*(
    const basic_point_2d<GraphicsMath>& lhs,
    const basic_point_2d<GraphicsMath>& rhs) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator/(float lhs,
    const basic_point_2d<GraphicsMath>& rhs) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator/(float lhs,
    const basic_point_2d<GraphicsMath>& rhs) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator/(const basic_point_2d<GraphicsMath>& lhs,
    const basic_point_2d<GraphicsMath>& rhs) noexcept;

template <class GraphicsSurfaces>
class basic_render_props;

template <class GraphicsSurfaces>
bool operator==(const basic_render_props<GraphicsSurfaces>& lhs,
    const basic_render_props<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator!=(const basic_render_props<GraphicsSurfaces>& lhs,
    const basic_render_props<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsSurfaces>
class basic_stroke_props;

template <class GraphicsSurfaces>
bool operator==(const basic_stroke_props<GraphicsSurfaces>& lhs,
    const basic_stroke_props<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator!=(const basic_stroke_props<GraphicsSurfaces>& lhs,
    const basic_stroke_props<GraphicsSurfaces>& rhs) noexcept;

template <class GraphicsSurfaces>
class basic_unmanaged_output_surface;

using bounding_box = basic_bounding_box<default_graphics_math>;
using brush = basic_brush<default_graphics_surfaces>;
using brush_props = basic_brush_props<default_graphics_surfaces>;
using circle = basic_circle<default_graphics_math>;
using clip_props = basic_clip_props<default_graphics_surfaces>;
using dashes = basic_dashes<default_graphics_surfaces>;
using display_point = basic_display_point<default_graphics_math>;
using figure_items = basic_figure_items<default_graphics_surfaces>;
using fill_props = basic_fill_props<default_graphics_surfaces>;
using image_surface = basic_image_surface<default_graphics_surfaces>;
using interpreted_path = basic_interpreted_path<default_graphics_surfaces>;
using mask_props = basic_mask_props<default_graphics_surfaces>;

Header <experimental/io2d> synopsis
using matrix_2d = basic_matrix_2d<default_graphics_math>;
using output_surface = basic_output_surface<default_graphics_surfaces>;
using path_builder = basic_path_builder<default_graphics_surfaces>;
using point_2d = basic_point_2d<default_graphics_math>;
using render_props = basic_render_props<default_graphics_surfaces>;
using stroke_props = basic_stroke_props<default_graphics_surfaces>;
using unmanaged_output_surface = basic_unmanaged_output_surface<default_graphics_surfaces>;

template <class GraphicsSurfaces>
basic_image_surface<GraphicsSurfaces> copy_surface(basic_image_surface<GraphicsSurfaces>& sfc) noexcept;

template <class GraphicsSurfaces>
basic_image_surface<GraphicsSurfaces> copy_surface(basic_output_surface<GraphicsSurfaces>& sfc) noexcept;

template <class T>
constexpr T degrees_to_radians(T d) noexcept;

template <class T>
constexpr T radians_to_degrees(T r) noexcept;

float angle_for_point(point_2d ctr, point_2d pt) noexcept;
point_2d point_for_angle(float ang, float rad = 1.0f) noexcept;
point_2d point_for_angle(float ang, point_2d rad) noexcept;
point_2d arc_start(point_2d ctr, float sang, point_2d rad, const matrix_2d& m = matrix_2d()) noexcept;
point_2d arc_center(point_2d cpt, float sang, point_2d rad, const matrix_2d& m = matrix_2d()) noexcept;
point_2d arc_end(point_2d cpt, float eang, point_2d rad, const matrix_2d& m = matrix_2d()) noexcept;
6 Colors

6.1 Introduction to color

Color involves many disciplines and has been the subject of many papers, treatises, experiments, studies, and research work in general.

While color is an important part of computer graphics, it is only necessary to understand a few concepts from the study of color for computer graphics.

A color model defines color mathematically without regard to how humans actually perceive color. These color models are composed of some combination of channels which each channel representing alpha or an ideal color. Color models are useful for working with color computationally, such as in composing operations, because their channel values are homogeneously spaced.

A color space, for purposes of computer graphics, is the result of mapping the ideal color channels from a color model, after making any necessary adjustment for alpha, to color channels that are calibrated to align with human perception of colors. Since the perception of color varies from person to person, color spaces use the science of colorimetry to define those perceived colors in order to obtain uniformity to the extent possible. As such, the uniform display of the colors in a color space on different output devices is possible. The values of color channels in a color space are not necessarily homogeneously spaced because of human perception of color.

Color models are often termed linear while color spaces are often termed gamma corrected. The mapping of a color model, such as the RGB color model, to a color space, such as the sRGB color space, is often the application of gamma correction.

Gamma correction is the process of transforming homogeneously spaced visual data to visual data that, when displayed, matches the intent of the untransformed visual data.

For example a color that is 50% of the maximum intensity of red when encoded as homogeneously spaced visual data, will likely have a different intensity value when it has been gamma corrected so that a human looking at on a computer display will see it as being 50% of the maximum intensity of red that the computer display is capable of producing. Without gamma correction, it would likely have appeared as though it was closer to the maximum intensity than the untransformed data intended it to be.

In addition to color channels, colors in computer graphics often have an alpha channel. The value of the alpha channel represents transparency of the color channels when they are combined with other visual data using certain composing algorithms. When using alpha, it should be used in a premultiplied format in order to obtain the desired results when applying multiple composing algorithms that utilize alpha.

6.2 Color usage requirements

During rendering and composing operations, color data is linear and, when it has an alpha channel associated with it, in premultiplied format. Implementations shall make any necessary conversions to ensure this.

6.3 Class rgba_color

6.3.1 rgba_color overview

The class rgba_color describes a four channel color in premultiplied format.

There are three color channels, red, green, and blue, each of which is a float.

There is also an alpha channel, which is a float.

Legal values for each channel are in the range \([0.0f, 1.0f]\).

6.3.2 rgba_color synopsis

```cpp
namespace std::experimental::io2d::v1 {
    class rgba_color {
        public:
            // 6.3.3, construct/copy/move/destroy:
            constexpr rgba_color() noexcept;
```
template <class T>
constexpr rgba_color(T r, T g, T b, T a = static_cast<T>(0xFF)) noexcept;

template <class U>
constexpr rgba_color(U r, U g, U b, U a = static_cast<U>(1.0f)) noexcept;

// 6.3.4, modifiers:
template <class T>
constexpr void r(T val) noexcept;

template <class U>
constexpr void r(U val) noexcept;

template <class T>
constexpr void g(T val) noexcept;

template <class U>
constexpr void g(U val) noexcept;

template <class T>
constexpr void b(T val) noexcept;

template <class U>
constexpr void b(U val) noexcept;

template <class T>
constexpr void a(T val) noexcept;

template <class U>
constexpr void a(U val) noexcept;

// 6.3.5, observers:
constexpr float r() const noexcept;
constexpr float g() const noexcept;
constexpr float b() const noexcept;
constexpr float a() const noexcept;

// 6.3.6, static members:
static const rgba_color alice_blue;
static const rgba_color antique_white;
static const rgba_color aqua;
static const rgba_color aquamarine;
static const rgba_color azure;
static const rgba_color beige;
static const rgba_color bisque;
static const rgba_color black;
static const rgba_color blanched_almond;
static const rgba_color blue;
static const rgba_color blue_violet;
static const rgba_color brown;
static const rgba_color burly_wood;
static const rgba_color cadet_blue;
static const rgba_color chartreuse;
static const rgba_color chocolate;
static const rgba_color coral;
static const rgba_color cornflower_blue;
static const rgba_color cornsilk;
static const rgba_color crimson;
static const rgba_color cyan;
static const rgba_color dark_blue;
static const rgba_color dark_cyan;
static const rgba_color dark_goldenrod;
static const rgba_color dark_gray;
static const rgba_color dark_green;
static const rgba_color dark_grey;
static const rgba_color dark_honeydew;
static const rgba_color dark_magenta;
static const rgba_color dark_olive_green;
static const rgba_color dark_orange;
static const rgba_color dark_orchid;
static const rgba_color dark_red;
static const rgba_color dark_salmon;
static const rgba_color dark_sea_green;
static const rgba_color dark_slate_blue;
static const rgba_color dark_slate_gray;
static const rgba_color dark_slate_grey;
static const rgba_color dark_turquoise;
static const rgba_color dark_violet;
static const rgba_color deep_pink;
static const rgba_color deep_sky_blue;
static const rgba_color dim_gray;
static const rgba_color dim_grey;
static const rgba_color dodger_blue;
static const rgba_color firebrick;
static const rgba_color floral_white;
static const rgba_color forest_green;
static const rgba_color fuchsia;
static const rgba_color gainsboro;
static const rgba_color ghost_white;
static const rgba_color gold;
static const rgba_color goldenrod;
static const rgba_color gray;
static const rgba_color green;
static const rgba_color green_yellow;
static const rgba_color grey;
static const rgba_color honeydew;
static const rgba_color hot_pink;
static const rgba_color indian_red;
static const rgba_color indigo;
static const rgba_color ivory;
static const rgba_color khaki;
static const rgba_color lavender;
static const rgba_color lemon_chiffon;
static const rgba_color light_blue;
static const rgba_color light_coral;
static const rgba_color light_cyan;
static const rgba_color light_goldenrod_yellow;
static const rgba_color light_gray;
static const rgba_color light_green;
static const rgba_color light_pink;
static const rgba_color light_salmon;
static const rgba_color light_sea_green;
static const rgba_color light_sky_blue;
static const rgba_color light_slate_gray;
static const rgba_color light_slate_grey;
static const rgba_color light_steel_blue;
static const rgba_color light_yellow;
static const rgba_color lime;
static const rgba_color lime_green;
static const rgba_color linen;
static const rgba_color magenta;
static const rgba_color maroon;
static const rgba_color medium_aquamarine;
static const rgba_color medium_blue;
static const rgba_color medium_orchid;
static const rgba_color medium_purple;
static const rgba_color medium_sea_green;
static const rgba_color medium_slate_blue;
static const rgba_color medium_spring_green;
static const rgba_color medium_turquoise;
static const rgba_color medium_violet_red;
static const rgba_color midnight_blue;
static const rgba_color mint_cream;
static const rgba_color misty_rose;
static const rgba_color moccasin;
static const rgba_color navajo_white;
static const rgba_color navy;
static const rgba_color old_lace;
static const rgba_color olive;
static const rgba_color olive_drab;
static const rgba_color orange;
static const rgba_color orange_red;
static const rgba_color orchid;
static const rgba_color pale_goldenrod;
static const rgba_color pale_green;
static const rgba_color pale_turquoise;
static const rgba_color pale_violet_red;
static const rgba_color papaya_whip;
static const rgba_color peach_puff;
static const rgba_color peru;
static const rgba_color pink;
static const rgba_color plum;
static const rgba_color powder_blue;
static const rgba_color purple;
static const rgba_color red;
static const rgba_color rosy_brown;
static const rgba_color royal_blue;
static const rgba_color saddle_brown;
static const rgba_color salmon;
static const rgba_color sandy_brown;
static const rgba_color sea_green;
static const rgba_color sea_shell;
static const rgba_color sienna;
static const rgba_color silver;
static const rgba_color sky_blue;
static const rgba_color slate_blue;
static const rgba_color slate_gray;
static const rgba_color slate_grey;
static const rgba_color snow;
static const rgba_color spring_green;
static const rgba_color steel_blue;
static const rgba_color tan;
static const rgba_color teal;
static const rgba_color thistle;
static const rgba_color tomato;
static const rgba_color transparent_black;
static const rgba_color turquoise;
static const rgba_color violet;
static const rgba_color wheat;
static const rgba_color white;
static const rgba_color white_smoke;
static const rgba_color yellow;
static const rgba_color yellow_green;

// 6.3.7, operators
template <class T>
constexpr rgba_color& operator*=(T rhs) noexcept;
template <class U>
constexpr rgba_color& operator*=(U rhs) noexcept;

// 6.3.7, operators:
constexpr bool operator==(const rgba_color& lhs, const rgba_color& rhs) noexcept;
constexpr bool operator!=(const rgba_color& lhs, const rgba_color& rhs) noexcept;
template <class T>
constexpr rgba_color operator*(const rgba_color& lhs, T rhs) noexcept;

template <class U>
constexpr rgba_color operator*(const rgba_color& lhs, U rhs) noexcept;

template <class T>
constexpr rgba_color operator*(T lhs, const rgba_color& rhs) noexcept;

template <class U>
constexpr rgba_color operator*(U lhs, const rgba_color& rhs) noexcept;

6.3.3 rgba_color constructors and assignment operators

constexpr rgba_color() noexcept;

Effects: Equivalent to: rgba_color{ 0.0f, 0.0f, 0.0f, 0.0f }.

template <class T>
constexpr rgba_color(T r, T g, T b, T a = static_cast<T>(255)) noexcept;

Requires: r >= 0 and r <= 255 and g >= 0 and g <= 255 and b >= 0 and b <= 255 and a >= 0 and a <= 255.

Effects: Constructs an object of type rgba_color. The alpha channel is a / 255.0F. The red channel is r / 255.0F * a / 255.0F. The green channel is g / 255.0F * a / 255.0F. The blue channel is b / 255.0F * a / 255.0F.

Remarks: This constructor shall not participate in overload resolution unless is_integral_v<T> is true.

template <class U>
constexpr rgba_color(U r, U g, U b, U a = static_cast<U>(1.0f)) noexcept;

Requires: r >= 0.0f and r <= 1.0f and g >= 0.0f and g <= 1.0f and b >= 0.0f and b <= 1.0f and a >= 0.0f and a <= 1.0f.

Effects: Constructs an object of type rgba_color. The alpha channel is a. The red channel is r * a. The green channel is g * a. The blue channel is b * a.

Remarks: This constructor shall not participate in overload resolution unless is_floating_point_v<U> is true.

6.3.4 rgba_color modifiers

template <class T>
constexpr void r(T val) noexcept;

Requires: val >= 0 and val <= 255.

Effects: The red channel is val / 255.0F * a().

Remarks: This function shall not participate in overload resolution unless is_integral_v<T> is true.

template <class U>
constexpr void r(U val) noexcept;

Requires: val >= 0.0f and val <= 1.0f.

Effects: The red channel is val * a().

Remarks: This function shall not participate in overload resolution unless is_floating_point_v<U> is true.

template <class T>
constexpr void g(T val) noexcept;

Requires: val >= 0 and val <= 255.

Effects: The green channel is val / 255.0F * a().

Remarks: This function shall not participate in overload resolution unless is_integral_v<T> is true.
template <class U>
constexpr void g(U val) noexcept;

  Requires: val >= 0.0f and val <= 1.0f.
  Effects: The green channel is val * a().
  Remarks: This function shall not participate in overload resolution unless is_floating_point_v<U> is true.

template <class T>
constexpr void b(T val) noexcept;

  Requires: val >= 0 and val <= 255.
  Effects: The blue channel is val / 255.0F * a().
  Remarks: This function shall not participate in overload resolution unless is_integral_v<T> is true.

template <class U>
constexpr void b(U val) noexcept;

  Requires: val >= 0.0f and val <= 1.0f.
  Effects: The blue channel is val * a().
  Remarks: This function shall not participate in overload resolution unless is_floating_point_v<U> is true.

template <class T>
constexpr void a(T val) noexcept;

  Requires: val >= 0 and val <= 255.
  Effects: If a() == 0.0f the alpha channel is val / 255.0F, otherwise:
  1. The red channel is set to (r() / a()) * val / 255.0F;
  2. The green channel is set to (g() / a()) * val / 255.0F;
  3. The blue channel is set to (b() / a()) * val / 255.0F;
  4. The alpha channel is set to val / 255.0F.
  Remarks: This function shall not participate in overload resolution unless is_integral_v<T> is true.

template <class U>
constexpr void a(U val) noexcept;

  Requires: val >= 0.0f and val <= 1.0f.
  Effects: If a() == 0.0f the alpha channel is val, otherwise:
  1. The red channel is set to (r() / a()) * val;
  2. The green channel is set to (g() / a()) * val;
  3. The blue channel is set to (b() / a()) * val;
  4. The alpha channel is val.
  Remarks: This function shall not participate in overload resolution unless is_floating_point_v<U> is true.

§ 6.3.5  rgba_color observers

constexpr float r() const noexcept;
  Returns: The red channel.

constexpr float g() const noexcept;
  Returns: The green channel.

constexpr float b() const noexcept;
  Returns: The blue channel.
constexpr float a() const noexcept;

Returns: The alpha channel.

6.3.6  rgba_color static members

The alpha value of all of the predefined rgba_color static member objects in Table 2 is 1.0F except for transparent_black, which has an alpha value of 0.0F.

Table 2 — rgba_color static members values

<table>
<thead>
<tr>
<th>Member name</th>
<th>red</th>
<th>green</th>
<th>blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice_blue</td>
<td>240</td>
<td>248</td>
<td>255</td>
</tr>
<tr>
<td>antique_white</td>
<td>250</td>
<td>235</td>
<td>215</td>
</tr>
<tr>
<td>aqua</td>
<td>0</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>aquamarine</td>
<td>127</td>
<td>255</td>
<td>212</td>
</tr>
<tr>
<td>azure</td>
<td>240</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>beige</td>
<td>245</td>
<td>245</td>
<td>220</td>
</tr>
<tr>
<td>bisque</td>
<td>255</td>
<td>228</td>
<td>196</td>
</tr>
<tr>
<td>black</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>blanched_almond</td>
<td>255</td>
<td>235</td>
<td>205</td>
</tr>
<tr>
<td>blue</td>
<td>0</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>blue_violet</td>
<td>138</td>
<td>43</td>
<td>226</td>
</tr>
<tr>
<td>brown</td>
<td>165</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>burly_wood</td>
<td>222</td>
<td>184</td>
<td>135</td>
</tr>
<tr>
<td>cadet_blue</td>
<td>95</td>
<td>158</td>
<td>160</td>
</tr>
<tr>
<td>chartreuse</td>
<td>127</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>chocolate</td>
<td>210</td>
<td>105</td>
<td>30</td>
</tr>
<tr>
<td>coral</td>
<td>255</td>
<td>127</td>
<td>80</td>
</tr>
<tr>
<td>cornflower_blue</td>
<td>100</td>
<td>149</td>
<td>237</td>
</tr>
<tr>
<td>cornsilk</td>
<td>255</td>
<td>248</td>
<td>220</td>
</tr>
<tr>
<td>crimson</td>
<td>220</td>
<td>20</td>
<td>60</td>
</tr>
<tr>
<td>cyan</td>
<td>0</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>dark_blue</td>
<td>0</td>
<td>0</td>
<td>139</td>
</tr>
<tr>
<td>dark_cyan</td>
<td>0</td>
<td>139</td>
<td>139</td>
</tr>
<tr>
<td>dark_goldenrod</td>
<td>184</td>
<td>134</td>
<td>11</td>
</tr>
<tr>
<td>dark_gray</td>
<td>169</td>
<td>169</td>
<td>169</td>
</tr>
<tr>
<td>dark_green</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>darkGrey</td>
<td>169</td>
<td>169</td>
<td>169</td>
</tr>
<tr>
<td>dark_khaki</td>
<td>189</td>
<td>183</td>
<td>107</td>
</tr>
<tr>
<td>dark_magenta</td>
<td>139</td>
<td>0</td>
<td>139</td>
</tr>
<tr>
<td>dark_olive_green</td>
<td>85</td>
<td>107</td>
<td>47</td>
</tr>
<tr>
<td>dark_orange</td>
<td>255</td>
<td>140</td>
<td>0</td>
</tr>
<tr>
<td>dark_orchid</td>
<td>153</td>
<td>50</td>
<td>204</td>
</tr>
<tr>
<td>dark_red</td>
<td>139</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>dark_salmon</td>
<td>233</td>
<td>150</td>
<td>122</td>
</tr>
<tr>
<td>dark_sea_green</td>
<td>143</td>
<td>188</td>
<td>142</td>
</tr>
<tr>
<td>dark_slate_blue</td>
<td>72</td>
<td>61</td>
<td>139</td>
</tr>
<tr>
<td>dark_slate_gray</td>
<td>47</td>
<td>79</td>
<td>79</td>
</tr>
<tr>
<td>dark_slate_grey</td>
<td>47</td>
<td>79</td>
<td>79</td>
</tr>
<tr>
<td>dark_turquoise</td>
<td>0</td>
<td>206</td>
<td>209</td>
</tr>
<tr>
<td>dark_violet</td>
<td>148</td>
<td>0</td>
<td>211</td>
</tr>
<tr>
<td>deep_pink</td>
<td>255</td>
<td>20</td>
<td>147</td>
</tr>
<tr>
<td>deep_sky_blue</td>
<td>0</td>
<td>191</td>
<td>255</td>
</tr>
<tr>
<td>dim_gray</td>
<td>105</td>
<td>105</td>
<td>105</td>
</tr>
<tr>
<td>dim_gray</td>
<td>105</td>
<td>105</td>
<td>105</td>
</tr>
<tr>
<td>dodger_blue</td>
<td>30</td>
<td>144</td>
<td>255</td>
</tr>
<tr>
<td>firebrick</td>
<td>178</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>floral_white</td>
<td>255</td>
<td>250</td>
<td>240</td>
</tr>
<tr>
<td>forest_green</td>
<td>34</td>
<td>139</td>
<td>34</td>
</tr>
</tbody>
</table>
Table 2 — rgba_color static members values (continued)

<table>
<thead>
<tr>
<th>Member name</th>
<th>red</th>
<th>green</th>
<th>blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>fuchsia</td>
<td>255</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>gainsboro</td>
<td>220</td>
<td>220</td>
<td>220</td>
</tr>
<tr>
<td>ghost_white</td>
<td>248</td>
<td>248</td>
<td>248</td>
</tr>
<tr>
<td>gold</td>
<td>255</td>
<td>215</td>
<td>0</td>
</tr>
<tr>
<td>goldenrod</td>
<td>218</td>
<td>165</td>
<td>32</td>
</tr>
<tr>
<td>gray</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>green</td>
<td>0</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>green_yellow</td>
<td>173</td>
<td>255</td>
<td>47</td>
</tr>
<tr>
<td>grey</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>honeydew</td>
<td>240</td>
<td>255</td>
<td>240</td>
</tr>
<tr>
<td>hot_pink</td>
<td>255</td>
<td>105</td>
<td>180</td>
</tr>
<tr>
<td>indian_red</td>
<td>205</td>
<td>92</td>
<td>92</td>
</tr>
<tr>
<td>indigo</td>
<td>75</td>
<td>0</td>
<td>130</td>
</tr>
<tr>
<td>ivory</td>
<td>255</td>
<td>255</td>
<td>240</td>
</tr>
<tr>
<td>khaki</td>
<td>240</td>
<td>230</td>
<td>140</td>
</tr>
<tr>
<td>lavender</td>
<td>230</td>
<td>230</td>
<td>250</td>
</tr>
<tr>
<td>lavender_blush</td>
<td>255</td>
<td>240</td>
<td>245</td>
</tr>
<tr>
<td>lawn_green</td>
<td>124</td>
<td>252</td>
<td>0</td>
</tr>
<tr>
<td>lemon_chiffon</td>
<td>255</td>
<td>250</td>
<td>205</td>
</tr>
<tr>
<td>light_blue</td>
<td>173</td>
<td>216</td>
<td>230</td>
</tr>
<tr>
<td>light_coral</td>
<td>240</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>light_cyan</td>
<td>224</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>light_goldenrod_yellow</td>
<td>250</td>
<td>250</td>
<td>210</td>
</tr>
<tr>
<td>light_gray</td>
<td>211</td>
<td>211</td>
<td>211</td>
</tr>
<tr>
<td>light_green</td>
<td>144</td>
<td>238</td>
<td>144</td>
</tr>
<tr>
<td>light_grey</td>
<td>211</td>
<td>211</td>
<td>211</td>
</tr>
<tr>
<td>light_pink</td>
<td>255</td>
<td>182</td>
<td>193</td>
</tr>
<tr>
<td>light_salmon</td>
<td>255</td>
<td>160</td>
<td>122</td>
</tr>
<tr>
<td>light_sea_green</td>
<td>32</td>
<td>178</td>
<td>170</td>
</tr>
<tr>
<td>light_sky_blue</td>
<td>135</td>
<td>206</td>
<td>250</td>
</tr>
<tr>
<td>light_slate_gray</td>
<td>119</td>
<td>136</td>
<td>153</td>
</tr>
<tr>
<td>light_slate-grey</td>
<td>119</td>
<td>136</td>
<td>153</td>
</tr>
<tr>
<td>light_steel_blue</td>
<td>176</td>
<td>196</td>
<td>222</td>
</tr>
<tr>
<td>light_yellow</td>
<td>255</td>
<td>255</td>
<td>224</td>
</tr>
<tr>
<td>lime</td>
<td>0</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>lime_green</td>
<td>50</td>
<td>205</td>
<td>50</td>
</tr>
<tr>
<td>linen</td>
<td>250</td>
<td>240</td>
<td>230</td>
</tr>
<tr>
<td>magenta</td>
<td>255</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>maroon</td>
<td>128</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>medium_aquamarine</td>
<td>102</td>
<td>205</td>
<td>170</td>
</tr>
<tr>
<td>medium_blue</td>
<td>0</td>
<td>0</td>
<td>205</td>
</tr>
<tr>
<td>medium_orchid</td>
<td>186</td>
<td>85</td>
<td>211</td>
</tr>
<tr>
<td>medium_purple</td>
<td>147</td>
<td>112</td>
<td>219</td>
</tr>
<tr>
<td>medium_sea_green</td>
<td>60</td>
<td>179</td>
<td>113</td>
</tr>
<tr>
<td>medium_slate_blue</td>
<td>123</td>
<td>104</td>
<td>238</td>
</tr>
<tr>
<td>medium_slate_blue</td>
<td>0</td>
<td>250</td>
<td>154</td>
</tr>
<tr>
<td>medium_turquoise</td>
<td>72</td>
<td>209</td>
<td>204</td>
</tr>
<tr>
<td>medium_violet_red</td>
<td>199</td>
<td>21</td>
<td>133</td>
</tr>
<tr>
<td>midnight_blue</td>
<td>25</td>
<td>25</td>
<td>112</td>
</tr>
<tr>
<td>mint_cream</td>
<td>245</td>
<td>255</td>
<td>250</td>
</tr>
<tr>
<td>misty_rose</td>
<td>255</td>
<td>228</td>
<td>225</td>
</tr>
<tr>
<td>moccasin</td>
<td>255</td>
<td>228</td>
<td>181</td>
</tr>
<tr>
<td>navajo_white</td>
<td>255</td>
<td>222</td>
<td>173</td>
</tr>
<tr>
<td>navy</td>
<td>0</td>
<td>0</td>
<td>128</td>
</tr>
<tr>
<td>old_lace</td>
<td>253</td>
<td>245</td>
<td>230</td>
</tr>
</tbody>
</table>
Table 2 — rgba_color static members values (continued)

<table>
<thead>
<tr>
<th>Member name</th>
<th>red</th>
<th>green</th>
<th>blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>olive</td>
<td>128</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>olive_drab</td>
<td>107</td>
<td>142</td>
<td>35</td>
</tr>
<tr>
<td>orange</td>
<td>255</td>
<td>69</td>
<td>0</td>
</tr>
<tr>
<td>orange_red</td>
<td>255</td>
<td>69</td>
<td>0</td>
</tr>
<tr>
<td>orchid</td>
<td>218</td>
<td>112</td>
<td>214</td>
</tr>
<tr>
<td>pale_goldenrod</td>
<td>238</td>
<td>232</td>
<td>170</td>
</tr>
<tr>
<td>pale_green</td>
<td>152</td>
<td>251</td>
<td>152</td>
</tr>
<tr>
<td>pale_turquoise</td>
<td>175</td>
<td>238</td>
<td>238</td>
</tr>
<tr>
<td>pale_violet_red</td>
<td>219</td>
<td>112</td>
<td>147</td>
</tr>
<tr>
<td>papaya whip</td>
<td>255</td>
<td>239</td>
<td>213</td>
</tr>
<tr>
<td>peach_puff</td>
<td>255</td>
<td>218</td>
<td>185</td>
</tr>
<tr>
<td>peru</td>
<td>205</td>
<td>133</td>
<td>63</td>
</tr>
<tr>
<td>pink</td>
<td>255</td>
<td>192</td>
<td>203</td>
</tr>
<tr>
<td>plum</td>
<td>221</td>
<td>160</td>
<td>221</td>
</tr>
<tr>
<td>powder_blue</td>
<td>176</td>
<td>224</td>
<td>230</td>
</tr>
<tr>
<td>purple</td>
<td>128</td>
<td>0</td>
<td>128</td>
</tr>
<tr>
<td>red</td>
<td>255</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rosy_brown</td>
<td>188</td>
<td>143</td>
<td>143</td>
</tr>
<tr>
<td>royal_blue</td>
<td>65</td>
<td>105</td>
<td>225</td>
</tr>
<tr>
<td>saddle_brown</td>
<td>139</td>
<td>69</td>
<td>19</td>
</tr>
<tr>
<td>salmon</td>
<td>250</td>
<td>128</td>
<td>114</td>
</tr>
<tr>
<td>sandy_brown</td>
<td>244</td>
<td>164</td>
<td>96</td>
</tr>
<tr>
<td>sea_green</td>
<td>46</td>
<td>139</td>
<td>87</td>
</tr>
<tr>
<td>sea_shell</td>
<td>255</td>
<td>245</td>
<td>238</td>
</tr>
<tr>
<td>sienna</td>
<td>160</td>
<td>82</td>
<td>45</td>
</tr>
<tr>
<td>silver</td>
<td>192</td>
<td>192</td>
<td>192</td>
</tr>
<tr>
<td>sky_blue</td>
<td>135</td>
<td>206</td>
<td>235</td>
</tr>
<tr>
<td>slate_blue</td>
<td>106</td>
<td>90</td>
<td>205</td>
</tr>
<tr>
<td>slate_gray</td>
<td>112</td>
<td>128</td>
<td>144</td>
</tr>
<tr>
<td>slate_grey</td>
<td>112</td>
<td>128</td>
<td>144</td>
</tr>
<tr>
<td>snow</td>
<td>255</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>spring_green</td>
<td>0</td>
<td>255</td>
<td>127</td>
</tr>
<tr>
<td>steel_blue</td>
<td>70</td>
<td>130</td>
<td>180</td>
</tr>
<tr>
<td>tan</td>
<td>210</td>
<td>180</td>
<td>140</td>
</tr>
<tr>
<td>teal</td>
<td>0</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>thistle</td>
<td>216</td>
<td>191</td>
<td>216</td>
</tr>
<tr>
<td>tomato</td>
<td>255</td>
<td>99</td>
<td>71</td>
</tr>
<tr>
<td>transparent_black</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>turquoise</td>
<td>64</td>
<td>244</td>
<td>208</td>
</tr>
<tr>
<td>violet</td>
<td>238</td>
<td>130</td>
<td>238</td>
</tr>
<tr>
<td>wheat</td>
<td>245</td>
<td>222</td>
<td>179</td>
</tr>
<tr>
<td>white</td>
<td>255</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>white_smoke</td>
<td>245</td>
<td>245</td>
<td>245</td>
</tr>
<tr>
<td>yellow</td>
<td>255</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>yellow_green</td>
<td>154</td>
<td>205</td>
<td>50</td>
</tr>
</tbody>
</table>

6.3.7  rgba_color operators

```cpp
template <class T>
constexpr rgba_color& operator*=(T rhs) noexcept;
```

1. Requires: rhs >= 0 and rhs <= 255.
2. Effects: r(min(r()) * rhs / 255.0F, 1.0F)).
3. g(min(g()) * rhs / 255.0F, 1.0F)).
4. b(min(b()) * rhs / 255.0F, 1.0F)).

§ 6.3.7
a(min(a() \* rhs / 255.0F, 1.0F)).

Returns: *this.

Remarks: This function shall not participate in overload resolution unless is_integral_v<T> is true.

template <class U>
constexpr rgba_color& operator*=(U rhs) noexcept;

Requires: rhs >= 0.0F and rhs <= 1.0F.
Effects: r(min(r() \* rhs, 1.0F)).
g(min(g() \* rhs, 1.0F)).
b(min(b() \* rhs, 1.0F)).
a(min(a() \* rhs, 1.0F)).

Returns: *this.

Remarks: This function shall not participate in overload resolution unless is_floating_point_v<T> is true.

constexpr bool operator==(const rgba_color& lhs, const rgba_color& rhs) noexcept;

Returns: lhs.r() == rhs.r() && lhs.g() == rhs.g() && lhs.b() == rhs.b() && lhs.a() == rhs.a().

template <class T>
constexpr rgba_color operator*(const rgba_color& lhs, T rhs) noexcept;

Requires: rhs >= 0 and rhs <= 255.
Returns:
rgba_color(min(lhs.r() \* rhs / 255.0F, 1.0F), min(lhs.g() \* rhs / 255.0F, 1.0F),
min(lhs.b() \* rhs / 255.0F, 1.0F), min(lhs.a() \* rhs / 255.0F, 1.0F))

Remarks: This function shall not participate in overload resolution unless is_integral_v<T> is true.

template <class U>
constexpr rgba_color& operator*(const rgba_color& lhs, U rhs) noexcept;

Requires: rhs >= 0.0F and rhs <= 1.0F.
Returns:
rgba_color(min(lhs.r() \* rhs, 1.0F), min(lhs.g() \* rhs, 1.0F),
min(lhs.b() \* rhs, 1.0F), min(lhs.a() \* rhs, 1.0F))

Remarks: This function shall not participate in overload resolution unless is_floating_point_v<U> is true.

template <class T>
constexpr rgba_color operator*(T lhs, const rgba_color& rhs) noexcept;

Requires: lhs >= 0 and lhs <= 255.
Returns:
rgba_color(min(lhs * rhs.r() / 255.0F, 1.0F), min(lhs * rhs.g() / 255.0F, 1.0F),
min(lhs * rhs.b() / 255.0F, 1.0F), min(lhs * rhs.a() / 255.0F, 1.0F))

Remarks: This function shall not participate in overload resolution unless is_integral_v<T> is true.

template <class U>
constexpr rgba_color& operator*(U lhs, const rgba_color& rhs) noexcept;

Requires: lhs >= 0.0F and lhs <= 1.0F.
Returns:
rgba_color(min(lhs * rhs.r(), 1.0F), min(lhs * rhs.g(), 1.0F),
min(lhs * rhs.b(), 1.0F), min(lhs * rhs.a(), 1.0F))

Remarks: This function shall not participate in overload resolution unless is_floating_point_v<U> is true.
7 Graphics math

7.1 General

This Clause describes components that are used to describe certain geometric types and to perform certain linear algebra operations. [Note: These types are intended for use in 2D graphics input/output operations. They are not meant to provide a full set of linear algebra types and operations. — end note]

The following subclauses describe graphics math requirements and components for linear algebra classes and geometry classes, as summarized in Table 3.

Table 3 — Graphics math summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td></td>
</tr>
<tr>
<td>Clause 9</td>
<td>&lt;experimental/io2d&gt;</td>
</tr>
<tr>
<td>Clause 10</td>
<td>&lt;experimental/io2d&gt;</td>
</tr>
</tbody>
</table>

7.2 Requirements

This subsection defines requirements on GraphicsMath types.

Most classes specified in Clause 9 through Clause 15 need a set of related types and functions to complete the definition of their semantics. These types and functions are provided as a set of member typedef-names and static member functions in the template parameter GraphicsMath used by each such template. This subclause defines the semantics of these members.

Let X be a GraphicsMath type.

Table 8 defines required typedef-names in X, which are identifiers for class types capable of storing all data required to support the corresponding class template.

Table 4 — X typedef-names

<table>
<thead>
<tr>
<th>typedef-name</th>
<th>Class data</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounding_box_data_type</td>
<td>basic_bounding_box&lt;X&gt;</td>
</tr>
<tr>
<td>circle_data_type</td>
<td>basic_circle&lt;X&gt;</td>
</tr>
<tr>
<td>display_point_data_type</td>
<td>basic_display_point&lt;X&gt;</td>
</tr>
<tr>
<td>matrix_2d_data_type</td>
<td>basic_matrix_2d&lt;X&gt;</td>
</tr>
<tr>
<td>point_2d_data_type</td>
<td>basic_point_2d&lt;X&gt;</td>
</tr>
</tbody>
</table>

In order to describe the observable effects of functions contained in Table 6, Table 5 describes the types contained in X as if they possessed certain member data.

Table 5 — GraphicsMath type member data

<table>
<thead>
<tr>
<th>Type</th>
<th>Member data</th>
<th>Member data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounding_box_data_type</td>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>bounding_box_data_type</td>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>bounding_box_data_type</td>
<td>w</td>
<td>float</td>
</tr>
<tr>
<td>bounding_box_data_type</td>
<td>h</td>
<td>float</td>
</tr>
<tr>
<td>circle_data_type</td>
<td>x</td>
<td>float</td>
</tr>
<tr>
<td>circle_data_type</td>
<td>y</td>
<td>float</td>
</tr>
<tr>
<td>circle_data_type</td>
<td>r</td>
<td>float</td>
</tr>
<tr>
<td>display_point_data_type</td>
<td>x</td>
<td>int</td>
</tr>
<tr>
<td>display_point_data_type</td>
<td>y</td>
<td>int</td>
</tr>
</tbody>
</table>
In Table 6, B denotes the type `X::bounding_box_data_type`, C denotes the type `X::circle_data_type`, D denotes the type `X::display_point_data_type`, M denotes the type `X::matrix_2d_data_type`, and P denotes the type `X::point_2d_data_type`.

All expressions in Table 6 are `noexcept`. For purposes of brevity, `noexcept` is omitted in the table.

Table 6 — GraphicsMath requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::create_point_2d()</code></td>
<td><code>P</code></td>
<td>Equivalent to: <code>create_point_2d(0.0f, 0.0f)</code></td>
<td></td>
</tr>
<tr>
<td><code>X::create_point_2d(float x, float y)</code></td>
<td><code>P</code></td>
<td>Returns an object <code>p</code>.</td>
<td>Postconditions: <code>p.x == x</code> and <code>p.y == y</code>.</td>
</tr>
<tr>
<td><code>X::x(P&amp; p, float x)</code></td>
<td><code>void</code></td>
<td>Returns <code>p.x</code>.</td>
<td>Postconditions: <code>p.x == x</code>.</td>
</tr>
<tr>
<td><code>X::y(P&amp; p, float y)</code></td>
<td><code>void</code></td>
<td>Returns <code>p.y</code>.</td>
<td>Postconditions: <code>p.y == y</code>.</td>
</tr>
<tr>
<td><code>X::x(const P&amp; p)</code></td>
<td><code>float</code></td>
<td>Returns <code>p.x</code>.</td>
<td></td>
</tr>
<tr>
<td><code>X::y(const P&amp; p)</code></td>
<td><code>float</code></td>
<td>Returns <code>p.y</code>.</td>
<td></td>
</tr>
<tr>
<td><code>X::dot(const P&amp; p1, const P&amp; p2)</code></td>
<td><code>float</code></td>
<td>Returns <code>p1.x * p2.x + p1.y * p2.y</code>.</td>
<td></td>
</tr>
<tr>
<td><code>X::magnitude(const P&amp; p)</code></td>
<td><code>float</code></td>
<td>Returns <code>sqrt(p.x * p.x + p.y * p.y)</code>.</td>
<td></td>
</tr>
<tr>
<td><code>X::magnitude_squared(const P&amp; p)</code></td>
<td><code>float</code></td>
<td>Returns <code>p.x * p.x + p.y * p.y</code>.</td>
<td></td>
</tr>
<tr>
<td><code>X::angular_direction(const P&amp; p)</code></td>
<td><code>float</code></td>
<td>Returns <code>atan2(p.y, p.x) &lt; 0.0f ? atan2(p.y, p.x) + two_pi&lt;float&gt; : atan2(p.y, p.x)</code>.</td>
<td>Remarks: The purpose of adding <code>two_pi&lt;float&gt;</code> if the result is negative is to produce values in the range <code>[0.0f, two_pi&lt;float&gt;]</code>.</td>
</tr>
<tr>
<td><code>X::to_unit(const P&amp; p)</code></td>
<td><code>P</code></td>
<td>Returns an object <code>r</code>.</td>
<td>Postconditions: <code>r.x == p.x / magnitude(p)</code> and <code>r.y == p.y / magnitude(p)</code>.</td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Operational semantics</td>
<td>Assertion/note</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
<td>-------------</td>
<td>-----------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>X::add(const P&amp; p1, const P&amp; p2)</td>
<td>P</td>
<td>Returns an object r.</td>
<td></td>
</tr>
<tr>
<td>X::add(const P&amp; p, float f)</td>
<td>P</td>
<td>Returns an object r.</td>
<td></td>
</tr>
<tr>
<td>X::add(float f, const P&amp; p)</td>
<td>P</td>
<td>Returns an object r.</td>
<td></td>
</tr>
<tr>
<td>X::subtract(const P&amp; p1, const P&amp; p2)</td>
<td>P</td>
<td>Returns an object r.</td>
<td></td>
</tr>
<tr>
<td>X::subtract(const P&amp; p, float f)</td>
<td>P</td>
<td>Returns an object r.</td>
<td></td>
</tr>
<tr>
<td>X::subtract(float f, const P&amp; p)</td>
<td>P</td>
<td>Returns an object r.</td>
<td></td>
</tr>
<tr>
<td>X::multiply(const P&amp; p1, const P&amp; p2)</td>
<td>P</td>
<td>Returns an object r.</td>
<td></td>
</tr>
<tr>
<td>X::multiply(const P&amp; p, float f)</td>
<td>P</td>
<td>Returns an object r.</td>
<td></td>
</tr>
<tr>
<td>X::multiply(float f, const P&amp; p)</td>
<td>P</td>
<td>Returns an object r.</td>
<td></td>
</tr>
<tr>
<td>X::divide(const P&amp; p1, const P&amp; p2)</td>
<td>P</td>
<td>Returns an object r.</td>
<td></td>
</tr>
<tr>
<td>X::divide(const P&amp; p, float f)</td>
<td>P</td>
<td>Returns an object r.</td>
<td></td>
</tr>
<tr>
<td>X::divide(float f, const P&amp; p)</td>
<td>P</td>
<td>Returns an object r.</td>
<td></td>
</tr>
<tr>
<td>X::equal(const P&amp; l, bool const P&amp; r)</td>
<td></td>
<td>Returns l.x == r.x &amp;&amp; l.y == r.y</td>
<td></td>
</tr>
</tbody>
</table>

§ 7.2
Table 6 — GraphicsMath requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::not_equal(const P&amp; l, const P&amp; r)</td>
<td>bool</td>
<td>Equivalent to return !equal(1, r);</td>
<td></td>
</tr>
<tr>
<td>X::negate(const P&amp; p)</td>
<td>P</td>
<td>Returns r, where r.x == -p.x and r.y == -p.y</td>
<td></td>
</tr>
<tr>
<td>X::create_matrix_2d()</td>
<td>M</td>
<td>Equivalent to return create_matrix_2d(1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f);</td>
<td></td>
</tr>
<tr>
<td>X::create_matrix_2d(float m00, float m01, float m10, float m11, float m20, float m21)</td>
<td>M</td>
<td>Returns an object m Postconditions: m.m00 == m00, m.m01 == m01, m.m02 == 0.0f, m.m10 == m10, m.m11 == m11, m.m12 == 0.0f, m.m20 == m20, m.m21 == m21, m.m22 == 1.0f</td>
<td></td>
</tr>
<tr>
<td>X::create_translate(const P&amp; p)</td>
<td>M</td>
<td>Equivalent to return X::create_matrix_2d(1.0f, 0.0f, 0.0f, 1.0f, p.x, p.y);</td>
<td></td>
</tr>
<tr>
<td>X::create_scale(const P&amp; p)</td>
<td>M</td>
<td>Equivalent to return X::create_matrix_2d(p.x, 0.0f, 0.0f, p.y, 0.0f, 0.0f);</td>
<td></td>
</tr>
<tr>
<td>X::create_rotate(float r)</td>
<td>M</td>
<td>Equivalent to return X::create_matrix_2d(cos(r), sin(r), sin(r), -cos(r), 0.0f, 0.0f);</td>
<td></td>
</tr>
<tr>
<td>X::create_rotate(float r, const P&amp; p)</td>
<td>M</td>
<td>Equivalent to return multiply(multiply(create_translate(p), create_rotate(r)), create_translate(negate(p)));</td>
<td></td>
</tr>
<tr>
<td>X::create_reflect(float r)</td>
<td>M</td>
<td>Equivalent to return X::create_matrix_2d(cos(r * 2.0f), sin(r * 2.0f), sin(r * 2.0f), -cos(r * 2.0f), 0.0f, 0.0f);</td>
<td></td>
</tr>
<tr>
<td>X::create_shear_x(float f)</td>
<td>M</td>
<td>Equivalent to return create_matrix_2d(1.0f, 0.0f, f, 1.0f, 0.0f, 0.0f);</td>
<td></td>
</tr>
<tr>
<td>X::create_shear_y(float f)</td>
<td>M</td>
<td>Equivalent to return create_matrix_2d(1.0f, f, 0.0f, 1.0f, 0.0f, 0.0f);</td>
<td></td>
</tr>
</tbody>
</table>
Table 6 — GraphicsMath requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::multiply(const M&amp; l, const M&amp; r)</td>
<td>M</td>
<td>Equivalent to return create_matrix_2d(l.m00 * r.m00 + l.m10 * r.m10, l.m00 * r.m01 + l.m01 * r.m10, l.m10 * r.m00 + l.m10 * r.m01, l.m01 * r.m00 + l.m01 * r.m10, l.m20 * r.m00 + l.m20 * r.m10, l.m20 * r.m01 + l.m20 * r.m1)</td>
<td></td>
</tr>
<tr>
<td>X::translate(M&amp; m, const P&amp; p)</td>
<td>void</td>
<td>Equivalent to m = multiply(m, create_translate(p));</td>
<td></td>
</tr>
<tr>
<td>X::scale(M&amp; m, const P&amp; p)</td>
<td>void</td>
<td>Equivalent to m = multiply(m, create_scale(p));</td>
<td></td>
</tr>
<tr>
<td>X::rotate(M&amp; m, float r)</td>
<td>void</td>
<td>Equivalent to m = multiply(m, create_rotate(r));</td>
<td></td>
</tr>
<tr>
<td>X::rotate(M&amp; m, float r, const P&amp; p)</td>
<td>void</td>
<td>Equivalent to m = multiply(m, create_rotate(r, p));</td>
<td></td>
</tr>
<tr>
<td>X::reflect(M&amp; m, float r)</td>
<td>void</td>
<td>Equivalent to m = multiply(m, create_rotate(r));</td>
<td></td>
</tr>
<tr>
<td>X::shear_x(M&amp; m, float f)</td>
<td>void</td>
<td>Equivalent to m = multiply(m, create_shear_x(f));</td>
<td></td>
</tr>
<tr>
<td>X::shear_y(M&amp; m, float f)</td>
<td>void</td>
<td>Equivalent to m = multiply(m, create_shear_y(f));</td>
<td></td>
</tr>
<tr>
<td>X::is_finite(const M&amp; m)</td>
<td>bool</td>
<td>Equivalent to return isfinite(m.m00) &amp;&amp; isfinite(m.m01) &amp;&amp; isfinite(m.m10) &amp;&amp; isfinite(m.m11) &amp;&amp; isfinite(m.m20) &amp;&amp; isfinite(m.m21);</td>
<td></td>
</tr>
<tr>
<td>X::is_invertible(const M&amp; m)</td>
<td>bool</td>
<td>Equivalent to return (m.m00 * m.m11 - m.m01 * m.m10) != 0.0f;</td>
<td></td>
</tr>
<tr>
<td>X::determinant(const M&amp; m)</td>
<td>float</td>
<td>Equivalent to return m.m00 * m.m11 - m.m01 * m.m10;</td>
<td></td>
</tr>
<tr>
<td>X::inverse(const M&amp; m)</td>
<td>M</td>
<td>Equivalent to: float id = 1.0f / determinant(m); return create_matrix_2d((m.m11 * 1.0f - 0.0f * m.m21) * id, -(m.m01 * 1.0f - 0.0f * m.m21) * id, (m.m00 * 1.0f - 0.0f * m.m20) * id, -(m.m00 * 1.0f - 0.0f * m.m20) * id, (m.m00 * 1.0f - 0.0f * m.m20) * id, -(m.m00 * 1.0f - 0.0f * m.m20) * id)</td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Operational semantics</td>
<td>Assertion/note pre-/post-condition</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-----------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>X::transform_-pt(const M&amp; m, const P&amp; p)</td>
<td>P</td>
<td>return create_point_2d(m.m00 * p.x + m10 * p.y + m.m20, m.01 * p.x + m.m11 * p.y + m.21);</td>
<td></td>
</tr>
<tr>
<td>X::equal(const M&amp; l, bool const M&amp; r)</td>
<td>bool</td>
<td>Returns l.m00 == r.m00 &amp;&amp; l.m01 == r.m01 &amp;&amp; l.m11 == r.m11 &amp;&amp; l.m20 == r.m20 &amp;&amp; l.m21 == r.m21.</td>
<td></td>
</tr>
<tr>
<td>X::not_equal(const M&amp; l, const M&amp; r)</td>
<td>bool</td>
<td>Equivalent to return !equal(l, r);</td>
<td></td>
</tr>
<tr>
<td>create_display_-point()</td>
<td>D</td>
<td>Equivalent to return create_display_point(0, 0);</td>
<td></td>
</tr>
<tr>
<td>create_display_-point(int x, int y)</td>
<td>D</td>
<td>Returns an object d.</td>
<td>Postconditions: d.x == x and d.y == y.</td>
</tr>
<tr>
<td>X::x(D&amp; d, int x)</td>
<td>void</td>
<td></td>
<td>Postconditions: d.x == x.</td>
</tr>
<tr>
<td>X::y(D&amp; d, int y)</td>
<td>void</td>
<td></td>
<td>Postconditions: d.y == y.</td>
</tr>
<tr>
<td>X::x(const D&amp; d) int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X::y(const D&amp; d) int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X::equal(const D&amp; l, const D&amp; r)</td>
<td>bool</td>
<td>Returns l.x == r.x &amp;&amp; l.y == r.y.</td>
<td></td>
</tr>
<tr>
<td>X::not_equal(const D&amp; l, const D&amp; r)</td>
<td>bool</td>
<td>Equivalent to return !equal(l, r);</td>
<td></td>
</tr>
<tr>
<td>X::create_bounding_-box()</td>
<td>B</td>
<td>Equivalent to return create_bounding_box(0.0f, 0.0f, 0.0f, 0.0f);</td>
<td></td>
</tr>
<tr>
<td>X::create_bounding_-box(float x, float y, float w, float h)</td>
<td>B</td>
<td>Returns an object b.</td>
<td>Postconditions: b.x == x, b.y == y, b.w == w, and b.h == h.</td>
</tr>
<tr>
<td>X::create_bounding_-box(const P&amp; tl, const P&amp; br)</td>
<td>B</td>
<td>Equivalent to return create_bounding_box(tl.x, tl.y, max(0.0f, br.x - tl.x), max(0.0f, br.y - tl.y));</td>
<td></td>
</tr>
<tr>
<td>X::x(B&amp; b, float x)</td>
<td>void</td>
<td></td>
<td>Postconditions: b.x == x.</td>
</tr>
<tr>
<td>X::y(B&amp; b, float y)</td>
<td>void</td>
<td></td>
<td>Postconditions: b.y == y.</td>
</tr>
<tr>
<td>X::width(B&amp; b, float w)</td>
<td>void</td>
<td></td>
<td>Postconditions: b.w == w.</td>
</tr>
<tr>
<td>X::height(B&amp; b, float h)</td>
<td>void</td>
<td></td>
<td>Postconditions: b.h == h.</td>
</tr>
<tr>
<td>X::top_left(B&amp; b, const P&amp; p)</td>
<td>void</td>
<td></td>
<td>Postconditions: b.x == p.x and b.y == p.y.</td>
</tr>
</tbody>
</table>
Table 6 — GraphicsMath requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::bottom_right(B &amp; b, const P &amp; p)</td>
<td>void</td>
<td>Postconditions: b.w == max(p.x - b.x, 0.0f) and b.h == max(p.y - b.y, 0.0f).</td>
<td></td>
</tr>
<tr>
<td>X::x(const B &amp; b)</td>
<td>float</td>
<td>Returns b.x.</td>
<td></td>
</tr>
<tr>
<td>X::y(const B &amp; b)</td>
<td>float</td>
<td>Returns b.y.</td>
<td></td>
</tr>
<tr>
<td>X::width(const B &amp; b)</td>
<td>float</td>
<td>Returns b.w.</td>
<td></td>
</tr>
<tr>
<td>X::height(const B &amp; b)</td>
<td>float</td>
<td>Returns b.h.</td>
<td></td>
</tr>
<tr>
<td>X::top_left(const B &amp; b)</td>
<td>P</td>
<td>Postconditions: p.x == b.x and p.y == b.y.</td>
<td>Returns an object p.</td>
</tr>
<tr>
<td>X::bottom_right(const B &amp; b)</td>
<td>P</td>
<td>Postconditions: p.x == b.x + b.w and p.y == b.y + b.h.</td>
<td>Returns an object p.</td>
</tr>
<tr>
<td>X::equal(const B &amp; l, const B &amp; r)</td>
<td>bool</td>
<td>Equivalent to return !equal(l, r);</td>
<td>Returns l.x == r.x &amp;&amp; l.y == r.y &amp;&amp; l.w == r.w &amp;&amp; l.h == r.h.</td>
</tr>
<tr>
<td>X::not_equal(const B &amp; l, const B &amp; r)</td>
<td>bool</td>
<td>Equivalent to return !equal(l, r);</td>
<td></td>
</tr>
<tr>
<td>X::create_circle()</td>
<td>C</td>
<td>Equivalent to return create_circle(create_point_2d(0.0f, 0.0f), 0.0f);</td>
<td>Returns an object c.</td>
</tr>
<tr>
<td>X::create_circle(const P &amp; p, float r)</td>
<td>C</td>
<td>Requires: r &gt;= 0.0f.</td>
<td></td>
</tr>
<tr>
<td>X::center(C &amp; c, const P &amp; p)</td>
<td>void</td>
<td>Postconditions: c.x == p.x and c.y == p.y.</td>
<td>Returns an object p.</td>
</tr>
<tr>
<td>X::radius(C &amp; c, float r)</td>
<td>void</td>
<td>Postconditions: c.x == p.x and c.y == p.y.</td>
<td>Requires: r &gt;= 0.0f.</td>
</tr>
<tr>
<td>X::center(const C &amp; c)</td>
<td>P</td>
<td>Postconditions: p.x == c.x and p.y == c.y.</td>
<td>Returns an object p.</td>
</tr>
<tr>
<td>X::radius(const C &amp; c)</td>
<td>float</td>
<td>Returns c.r.</td>
<td></td>
</tr>
<tr>
<td>X::equal(const C &amp; c, bool l, const C &amp; r)</td>
<td>bool</td>
<td>Equivalent to return !equal(l, r);</td>
<td>Returns l.x == r.x &amp;&amp; l.y == r.y &amp;&amp; l.r == r.r.</td>
</tr>
<tr>
<td>X::not_equal(const C &amp; l, const C &amp; r)</td>
<td>bool</td>
<td>Equivalent to return !equal(l, r);</td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Operational semantics</td>
<td>Assertion/note</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>X::point_for_angle(float a, float m)</td>
<td>P</td>
<td>Returns <code>transform_pt(create_rotate(a), create_point_2d(m, 0.0f))</code></td>
<td></td>
</tr>
<tr>
<td>X::point_for_angle(float a, const P&amp; r)</td>
<td>P</td>
<td>Returns <code>multiply(transform_pt(create_rotate(a), create_point_2d(1.0f, 0.0f)), r)</code></td>
<td></td>
</tr>
<tr>
<td>X::angle_for_point(const P&amp; c, const P&amp; p)</td>
<td>float</td>
<td>Equivalent to: `const float co = pi&lt;float&gt; / 180'000.0f; auto a = atan2(-(p.y - c.y), p.x - c.x); if (abs(a) &lt; co</td>
<td></td>
</tr>
<tr>
<td>X::arc_start(const P&amp; c, float sa, const P&amp; r, const M&amp; m)</td>
<td>P</td>
<td>Equivalent to: <code>auto lm = m; lm.m20 = 0.0f; lm.m21 = 0.0f; return add(c, transform_pt(lm, point_for_angle(sa, r)))</code></td>
<td></td>
</tr>
<tr>
<td>X::arc_center(const P&amp; c, float sa, const P&amp; r, const M&amp; m)</td>
<td>P</td>
<td>Equivalent to: <code>auto lm = m; lm.m20 = 0.0f; lm.m21 = 0.0f; auto o = point_for_angle(two_pi&lt;float&gt; - sa, r); o.y = -o.y; return subtract(c, transform_pt(lm, o));</code></td>
<td></td>
</tr>
<tr>
<td>X::arc_end(const P&amp; c, float ea, const P&amp; r, const M&amp; m)</td>
<td>P</td>
<td>Equivalent to: <code>auto lm = m; lm.m20 = 0.0f; lm.m21 = 0.0f; auto pt = transform_pt(create_rotate(ea), r); pt.y = -pt.y; return add(c, transform_pt(lm, pt));</code></td>
<td></td>
</tr>
</tbody>
</table>
8 Graphics surfaces

8.1 General

This Clause defines requirements on GraphicsSurfaces types.

Most classes specified in Clause 12, Clause 13, and Clause 15 need a set of related types and functions to complete the definition of their semantics. These types and functions are provided as a set of typedef-names and nested classes containing typedef-names and static member functions in the template parameter GraphicsSurfaces used by each such template. This Clause defines the names of the classes and the semantics of their members.

8.2 Requirements

8.2.1 Classes

A GraphicsSurfaces type is a class template with one type parameter. The template type argument of an instantiation of a GraphicsSurfaces specialization shall meet the requirements of a GraphicsMath type (See: Clause 7).

A GraphicsSurfaces type contains a typedef-name graphics_math_type, which is an identifier for the template argument. It also contains a typedef-name graphics_surfaces_type, which is an identifier for the GraphicsSurfaces type.

Example:

```
template <class GraphMath>
struct GraphSurf {
    using graphics_math_type = GraphMath;
    using graphics_surfaces_type = GraphSurf;
    // ...
};
```

A GraphicsSurfaces is required to have the following public nested classes:

1. additional_image_file_formats
2. additional_formats
3. brushes
4. paths
5. surface_states
6. surfaces

8.2.2 additional_image_file_formats requirements

Let X be a GraphicsSurfaces type.

The X::additional_image_file_formats class contains zero or more image_file_format enumerators that represent implementation-defined additional data formats that the implementation can both construct an image_surface object from using the appropriate constructor and save an image_surface object to using image_surface::save. These are called read/write image format enumerators.

The values of read/write image format enumerators shall be in the range [10000, 19999].

The X::additional_image_file_formats class also contains the following nested classes:

1. read_only
2. write_only

The additional_image_file_formats class contains zero or more image_file_format enumerators that represent implementation-defined additional data formats that the implementation can construct an image_surface object from using the appropriate constructor but cannot save an image_surface object to using image_surface::save. These are called read only image format enumerators.
The values of read only image format enumerators shall be in the range \([20000, 29999]\).

The `additional_image_file_formats::write_only` class contains zero or more `image_file_format` enumerators that represent implementation-defined additional data formats that the implementation can construct an `image_surface` object from using the appropriate constructor and save an `image_surface` object to using `image_surface::save` but cannot construct an `image_surface` object from using any constructor. These are called `write only image format enumerators`.

The values of write only image format enumerators shall be in the range \([30000, 39999]\).

**8.2.3 additional_formats requirements** \[io2d.graphsurf.reqs.addform\]

1. Let \(X\) be a GraphicsSurfaces type.
2. The `X::additional_formats` class contains zero or more `format` enumerators that represent implementation-defined additional visual data formats that the implementation supports.
3. The size in bytes, byte order, and interpretation of values within each channel of each additional visual data format is *implementation-defined*.
4. The values of the additional visual data format enumerators shall be in the range \([10000, 39999]\).

**8.2.4 brushes requirements** \[io2d.graphsurf.reqs.brushes\]

1. Brushes are described in Clause 13.
2. Let \(X\) be a GraphicsSurfaces type.
3. Let \(M\) be `X::graphics_math_type`.
4. Table 7 describes the observable effects of a member functions of `X::brushes`.
5. `X::brushes` contains a *typedef-name*, `brush_data_type`, which is an identifier for a class type capable of storing all data required to support a brush of any type described in Clause 13. \[Note: The information in 13.7.3 is particularly important. —end note\]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::brushes::create_brush(const rgba_color&amp; c)</code></td>
<td><code>brush_data_type</code></td>
<td>Returns an object (b).</td>
<td>Postconditions: (b) is a solid color brush, its visual data is (c) (See: 13.7.3.1), and (X::brushes::get_brush_type(b) == brush_type::solid_color). [Note: Solid color does not imply opaque. The color may be translucent or even transparent. —end note]</td>
</tr>
</tbody>
</table>
Table 7 — `X::brushes` requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;class InputIterator&gt; create_brush(const basic_point_2d&lt;M&gt;&amp; be, const basic_point_2d&lt;M&gt;&amp; en, InputIterator first, InputIterator last)</code></td>
<td>brush_data_type</td>
<td>Returns an object <code>b</code>.</td>
<td>Postconditions: <code>b</code> is a linear gradient brush, its begin point is <code>be</code>, its end point is <code>en</code>, its gradient stops are formed using the sequential series of <code>gradient_stop</code> objects beginning at <code>first</code> and ending at <code>last</code> - 1 (See 13.2.2 and 13.2.4), and <code>X::brushes::get_brush_type(b) == brush_type::linear</code>.</td>
</tr>
<tr>
<td><code>create_brush(const basic_point_2d&lt;M&gt;&amp; be, const basic_point_2d&lt;M&gt;&amp; en, initializer_list&lt;gradient_stop&gt; il)</code></td>
<td>brush_data_type</td>
<td>Returns an object <code>b</code>.</td>
<td>Postconditions: <code>b</code> is a linear gradient brush, its begin point is <code>be</code>, its end point is <code>en</code>, its gradient stops are formed using the sequential series of <code>gradient_stop</code> objects in <code>il</code> (See 13.2.2 and 13.2.4), and <code>X::brushes::get_brush_type(b) == brush_type::linear</code>.</td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Operational semantics</td>
<td>Assertion/note</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>--------------------</td>
<td>----------------------------------------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>template &lt;class InputIterator&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>create_brush(const basic_circle&lt;M&gt;&amp; be,</td>
<td>brush_data_type</td>
<td>Returns an object b.</td>
<td>Postconditions:</td>
</tr>
<tr>
<td>const basic_circle&lt;M&gt;&amp; en,</td>
<td></td>
<td></td>
<td>b is a radial</td>
</tr>
<tr>
<td>InputIterator first,</td>
<td></td>
<td></td>
<td>gradient brush,</td>
</tr>
<tr>
<td>InputIterator last)</td>
<td></td>
<td></td>
<td>its start circle is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>be, its end circle is en,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>its gradient stops</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>are formed using the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sequential series</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>of gradient_stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>objects beginning at first and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ending at last - 1 (See 13.2.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>and 13.2.4), and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X::brushes::get_-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>brush_type(b) == brush_-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>type::radial.</td>
</tr>
<tr>
<td>create_brush(const basic_circle&lt;M&gt;&amp; be,</td>
<td>brush_data_type</td>
<td>Returns an object b.</td>
<td>Postconditions:</td>
</tr>
<tr>
<td>const basic_circle&lt;M&gt;&amp; en,</td>
<td></td>
<td></td>
<td>b is a radial</td>
</tr>
<tr>
<td>initializer_list&lt;gradient_stop&gt; il)</td>
<td></td>
<td></td>
<td>gradient brush,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>its start circle is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>be, its end circle is en,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>its gradient stops</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>are formed using the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sequential series</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>of gradient_stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>objects in il</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(See 13.2.3 and 13.2.4), and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X::brushes::get_-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>brush_type(b) == brush_-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>type::radial.</td>
</tr>
<tr>
<td>create_brush(basic_image_surface&lt;graphics_</td>
<td>brush_data_type</td>
<td>Returns an object b.</td>
<td>Postconditions:</td>
</tr>
<tr>
<td>surfaces_type&gt;&amp;&amp; i)</td>
<td></td>
<td></td>
<td>b is a surface brush,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>its visual data is the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>raster graphics data from i,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>and X::brushes::get_-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>brush_type(b) == brush_-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>type::surface</td>
</tr>
</tbody>
</table>
8.2.5 paths requirements

Paths are described in Clause 12.

Let X be a GraphicsSurfaces type.

Let G be X::graphics_math_type.

Table 8 describes the observable effects of the member functions of X::paths.

Table 8 defines the required typedef-names in X::paths, which are identifiers for class types capable of storing all data required to support the corresponding class template.

### Table 8 — X::paths typedef-names

<table>
<thead>
<tr>
<th>typedef-name</th>
<th>Class data</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs_cubic_curve_data_type</td>
<td>basic_figure_items&lt;X&gt;::abs_cubic_curve</td>
</tr>
<tr>
<td>abs_line_data_type</td>
<td>basic_figure_items&lt;X&gt;::abs_line</td>
</tr>
<tr>
<td>abs_matrix_data_type</td>
<td>basic_figure_items&lt;X&gt;::abs_matrix</td>
</tr>
<tr>
<td>abs_new_figure_data_type</td>
<td>basic_figure_items&lt;X&gt;::abs_new_figure</td>
</tr>
<tr>
<td>abs_quadratic_curve_data_type</td>
<td>basic_figure_items&lt;X&gt;::abs_quadratic_curve</td>
</tr>
<tr>
<td>arc_data_type</td>
<td>basic_figure_items&lt;X&gt;::arc</td>
</tr>
<tr>
<td>close_figure_data_type</td>
<td>basic_figure_items&lt;X&gt;::close_data</td>
</tr>
<tr>
<td>interpreted_path_data_type</td>
<td>basic_interpreted_path&lt;X&gt;</td>
</tr>
<tr>
<td>rel_cubic_curve_data_type</td>
<td>basic_figure_items&lt;X&gt;::rel_cubic_curve</td>
</tr>
<tr>
<td>rel_line_data_type</td>
<td>basic_figure_items&lt;X&gt;::rel_line</td>
</tr>
<tr>
<td>rel_matrix_data_type</td>
<td>basic_figure_items&lt;X&gt;::rel_matrix</td>
</tr>
<tr>
<td>rel_new_figure_data_type</td>
<td>basic_figure_items&lt;X&gt;::rel_new_figure</td>
</tr>
<tr>
<td>rel_quadratic_curve_data_type</td>
<td>basic_figure_items&lt;X&gt;::rel_quadratic_curve</td>
</tr>
<tr>
<td>revert_matrix_data_type</td>
<td>basic_figure_items&lt;X&gt;::revert_matrix</td>
</tr>
</tbody>
</table>

6 [Note: An object of type basic_interpreted_path<X> is an immutable object. As such, the contents of the class type for which X::paths::interpreted_path_data_type is an identifier are able to be highly tailored to the platform and environment targeted by X. — end note]

7 In Table 10, AC denotes the type X::paths::abs_cubic_curve_data_type, AL denotes the type X::paths::abs_line_data_type, AN denotes the type X::paths::abs_matrix_data_type, AN denotes the type X::paths::abs_new_figure_data_type, AQ denotes the type X::paths::abs_quadratic_curve_data_type, ARC denotes the type X::paths::arc_data_type, IP denotes the type X::paths::interpreted_path_data_type, RC denotes the type X::paths::rel_cubic_curve_data_type, RL denotes the type X::paths::rel_line_data_type, RM denotes the type X::paths::rel_matrix_data_type, RN denotes the type X::paths::rel_new_figure_data_type, RQ denotes the type X::paths::rel_quadratic_curve_data_type, BC denotes the type basic_matrix_2d<X::graphics_math_type>, and P denotes the type basic_point_2d<X::graphics_math_type>.

8 In order to describe the observable effects of functions contained in Table 10, Table 9 describes the types contained in X as if they possessed certain member data.

9 [Note: Certain types do not require any member data to describe the observable effects of the functions they are used by and thus do not appear in Table (9). — end note]
Table 9 — X::paths type member data

<table>
<thead>
<tr>
<th>Type</th>
<th>Member data</th>
<th>Member data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs_cubic_curve_data_type</td>
<td>cpt1</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>abs_cubic_curve_data_type</td>
<td>cpt2</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>abs_cubic_curve_data_type</td>
<td>ept</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>abs_line_data_type</td>
<td>pt</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>abs_matrix_data_type</td>
<td>m</td>
<td>basic_matrix_2d&lt;G&gt;</td>
</tr>
<tr>
<td>abs_new_figure_data_type</td>
<td>pt</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>abs_quadratic_curve_data_type</td>
<td>cpt</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>abs_quadratic_curve_data_type</td>
<td>ept</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>arc_data_type</td>
<td>rad</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>arc_data_type</td>
<td>rot</td>
<td>float</td>
</tr>
<tr>
<td>arc_data_type</td>
<td>sa</td>
<td>float</td>
</tr>
<tr>
<td>rel_cubic_curve_data_type</td>
<td>cpt1</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>rel_cubic_curve_data_type</td>
<td>cpt2</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>rel_cubic_curve_data_type</td>
<td>ept</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>rel_line_data_type</td>
<td>pt</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>rel_matrix_data_type</td>
<td>m</td>
<td>basic_matrix_2d&lt;G&gt;</td>
</tr>
<tr>
<td>rel_new_figure_data_type</td>
<td>pt</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>rel_quadratic_curve_data_type</td>
<td>cpt</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
<tr>
<td>rel_quadratic_curve_data_type</td>
<td>ept</td>
<td>basic_point_2d&lt;G&gt;</td>
</tr>
</tbody>
</table>

Table 10 — X::paths requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::paths::create_abs_cubic_curve()</td>
<td>AC</td>
<td>Effects: Equivalent to return create_abs_cubic_curve(P(), P(), P());</td>
<td></td>
</tr>
<tr>
<td>X::paths::create_abs_cubic_curve(const P&amp; cpt1, const P&amp; cpt2, const P&amp; ept)</td>
<td>abs_cubic_curve_data_type</td>
<td>Returns: An object ac.</td>
<td></td>
</tr>
<tr>
<td>X::paths::control_pt1(AC&amp; ac, const P&amp; pt)</td>
<td>void</td>
<td>Postconditions: ac.cpt1 == pt, ac.cpt2 == ept, and ac.cpt1 == ept</td>
<td></td>
</tr>
<tr>
<td>X::paths::control_pt2(AC&amp; ac, const P&amp; pt)</td>
<td>void</td>
<td>Postconditions: ac.cpt2 == pt.</td>
<td></td>
</tr>
<tr>
<td>X::paths::end_pt(AC&amp; ac, const P&amp; pt)</td>
<td>void</td>
<td>Postconditions: ac.ept == pt.</td>
<td></td>
</tr>
<tr>
<td>X::paths::control_pt1(const AC&amp; ac)</td>
<td>P</td>
<td>Returns: ac.cpt1.</td>
<td></td>
</tr>
<tr>
<td>X::paths::control_pt2(const AC&amp; ac)</td>
<td>P</td>
<td>Returns: ac.cpt2.</td>
<td></td>
</tr>
<tr>
<td>X::paths::end_pt(const AC&amp; ac)</td>
<td>P</td>
<td>Returns: ac.ept.</td>
<td></td>
</tr>
<tr>
<td>X::paths::create_abs_line()</td>
<td>AL</td>
<td>Effects: Equivalent to return create_abs_line(P);</td>
<td></td>
</tr>
</tbody>
</table>

§ 8.2.5
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::paths::to(AL&amp; al, void const P&amp; p)</td>
<td>P</td>
<td>Returns: al.pt.</td>
<td></td>
</tr>
<tr>
<td>X::paths::to(const AL&amp; al)</td>
<td>P</td>
<td>Returns: al.pt.</td>
<td></td>
</tr>
<tr>
<td>X::paths::create_abs_matrix()</td>
<td>AM</td>
<td>Equivalent to return create_abs_matrix(M());</td>
<td></td>
</tr>
<tr>
<td>X::paths::create_abs_matrix(const M&amp; m)</td>
<td>AM</td>
<td>Returns: An object am.</td>
<td>Postconditions: am.m == m.</td>
</tr>
<tr>
<td>X::paths::matrix(AM&amp; am, const M&amp; m)</td>
<td>P</td>
<td>Returns: am.m.</td>
<td>Postconditions: am.m == m.</td>
</tr>
<tr>
<td>X::paths::matrix(const AM&amp; am)</td>
<td>P</td>
<td>Returns: am.m.</td>
<td></td>
</tr>
<tr>
<td>X::paths::create_abs_new_figure()</td>
<td>AN</td>
<td>Effects: Equivalent to return create_abs_new_figure(P());</td>
<td></td>
</tr>
<tr>
<td>X::paths::at(const AN&amp; an)</td>
<td>P</td>
<td>Returns: an.pt.</td>
<td></td>
</tr>
<tr>
<td>X::paths::create_abs_quadratic_curve()</td>
<td>AQ</td>
<td>Equivalent to return create_abs_quadratic_curve(P(), P());</td>
<td></td>
</tr>
<tr>
<td>X::paths::create_abs_quadratic_curve(const P&amp; cpt, const P&amp; ept)</td>
<td>AQ</td>
<td>Returns: An object aq.</td>
<td>Postconditions: aq.cpt == cpt and aq.ept == ept.</td>
</tr>
<tr>
<td>X::paths::control_pt(const AQ&amp; aq)</td>
<td>P</td>
<td>Returns: aq.cpt.</td>
<td></td>
</tr>
<tr>
<td>X::paths::end_pt(const AQ&amp; aq)</td>
<td>P</td>
<td>Returns: aq.ept.</td>
<td></td>
</tr>
<tr>
<td>X::paths::create_arc()</td>
<td>ARC</td>
<td>Effects: Equivalent to return create_arc(P(), 0.0f, 0.0f);</td>
<td>Postconditions: arc.rad == rad, arc.rot == rot, and arc.sa == sa.</td>
</tr>
<tr>
<td>X::paths::create_arc(const P&amp; rad, float rot, float sa)</td>
<td>ARC</td>
<td>Returns: An object arc.</td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Operational semantics</td>
<td>Assertion/note pre-/post-condition</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-----------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>X::paths::radius(ARC&amp; arc, const P&amp; rad)</td>
<td>void</td>
<td>Postconditions: arc.rad == rad.</td>
<td></td>
</tr>
<tr>
<td>X::paths::rotation(ARC&amp; arc, float rot)</td>
<td>void</td>
<td>Postconditions: arc.rot == rot.</td>
<td></td>
</tr>
<tr>
<td>X::paths::start_angle(ARC&amp; arc, float sa)</td>
<td>void</td>
<td>Postconditions: arc.sa == sa.</td>
<td></td>
</tr>
<tr>
<td>X::paths::radius(const ARC&amp; arc)</td>
<td>P</td>
<td>Returns: arc.rad.</td>
<td></td>
</tr>
<tr>
<td>X::paths::rotation(const ARC&amp; arc)</td>
<td>float</td>
<td>Returns: arc.rot.</td>
<td></td>
</tr>
<tr>
<td>X::paths::start_angle(const ARC&amp; arc)</td>
<td>float</td>
<td>Returns: arc.sa.</td>
<td></td>
</tr>
<tr>
<td>X::paths::center(const ARC&amp; arc, const P&amp; spt, const M&amp; m)</td>
<td>Returns: As-if: auto lmtx = m; lmtx.m20 = 0.0f; lmtx.m21 = 0.0f; auto ctrOffset = point_for_angle&lt;G&gt;(two_pi&lt;float&gt; - arc.sa, arc.rad); ctrOffset.y(-ctrOffset.y); return spt - ctrOffset * lmtx;</td>
<td>[Note: spt is the starting point of the arc. m is the transformation matrix being used. — end note]</td>
<td></td>
</tr>
<tr>
<td>X::paths::end_pt(const ARC&amp; arc, const P&amp; spt, const M&amp; m)</td>
<td>P</td>
<td>Returns: As-if: auto lmtx = m; lmtx.m20 = 0.0f; lmtx.m21 = 0.0f; auto tffm = M::create_rotate(arc.sa + arc.rot); auto pt = arc.rad * tffm; pt.y(-pt.y()); return spt + pt * lmtx;</td>
<td>[Note: spt is the starting point of the arc. m is the transformation matrix being used. — end note]</td>
</tr>
<tr>
<td>X::paths::create_interpreted_path()</td>
<td>IP</td>
<td>Returns: An object ip.</td>
<td>Postconditions: ip has zero figures (See: 12.3.16)</td>
</tr>
</tbody>
</table>

Table 10 — X::paths requirements (continued)
Table 10 — X::paths requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;class ForwardIterator&gt; X::paths::create_interpreted_ path&lt;ForwardIterator first, ForwardIterator last</td>
<td>IP</td>
<td>Returns: An object ip.</td>
<td>Postconditions: ip contains a zero or more figure items as determined by evaluating the sequence of figure_::figure_::item objects beginning with first and ending with last in the manner described in 12.3.16. Remarks: The internal data of the interpreted path should be in a form that is best suited to take advantage of the platform and environment targeted by X.</td>
</tr>
<tr>
<td>X::paths::create_rel_cubic_curve()</td>
<td>RC</td>
<td>Effects: Equivalent to return create_rel_cubic_curve(P(), P(), P());</td>
<td></td>
</tr>
<tr>
<td>X::paths::create_rel_cubic_curve(const P&amp; cpt1, const P&amp; cpt2, const P&amp; ept)</td>
<td>RC</td>
<td>Returns: An object rc.</td>
<td>Postconditions: rc.cpt1 == cpt1, rc.cpt2 == cpt2, and rc.ept == ept.</td>
</tr>
<tr>
<td>X::paths::control_pt1(RC&amp; rc, const P&amp; pt)</td>
<td>void</td>
<td>Postconditions: rc.cpt1 == pt.</td>
<td></td>
</tr>
<tr>
<td>X::paths::end_pt(RC&amp; rc, const P&amp; pt)</td>
<td>void</td>
<td>Postconditions: rc.ept == pt.</td>
<td></td>
</tr>
<tr>
<td>X::paths::control_pt1(const RC&amp; a)</td>
<td>P</td>
<td>Returns: rc.cpt1.</td>
<td></td>
</tr>
<tr>
<td>X::paths::control_pt2(const RC&amp; rc)</td>
<td>P</td>
<td>Returns: rc.cpt2.</td>
<td></td>
</tr>
</tbody>
</table>
Table 10 — X::paths requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::paths::end_pt(const RC&amp; rc)</td>
<td>P</td>
<td>Returns: rc.ept.</td>
<td></td>
</tr>
<tr>
<td>X::paths::create_rel_line()</td>
<td>RL</td>
<td>Effects: Equivalent to return create_rel_line(P);</td>
<td></td>
</tr>
</tbody>
</table>
| X::paths::create_rel_line(const P& p)           | RL          | Returns: An object rl.
|                                                  |             | Postconditions:       |
| X::paths::to(RL& al, void const P& p)           | P           | Returns: rl.pt.       |
| X::paths::to(const RL& rl)                      |             | Postconditions:       |
| X::paths::create_rel_matrix()                   | RM          | Equivalent to return create_rel_matrix(M()); |                |
| X::paths::create_rel_matrix(const M& m)         | RM          | Returns: An object rm.
|                                                  |             | Postconditions:       |
| X::paths::matrix(RM& void am, const M& m)       | M           | Returns: am.m.        |
| X::paths::create_rel_new_figure()               | RN          | Effects: Equivalent to return create_rel_new_figure(P()); |                |
| X::paths::create_rel_new_figure(const P& p)      | RN          | Returns: An object rn.
|                                                  |             | Postconditions:       |
| X::paths::at(RN& rn, void const P& p)           | P           | Returns: rn.pt.       |
| X::paths::at(const RN& rn)                      |             | Postconditions:       |
| X::paths::create_rel_quadratic_curve()          | RQ          | Equivalent to return create_rel_quadratic_curve(P(), P()); |                |
| X::paths::create_rel_quadratic_curve(const P& cpt, constant P& ept) | RQ          | Returns: An object rq.|
| X::paths::control_pt(RQ& rq, const P& p)        | void        | Postconditions:       |
| X::paths::end_pt(RQ& rq, const P& p)            | void        | Postconditions:       |
| X::paths::control_pt(const RQ& rq)              | P           | Returns: rq.cpt.      |
| X::paths::end_pt(const RQ& rq)                  | P           | Returns: rq.ept.      |
8.2.6 surface_state_props requirements

Surface state data are described in Clause 14.

Let $X$ be a GraphicsSurfaces type.

Let $G$ be $X::graphics_math_type$.

Table 13 describes the observable effects of the member functions of $X::surface_state_props$.

Table 11 defines the required typedef-names in $X::surface_state_props$, which are identifiers for class types capable of storing all data required to support the corresponding class template.

### Table 11 — $X::surface_state_props$ typedef-names

<table>
<thead>
<tr>
<th>typedef-name</th>
<th>Class data</th>
</tr>
</thead>
<tbody>
<tr>
<td>render_props_data_type</td>
<td>basic_render_props</td>
</tr>
<tr>
<td>brush_props_data_type</td>
<td>basic_brush_props</td>
</tr>
<tr>
<td>clip_props_data_type</td>
<td>basic_clip_props</td>
</tr>
<tr>
<td>stroke_props_data_type</td>
<td>basic_stroke_props</td>
</tr>
<tr>
<td>mask_props_data_type</td>
<td>basic_mask_props</td>
</tr>
<tr>
<td>dashes_data_type</td>
<td>basic_dashes</td>
</tr>
</tbody>
</table>

In Table 13, RE denotes the type $X::surface_state_props::render_props_data_type$, BR denotes the type $X::surface_state_props::brush_props_data_type$, CL denotes the type $X::surface_state_props::clip_props_data_type$, ST denotes the type $X::surface_state_props::stroke_props_data_type$, FP denotes the type $X::surface_state_props::fill_props_data_type$, MA denotes the type $X::surface_state_props::mask_props_data_type$, DA denotes the type $X::surface_state_props::dashes_data_type$, BB denotes the type $basic_bounding_box<G>$, IP denotes the type $basic_interpreted_path<X>$, FI denotes the type $basic_figure_items<X>::figure_item$, M denotes the type $basic_matrix_2d<G>$, and P denotes the type $basic_point_2d<G>$.

In order to describe the observable effects of functions contained in Table 13, Table 12 describes the types contained in $X$ as-if they possessed certain member data.

### Table 12 — $X::surface_state_props$ type member data

<table>
<thead>
<tr>
<th>Type</th>
<th>Member data</th>
<th>Member data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>render_props_data_type</td>
<td>fi</td>
<td>filter</td>
</tr>
<tr>
<td>render_props_data_type</td>
<td>m</td>
<td>M</td>
</tr>
<tr>
<td>render_props_data_type</td>
<td>c</td>
<td>compositing_op</td>
</tr>
<tr>
<td>brush Props_data_type</td>
<td>w</td>
<td>wrap_mode</td>
</tr>
<tr>
<td>brush Props_data_type</td>
<td>fi</td>
<td>filter</td>
</tr>
<tr>
<td>brush Props_data_type</td>
<td>m</td>
<td>M</td>
</tr>
<tr>
<td>clip Props_data_type</td>
<td>optional&lt;IP&gt;</td>
<td>c</td>
</tr>
<tr>
<td>clip Props_data_type</td>
<td>fr</td>
<td>fill_rule</td>
</tr>
<tr>
<td>stroke Props_data_type</td>
<td>lw</td>
<td>float</td>
</tr>
<tr>
<td>stroke Props_data_type</td>
<td>ml</td>
<td>float</td>
</tr>
<tr>
<td>stroke Props_data_type</td>
<td>lc</td>
<td>line_cap</td>
</tr>
<tr>
<td>stroke Props_data_type</td>
<td>lj</td>
<td>line_join</td>
</tr>
<tr>
<td>stroke Props_data_type</td>
<td>aa</td>
<td>antialias</td>
</tr>
<tr>
<td>fill Props_data_type</td>
<td>fr</td>
<td>fill_rule</td>
</tr>
<tr>
<td>fill Props_data_type</td>
<td>aa</td>
<td>antialias</td>
</tr>
<tr>
<td>mask Props_data_type</td>
<td>wm</td>
<td>wrap_mode</td>
</tr>
<tr>
<td>mask Props_data_type</td>
<td>fi</td>
<td>filter</td>
</tr>
<tr>
<td>mask Props_data_type</td>
<td>m</td>
<td>M</td>
</tr>
<tr>
<td>dashes Props_data_type</td>
<td>o</td>
<td>float</td>
</tr>
<tr>
<td>dashes Props_data_type</td>
<td>p</td>
<td>vector&lt;float&gt;</td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Operational semantics</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>X::surface_state::create_render_props(filter f = filter::good, M m = M{}, compositing op c = compositing::over)</td>
<td>RE</td>
<td>Returns: An object r.</td>
</tr>
<tr>
<td>X::surface_state::filtering(RA&amp; r, filter fi)</td>
<td>void</td>
<td>Postconditions: r.fi == fi.</td>
</tr>
<tr>
<td>X::surface_state::surface_matrix(RA&amp; r, const M&amp; m)</td>
<td>void</td>
<td>Postconditions: r.m == m.</td>
</tr>
<tr>
<td>X::surface_state::compositing(RA&amp; r, compositing_op c)</td>
<td>void</td>
<td>Postconditions: r.c == c.</td>
</tr>
<tr>
<td>X::surface_state::filtering(const RA&amp; r)</td>
<td>filter</td>
<td>Returns: r.fi.</td>
</tr>
<tr>
<td>X::surface_state::surface_matrix(const RA&amp; r)</td>
<td>M</td>
<td>Returns: r.m.</td>
</tr>
<tr>
<td>X::surface_state::compositing(const RA&amp; r)</td>
<td>compositing_op</td>
<td>Returns: r.c.</td>
</tr>
<tr>
<td>X::surface_state::create_brush_props(wrap_mode w = wrap_mode::none, filter fi = filter::good, const M&amp; m = M{})</td>
<td>BR</td>
<td>Returns: An object b.</td>
</tr>
<tr>
<td>X::surface_state::brush_wrap_mode(BR&amp; b, wrap_mode w)</td>
<td>void</td>
<td>Postconditions: b.w == w.</td>
</tr>
<tr>
<td>X::surface_state::brush_filter(BR&amp; b, filter fi)</td>
<td>void</td>
<td>Postconditions: b.fi == fi.</td>
</tr>
<tr>
<td>X::surface_state::brush_matrix(BR&amp; b, const M&amp; m)</td>
<td>void</td>
<td>Postconditions: b.m == m.</td>
</tr>
<tr>
<td>X::surface_state::wrap_mode</td>
<td>return type</td>
<td>Returns: b.w.</td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Operational semantics</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>X::surface_state_props::brush_filter(const BR&amp; b)</td>
<td>filter</td>
<td>Returns: b.fi.</td>
</tr>
<tr>
<td>X::surface_state_props::brush_matrix(const BR&amp; b)</td>
<td>M</td>
<td>Returns: b.m.</td>
</tr>
<tr>
<td>X::surface_state_props::create_clip_props()</td>
<td>CL</td>
<td>Returns: An object c.</td>
</tr>
<tr>
<td>X::surface_state_props::create_clip_props(const BB&amp; b, fill_rule fr)</td>
<td>CL</td>
<td>Returns: An object c.</td>
</tr>
<tr>
<td>template &lt;class Allocator&gt; X::surface_state_props::create_clip_props(const basic_path_builder&lt;X, Allocator&gt;&amp; pb, fill_rule fr)</td>
<td>CL</td>
<td>Returns: An object c.</td>
</tr>
<tr>
<td>template &lt;class InputIterator&gt; X::surface_state_props::create_clip_props(InputIterator first, InputIterator last, fill_rule fr)</td>
<td>CL</td>
<td>Returns: An object c.</td>
</tr>
<tr>
<td>X::surface_state_props::create_clip_props(initializer_list&lt;FI&gt; il, fill_rule fr)</td>
<td>CL</td>
<td>Returns: An object c.</td>
</tr>
<tr>
<td>X::surface_state_props::create_clip_props(const IP&amp; ip, fill_rule fr)</td>
<td>CL</td>
<td>Returns: An object c.</td>
</tr>
<tr>
<td>X::surface_state_props::clip(CL&amp; c, nullopt_t)</td>
<td>void</td>
<td></td>
</tr>
<tr>
<td>X::surface_state_props::clip(CL&amp; c, const BB&amp; b)</td>
<td>void</td>
<td></td>
</tr>
</tbody>
</table>
Table 13 — `X::surface_state_props` requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;class Allocator&gt; X::surface_state_props::clip(CL&amp; c, const basic_path_builder&lt;X, Allocator&gt;&amp; pb)</code></td>
<td>void</td>
<td>Postconditions:</td>
<td><code>c.c == IP(pb).</code></td>
</tr>
<tr>
<td><code>template &lt;class InputIterator&gt; X::surface_state_props::clip(CL&amp; c, InputIterator first, InputIterator last)</code></td>
<td>void</td>
<td>Postconditions:</td>
<td><code>c.c == IP(first, last).</code></td>
</tr>
<tr>
<td><code>X::surface_state_props::clip(CL&amp; c, initializer_list&lt;FI&gt; il)</code></td>
<td>void</td>
<td>Postconditions:</td>
<td><code>c.c == IP(il).</code></td>
</tr>
<tr>
<td><code>X::surface_state_props::clip(CL&amp; c, const IP&amp; ip)</code></td>
<td>void</td>
<td>Postconditions:</td>
<td><code>c.c == ip.</code></td>
</tr>
<tr>
<td><code>X::surface_state_props::clip_fill_rule(CL&amp;c, fill_rule fr)</code></td>
<td>void</td>
<td>Postconditions:</td>
<td><code>c.fr == fr.</code></td>
</tr>
<tr>
<td><code>X::surface_state_props::create_stroke_props(float lw = 2.0f, line_cap lc = line_cap::none, line_join lj = line_join::miter, float ml = 10.0f, antialias aa = antialias::good)</code></td>
<td>ST</td>
<td>Returns: An object s.</td>
<td>Requires: <code>lw &gt;= 0.0f, ml &gt;= 1.0f, and ml &lt;= max_miter_limit().</code> Postconditions: <code>s.lw == lw, s.lc == lc, s.lj == lj, s.ml == ml, and s.aa == aa.</code></td>
</tr>
<tr>
<td><code>X::surface_state_props::line_width(ST&amp; s, float lw)</code></td>
<td>void</td>
<td>Postconditions:</td>
<td><code>s.lw == lw.</code></td>
</tr>
<tr>
<td><code>X::surface_state_props::stroke_line_cap(ST&amp; s, line_cap lc)</code></td>
<td>void</td>
<td>Postconditions:</td>
<td><code>s.lc == lc.</code></td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Operational semantics</td>
<td>Assertion/note pre-/post-condition</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>-------------</td>
<td>---------------------------------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td><code>X::surface_state_props::stroke_line_join(ST&amp; s, line_join lj)</code></td>
<td>void</td>
<td></td>
<td>Postconditions: s.lj == lj.</td>
</tr>
<tr>
<td><code>X::surface_state_props::miter_limit(ST&amp; s, float ml)</code></td>
<td>void</td>
<td></td>
<td>Postconditions: s.ml == ml.</td>
</tr>
<tr>
<td><code>X::surface_state_props::anti_aliasing(ST&amp; s, antialias aa)</code></td>
<td>void</td>
<td></td>
<td>Postconditions: s.aa == aa.</td>
</tr>
<tr>
<td><code>X::surface_state_props::line_width(const ST&amp; s)</code></td>
<td>float</td>
<td>Returns: s.lw.</td>
<td></td>
</tr>
<tr>
<td><code>X::surface_state_props::stroke_line_cap(const ST&amp; s)</code></td>
<td>line_cap</td>
<td>Returns: s.lc.</td>
<td></td>
</tr>
<tr>
<td><code>X::surface_state_props::stroke_line_join(const ST&amp; s)</code></td>
<td>line_join</td>
<td>Returns: s.lj.</td>
<td></td>
</tr>
<tr>
<td><code>X::surface_state_props::miter_limit(const ST&amp; s)</code></td>
<td>float</td>
<td>Returns: s.ml.</td>
<td></td>
</tr>
<tr>
<td><code>X::surface_state_props::anti_aliasing(const ST&amp; s)</code></td>
<td>antialias</td>
<td>Returns: s.aa.</td>
<td></td>
</tr>
<tr>
<td><code>X::surface_state_props::max_miter_limit()</code></td>
<td>float</td>
<td>Returns: An implementation-defined maximum value for ST::ml.</td>
<td>Postconditions: f.fr == fr and f.aa == aa.</td>
</tr>
<tr>
<td><code>X::surface_state_props::create_fill_props(fill_rule fr = fill_rule::winding, antialias aa = antialias::good)</code></td>
<td>FP</td>
<td>Returns: An object f.</td>
<td></td>
</tr>
<tr>
<td><code>X::surface_state_props::fill_fill_rule(FP&amp; f, fill_rule fr)</code></td>
<td>void</td>
<td></td>
<td>Postconditions: f.fr == fr.</td>
</tr>
<tr>
<td><code>X::surface_state_props::antialiasing(FP&amp; f, antialias aa)</code></td>
<td>void</td>
<td></td>
<td>Postconditions: f.aa == aa.</td>
</tr>
<tr>
<td><code>X::surface_state_props::fill_fill_rule(const FP&amp; f)</code></td>
<td>fill_rule</td>
<td>Returns: f.fr.</td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Operational semantics</td>
<td>Assertion/note pre-/post-condition</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>X::surface_state_-&gt;antialiasing(const FP&amp; f)</td>
<td>Returns: f.aa.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X::surface_state_-&gt;props::create_mask_&lt;br&gt;props(wrap_mode wm = wrap_mode::none,&lt;br&gt;filter fi = filter::good, const M&amp; m = M())</td>
<td>MA</td>
<td>Returns: An object ma.</td>
<td>Postconditions: ma.wm == wm, ma.fi == fi, and ma.m == m.</td>
</tr>
<tr>
<td>X::surface_state_-&gt;props::mask_wrap_&lt;br&gt;mode(MA&amp; ma, wrap_mode wm)</td>
<td>void</td>
<td></td>
<td>Postconditions: ma.wm == wm.</td>
</tr>
<tr>
<td>X::surface_state_-&gt;props::mask_&lt;br&gt;filter(MA&amp; ma, filter fi)</td>
<td>void</td>
<td></td>
<td>Postconditions: ma.fi == fi.</td>
</tr>
<tr>
<td>X::surface_state_-&gt;props::mask_&lt;br&gt;matrix(MA&amp; ma, const M&amp; m)</td>
<td>void</td>
<td></td>
<td>Postconditions: ma.m == m.</td>
</tr>
<tr>
<td>X::surface_state_-&gt;props::create_&lt;br&gt;dashes()</td>
<td>DA</td>
<td>Returns: ma.wm.</td>
<td></td>
</tr>
<tr>
<td>template&lt;class InputIterator&gt; X::surface_state_-&gt;props::create_&lt;br&gt;dashes(float o, InputIterator first, InputIterator last)</td>
<td>DA</td>
<td>Returns: An object d.</td>
<td>Postconditions: d.o == o and d.p == vector&lt;float&gt;(first, last).</td>
</tr>
<tr>
<td>X::surface_state_-&gt;props::create_&lt;br&gt;dashes(float o, initializer_list&lt;float&gt; il)</td>
<td>DA</td>
<td>Returns: An object d.</td>
<td>Postconditions: d.o == o and d.p == vector&lt;float&gt;(il).</td>
</tr>
</tbody>
</table>
Table 13 — X::surface_state_props requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::surface_state_-props::equal(const DA&amp; lhs, const DA&amp; rhs)</td>
<td>bool</td>
<td>Returns: lhs.o == rhs.o &amp;&amp; lhs.p == rhs.p.</td>
<td></td>
</tr>
<tr>
<td>X::surface_state_-props::not_-equal(const DA&amp; lhs, const DA&amp; rhs)</td>
<td>bool</td>
<td>Returns: lhs.o != rhs.o</td>
<td></td>
</tr>
</tbody>
</table>

8.2.7 surfaces requirements

1. Let X be a GraphicsSurfaces type.
2. Let G be a GraphicsMath type.
3. Let IM be an object of unspecified type that contains visual data.
4. Let OU be an object of unspecified type that provides all functionality needed to display visual data on an output device and to process all operations required to create, maintain, and destroy the mechanism used to display visual data. [Example: In a windowing environment the mechanism would typically be a window. — end example]
5. Let UN be an object of unspecified type that provides all functionality needed to display visual data on an output device which does not process the operations required to create, maintain, and destroy the mechanism used to display visual data. [Note: This type lets the user draw on an existing output mechanism which the user manages. — end note]

6. The types OU and UN may be the same type.
7. The definition of an output device is provided in 15.3.8.
8. Table 16 describes the observable effects of the member functions of X::surfaces.
9. Table 14 defines the required typedef-names in X::surfaces, which are identifiers for class types capable of storing all data required to support the corresponding class template.

Table 14 — X::surfaces typedef-names

<table>
<thead>
<tr>
<th>typedef-name</th>
<th>Class data</th>
</tr>
</thead>
<tbody>
<tr>
<td>image_surface_data_type</td>
<td>basic_image_surface</td>
</tr>
<tr>
<td>output_surface_data_type</td>
<td>basic_output_surface</td>
</tr>
<tr>
<td>unmanaged_output_surface_data_type</td>
<td>basic_unmanaged_output_surface</td>
</tr>
</tbody>
</table>

10. In Table 15 and Table 16, I denotes the type image_surface_data_type, O denotes the type output_surface_data_type, U denotes the type unmanaged_output_surface_data_type, BB denotes the type basic_bounding_box<G>, BP denotes the type basic_brush_props<X>, BR denotes the type basic_brush<X>, CP denotes the type basic_clip_props<X>, D denotes the type basic_dashes<X>, DP denotes the type basic_display_point<G>, FI denotes the type basic_figure_items<X>::figure_item, IMS denotes the type basic_image_surface<X>, IP denotes the type basic_interpreted_path<X>, M denotes the type basic_matrix_2d<G>, MP denotes the type basic_mask_properties<X>, OUS denotes the type basic_output_surface<X>, P denotes the type basic_point_2d<G>, RP denotes the type basic_render_props<X>, SP denotes the type basic_stroke_props<X>, and UOS denotes the type basic_unmanaged_output_surface<X>.
11. In order to describe the observable effects of functions contained in Table 16, Table 15 describes the types contained in X as-if they possessed certain member data.

Table 15 — X::surfaces type member data

<table>
<thead>
<tr>
<th>Type</th>
<th>Member data</th>
<th>Member data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>im</td>
<td>IM</td>
</tr>
</tbody>
</table>

§ 8.2.7
Note: In the same way that `stdin`, `stdout`, and `stderr` do not specify how they meet certain requirements, the requirements set forth in Table 16 also do not specify how they meet certain requirements, most or all of which relate to the output device. — end note]

Note: Operations on objects of types `IM`, `OU`, and `UO` follow the C++ requirements regarding observable behavior (See: C++ 2017[intro.execution]). Successive operations on such objects are not observable unless and until the visual data of such objects can be observed, such as when the visual data is displayed on an output device or is written out to a file. As such, implementations that use graphics acceleration hardware can use batching and other deferred processing techniques to improve performance. — end note]

### Table 16 — Graphics surfaces requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::surfaces::max_image_dimensions()</code></td>
<td>DP</td>
<td>Returns: An object dp where dp.x() is the maximum width in pixels of the visual data of an object of type IM and dp.y() is the maximum height in pixels of the visual data of an object of type IM.</td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::max_output_dimensions()</code></td>
<td>DP</td>
<td>Returns: An object dp where dp.x() is the maximum width in pixels of the visual data of an object of type OU and dp.y() is the maximum height in pixels of the visual data of an object of type OU.</td>
<td></td>
</tr>
</tbody>
</table>
Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::surfaces::max_unmanaged_output_dimensions()</td>
<td>DP</td>
<td>Returns: An object dp where dp.x() is the maximum width in pixels of the visual data of an object of type UN and dp.y() is the maximum height in pixels of the visual data of an object of type UN.</td>
<td></td>
</tr>
<tr>
<td>X::surfaces::create_image_surface(format fmt, int w, int h)</td>
<td>I</td>
<td>Returns: An object i. Requires: fmt != format::invalid, w &gt; 0, w &lt;= max_image_dimensions().x(), h &gt; 0, and h &lt;= max_image_dimensions().y(). Postconditions: The bounds of i.im are [0, w) along the x axis and [0, h) along the y axis, the visual data format of i.im is fmt, i.fmt == fmt, and i.dm == DP(x, y). The values of the visual data of i.im are unspecified.</td>
<td></td>
</tr>
</tbody>
</table>
Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::surfaces::create_image_surface(filesystem::path f, image_file_format iff, format fmt)</td>
<td>Returns: An object i.</td>
<td>Requires: f is a file, the contents of f are valid data in the data format (15.2.1) specified by iff, the bounds of the visual data contained in the contents of f do not exceed the values returned by max_image_dimensions(), and fmt != format::invalid.</td>
<td></td>
</tr>
<tr>
<td>X::surfaces::create_image_surface(filesystem::path f, image_file_format iff, format fmt, error_code&amp; ec)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>noexcept</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

§ 8.2.7
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Continued...)</td>
<td></td>
<td></td>
<td>Throws: As specified in Error reporting (Clause 4). Error conditions: Any error that could result from trying to access ( f ), open ( f ) for reading, or reading data from ( f ). Other errors, if any, are implementation-defined.</td>
</tr>
<tr>
<td>( X::surfaces::save(I&amp; \ i, filesystem::path \ f, image_file_format \ iff) )</td>
<td>void</td>
<td>Any pending rendering and composing operations (15.3.2) on ( i.im ) are performed. The visual data of ( i.im ) is written to ( f ) in the data format specified by ( iff ).</td>
<td>Requires: ( f ) shall be a valid path to a file. It is not required that the file exist provided that the other components of the path are valid. Throws: As specified in Error reporting (Clause 4). Error conditions: Any error that could result from trying to access ( f ), open ( f ) for writing, or write data to ( f ). Other errors, if any, are implementation-defined.</td>
</tr>
<tr>
<td>( X::surfaces::format(const I&amp; \ i) ) noexcept</td>
<td>( \text{id2d::format} )</td>
<td>Returns: ( i.fmt ).</td>
<td></td>
</tr>
<tr>
<td>( X::surfaces::dimensions(const I&amp; \ i) ) noexcept</td>
<td>( \text{DP} )</td>
<td>Returns: ( i.dm ).</td>
<td></td>
</tr>
</tbody>
</table>

§ 8.2.7
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::surfaces::clear(I&amp; void i)</td>
<td>void</td>
<td>Effects: Equivalent to:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X::surfaces::paint(i,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BR(rgba_color::transparent_black),</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>nullopt, make_-</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>optional&lt;RP&gt;(antialias::none,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>M(),</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>compositing_op::clear);</td>
<td></td>
</tr>
<tr>
<td>X::surfaces::paint(I&amp; void i, const BB&amp; b, const BP&amp; bp, const RP&amp; rp, const CP&amp; cl);</td>
<td>void</td>
<td>Effects: Perform the painting operation on i.im as specified in 15.3.4. b is the source brush. bp is the brush properties. rp is the surface properties. cl is the clip properties.</td>
<td></td>
</tr>
<tr>
<td>X::surfaces::stroke(I&amp;void i, const BB&amp; b, const IP&amp; ip, const BP&amp; bp, const SP&amp; sp, const D&amp; d, const RP&amp; rp, const CP&amp; cl);</td>
<td>void</td>
<td>Effects: Perform the stroking operation on i.im as specified in 15.3.6.</td>
<td></td>
</tr>
<tr>
<td>X::surfaces::fill(I&amp; void i, const BB&amp; b, const IP&amp; ip, const BP&amp; bp, const RP&amp; rp, const CP&amp; cl);</td>
<td>void</td>
<td>Effects: Perform the filling operation on i.im as specified in 15.3.5.</td>
<td></td>
</tr>
<tr>
<td>X::surfaces::mask(I&amp; void i, const BB&amp; b, const BB&amp; m, const BP&amp; bp, const MP&amp; mp, const RP&amp; rp, const CP&amp; cl);</td>
<td>void</td>
<td>Effects: Perform the masking operation on i.im as specified in 15.3.7.</td>
<td></td>
</tr>
</tbody>
</table>
Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::surfaces::create_0 output_surface(int pw, int ph, io2d::format pfmt,</td>
<td>Returns: An object o.</td>
<td>Requires: pw &gt; 0, pw &lt;= min(max_image_dimensions().x(), max_output_dimensions().x()), ph &gt; 0, ph &lt;= min(max_image_dimensions().y(), max_output_dimensions().y()), pfmt != format::invalid, and fps &lt; 0.0f.</td>
<td></td>
</tr>
<tr>
<td>io2d::scaling scl, io2d::refresh_style rs, float fps)</td>
<td></td>
<td>Postconditions: The bounds of o.ou are [0, pw) along the x axis and [0, ph) along the y axis. The visual data format of o.ou is fmt or, if pfmt is not supported for o.ou then an implementation-defined visual data format. o.fmt is set to the format enumerator that corresponds to the visual data format of o.ou, which may be a value in X::additional_formats. o.dm == DP(pw, ph). The values of the visual data of o.ou are unspecified.</td>
<td></td>
</tr>
<tr>
<td>X::surfaces::create_0 output_surface(int pw, int ph, io2d::format pfmt,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>error_code&amp; ec, io2d::scaling scl, io2d::refresh_style rs, float fps)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>noexcept</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Continued...)</td>
<td>The bounds of o.bb.im are [0, pw) along the x axis and [0, ph) along the y axis, the visual data format of o.bb.im is pfmt, o.bb.fmt == pfmt, and o.bb.dm == DP(pw, ph). The values of the visual data of o.bb.im are unspecified. o.lb.value() == BR(rgba_color::black). o.lbp == BP(). o.sc == scaling::letterbox. o.ac == false. o.rr == false. o.rs == rs. o.dfr == fps. o.dc == nullptr. o.scc == nullptr.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remarks: Implementations may defer the creation of o.ou and o.bb until begin_show(o, ...) is called. Implementations may defer the creation of the visual data of the object contained in o.lb until it is used.
Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Continued...)</td>
<td></td>
<td>When o.ou and o.bb are created, the implementation shall ensure that the values of o.dm and o.bb.dm are set to the bounds of o.ou, and if either value changed it shall then invoke o.scc if o.scc != nullptr. <strong>Throws:</strong> As specified in Error reporting (Clause 4). <strong>Error conditions:</strong> <em>errc::not_supported</em> if creating o would exceed the maximum number of simultaneous basic_display_surface objects or combination of basic_display_surface objects and basic_unmanaged_display_surface objects supported by the implementation (See: 15.3.8).</td>
<td></td>
</tr>
</tbody>
</table>
Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::surfaces::create_output_surface(int pw, int ph, io2d::format pfmt, int pdw, int pdh, io2d::format pdfmt, io2d::scaling scl, io2d::refresh_style rs, float fps) noexcept</td>
<td>Returns: An object o. Requires: pw &gt; 0, pw &lt;= max_image_dimensions().x(), ph &gt; 0, ph &lt;= max_image_dimensions().y(), pfmt != format::invalid, pdw &gt; 0, pdh &lt;= max_output_dimensions().x(), pdh &gt; 0, pdh &lt;= max_output_dimensions().y(), pdfmt != format::invalid, and fps &lt; 0.0f. Postconditions: The bounds of o.ou are [0, pdw) along the x axis and [0, pdh) along the y axis. The visual data format of o.ou is pdfmt or, if pdfmt is not supported for o.ou then an implementation-defined visual data format. o.fmt is set to the format enumerator that corresponds to the visual data format of o.ou, which may be a value in X::additional_formats. o.dm == DP(pdw, pdh). The values of the visual data of o.ou are unspecified.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

§ 8.2.7
Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Continued...)</td>
<td></td>
<td></td>
<td>The bounds of o.bb.im are $[0, \text{pw})$ along the $x$ axis and $[0, \text{ph})$ along the $y$ axis, the visual data format of o.bb.im is pfmt, o.bb.fmt == pfmt, and o.bb.dm == DP(pw, ph). The values of the visual data of o.bb.im are unspecified. o.lb.value() == BR(rgba_\text{color}::black). o.lbp == BP(). o.sc == scaling::letterbox. o.ac == false. o.rr == false. o.rs == rs. o.dfr == fps. o.dc == nullptr. o.scc == nullptr.</td>
</tr>
</tbody>
</table>

Remarks: Implementations may defer the creation of o.ou and o.bb until begin_show(o, ...) is called. Implementations may defer the creation of the visual data of the object contained in o.lb until it is used.

§ 8.2.7
Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Continued...)</td>
<td></td>
<td>When <code>o.ou</code> and <code>o.bb</code> are created, the implementation shall ensure that the values of <code>o.dm</code> and <code>o.bb.dm</code> are set to the bounds of <code>o.ou</code>, and if either value changed it shall then invoke <code>o.scc</code> if <code>o.scc</code> != <code>nullptr</code>. Throws: As specified in Error reporting (Clause 4). Error conditions: <code>errc::not_supported</code> if creating <code>o</code> would exceed the maximum number of simultaneous <code>basic_display_surface</code> objects or combination of <code>basic_display_surface</code> objects and <code>basic_unmanaged_display_surface</code> objects supported by the implementation (See: 15.3.8).</td>
<td></td>
</tr>
</tbody>
</table>
Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::surfaces::begin_- show(O&amp; o, OUS&amp; sfc) X::surfaces::begin_- show(O&amp; o, OUS&amp; src, error_code&amp; ec)</td>
<td>noexcept</td>
<td>Performs the following actions in a continuous loop: 1. Handle any implementation and host environment matters, including updating the value of o.dm if the output device bounds have changed; then, 2. If the value of o.dm changed and o.scc != nullptr, invoke o.scc; then, 3. If o.rr == true or the values of o.rs and o.dfr require that o.dc be called: a) Set o.rr to false; then, b) If o.ac == true, invoke clear(o.bb); then, c) Invoke o.dc; then, d) Transfer o.bb.im to o.ou, performing the scaling and letterboxing, if any, required by o.sc and the color space conversion, if any, required to transform o.bb.im from o.bb.fmt to o.fmt.</td>
<td>Requires: sfc.data() == o. Throws: As specified in Error reporting (Clause 4). Error conditions: errc::not_ supported if creating or displaying o.im would exceed the maximum number of simultaneous basic_display_surface objects or combination of basic_display_surface objects supported by the implementation (See: 15.3.8).</td>
</tr>
</tbody>
</table>

§ 8.2.7
Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Continued...)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remarks: All observable effects to visual data produced as a result of steps 3-b and 3-c above are as-if they were applied to o.bb.im in those steps. Implementations may apply those observable effects directly to o.ou provided that they do so as-if the scaling, letterboxing, and color space conversion behavior specified in 3-d occurs. [Note: This allows implementations which do not wish to use a back buffer the freedom to do so. — end note]

X::surfaces::end_- void show(0& o)

Initiates the process of exiting the continuous loop resulting from the invocation of begin_show(o, ...). Implementations should follow any procedures that the host environment requires in order to stop the continuous loop without error. If the continuous loop resulting from the invocation of begin_show(o, ...) is not executing or is already exiting due to a previous call to this function, this function does nothing.

Remarks: This function shall not wait until the continuous loop from begin_show(o, ...) ends before returning. [Note: The correct way to exit the begin_show(o, ...) continuous loop is to call this function from o.dc or from another thread. — end note]
Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimensions(O&amp; o, const DP&amp; dm)</td>
<td>void</td>
<td>Attempts to change the bounds of o.bb.im to dm. If successful, o.bb.dm == dm, otherwise there are no effects.</td>
<td>Requires: dm.x() &gt; 0, dm.x() &lt;= max_image_dimensions().x(), dm.y() &gt; 0, dm.y() &lt;= max_image_dimensions().y()</td>
</tr>
<tr>
<td>output_dimensions(O&amp; o, const DP&amp; dm)</td>
<td>void</td>
<td>Attempts to change the bounds of o.ou to dm. If successful sets o.dm to the value of dm and then invokes o.scc unless o.scc != nullptr, otherwise there are no effects.</td>
<td>Requires: dm.x() &gt; 0, dm.x() &lt;= max_output_dimensions().x(), dm.y() &gt; 0, dm.y() &lt;= max_output_dimensions().y()</td>
</tr>
<tr>
<td>scaling(O&amp; o, io2d::scaling sc)</td>
<td>void</td>
<td>Postconditions: o.sc == sc.</td>
<td></td>
</tr>
<tr>
<td>refresh_style(O&amp; o, io2d::refresh_style rs)</td>
<td>void</td>
<td>Postconditions: o.rs == rs.</td>
<td></td>
</tr>
<tr>
<td>desired_frame_rate(O&amp; o, float dfr)</td>
<td>void</td>
<td>Requires: dfr &gt; 0.0f. Postconditions: o.dfr == dfr.</td>
<td></td>
</tr>
<tr>
<td>letterbox_brush(O&amp; o, const optional&lt;BB&gt;&amp; lb, const optional&lt;BP&gt;&amp; lbp)</td>
<td>void</td>
<td>Postconditions: If lb.has_value() == true then o.lb == lb.value(), otherwise o.lb == BB(rgba_color::black). If lbp.has_value() == true then o.lbp == lbp.value(), otherwise o.lbp == BP().</td>
<td></td>
</tr>
<tr>
<td>letterbox_brush_properties(O&amp; o, const optional&lt;BP&gt;&amp; lbp)</td>
<td>void</td>
<td>Postconditions: If lbp.has_value() == true then o.lbp == lbp.value(), otherwise o.lbp == BP().</td>
<td></td>
</tr>
<tr>
<td>auto_clear(O&amp; o, bool ac)</td>
<td>void</td>
<td>Postconditions: o.ac == ac.</td>
<td></td>
</tr>
<tr>
<td>redraw_required(O&amp; o, bool rr)</td>
<td>void</td>
<td>Postconditions: o.rr == rr.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>draw_callback(O&amp; o, function&lt;void(OUS&amp;)&gt;::dc)</code></td>
<td>void</td>
<td></td>
<td>Postconditions: o.dc == dc.</td>
</tr>
<tr>
<td><code>size_change_callback(O&amp; o, function&lt;void(OUS&amp;)&gt;::scc)</code></td>
<td>void</td>
<td></td>
<td>Postconditions: o.scc == scc.</td>
</tr>
<tr>
<td><code>X::surfaces::clear(O&amp; o)</code></td>
<td></td>
<td>Effects: Equivalent to: <code>paint(o.bb)</code></td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::paint(O&amp; o, const BB&amp; b, const BP&amp; bp, const RP&amp; rp, const CP&amp; cl)</code></td>
<td>void</td>
<td>Effects: Equivalent to: <code>paint(o.bb, &lt;TODO&gt;)</code></td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::stroke(I&amp; i, const BB&amp; b, const IP&amp; ip, const BP&amp; bp, const SP&amp; sp, const D&amp; d, const RP&amp; rp, const CP&amp; cl)</code></td>
<td>void</td>
<td>Perform the stroking operation on i.im as specified in 15.3.6.</td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::fill(I&amp; i, const BB&amp; b, const IP&amp; ip, const BP&amp; bp, const RP&amp; rp, const CP&amp; cl)</code></td>
<td>void</td>
<td>Perform the filling operation on i.im as specified in 15.3.5.</td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::mask(I&amp; i, const BB&amp; b, const BB&amp; m, const BP&amp; bp, const MP&amp; mp, const RP&amp; rp, const CP&amp; cl)</code></td>
<td>void</td>
<td>Perform the masking operation on i.im as specified in 15.3.5.</td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::dimensions(const O&amp; o) noexcept</code></td>
<td><code>DP</code></td>
<td>Returns: o.bb.dm.</td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::output_dimensions(const O&amp; o) noexcept</code></td>
<td><code>DP</code></td>
<td>Returns: o.dm.</td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::refresh_style(const O&amp; o)</code></td>
<td><code>io2d::refresh_style</code></td>
<td>Returns: o.rs</td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::desired_frame_rate(const O&amp; o)</code></td>
<td><code>float</code></td>
<td>Returns: o.dfr.</td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::scaling(const O&amp; o)</code></td>
<td><code>io2d::scaling</code></td>
<td>Returns: o.sc.</td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::letterbox_brush(const O&amp; o)</code></td>
<td><code>optional&lt;BB&gt;</code></td>
<td>Returns: o.lb</td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::letterbox_brush_props(const O&amp; o)</code></td>
<td><code>BP</code></td>
<td>Returns: o.lbp</td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::auto_clear(const O&amp; o)</code></td>
<td><code>bool</code></td>
<td>Returns: o.ac</td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Operational semantics</td>
<td>Assertion/note</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-------------</td>
<td>-----------------------</td>
<td>----------------</td>
</tr>
<tr>
<td><code>X::surfaces::redraw_ - required(const O&amp; o) noexcept</code></td>
<td><code>bool</code></td>
<td>Returns: <code>o.rr</code></td>
<td></td>
</tr>
<tr>
<td><code>X::surfaces::copy_ - surface(const I&amp; i)</code></td>
<td><code>IMS</code></td>
<td>Returns: An object c.</td>
<td>Postconditions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c.data().im is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a copy of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>data in i.im.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c.data().fmt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>== i.fmt.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c.data().dm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>== i.dm.</td>
</tr>
<tr>
<td><code>X::surfaces::copy_ - surface(const O&amp; o)</code></td>
<td><code>IMS</code></td>
<td>Returns: An object c.</td>
<td>Postconditions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c.data().im is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a copy of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>data in o.bb.im</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c.data().fmt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>== o.bb.fmt.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c.data().dm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>== o.bb.dm.</td>
</tr>
</tbody>
</table>
Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::surfaces::create_unmanaged_output_surface(/* implementation-defined */)</td>
<td></td>
<td>All details of this function other than its name and return type are implementation-defined. It is not required that this function be provided by an implementation. This function may be overloaded.</td>
<td></td>
</tr>
<tr>
<td>template &lt;class F&gt; void X::surfaces::draw_callback(UN&amp; un, F&amp;&amp; f)</td>
<td>void</td>
<td>&lt;TODO&gt;</td>
<td>Requires: f shall be CopyConstructible.</td>
</tr>
</tbody>
</table>

[Note: This function exists to allow users to take an existing output device, such as a window or a smart phone display, and draw to it using this library via the basic_unmanaged_output_surface class template. Implementers are not required to support this functionality; among other reasons, it may be impossible to provide it on certain platforms. If this function is not provided, it is impossible for the basic_unmanaged_output_surface class template to be instantiated. — end note]
Table 16 — Graphics surfaces requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

§ 8.2.7
9 Linear algebra

9.1 Class basic_point_2d

9.1.1 basic_point_2d description

1 The class template basic_point_2d is used as both a point and as a two-dimensional Euclidean vector.

2 It has an \(x\) coordinate of type float and a \(y\) coordinate of type float.

3 The data are stored in an object of type typename GraphicsMath::point_2d_data_type. It is accessible using the data member functions.

9.1.2 basic_point_2d synopsis

namespace std::experimental::io2d::v1 {
  template <class GraphicsMath>
  class basic_point_2d {
    public:
      using data_type = typename GraphicsMath::point_2d_data_type;

      // 9.1.3, constructors:
      basic_point_2d() noexcept;
      basic_point_2d(float x, float y) noexcept;
      basic_point_2d(const typename GraphicsMath::point_2d_data_type& data) noexcept;

      // 9.1.4, accessors:
      const data_type& data() const noexcept;
      data_type& data() noexcept;

      // 9.1.5, modifiers:
      void x(float val) noexcept;
      void y(float val) noexcept;

      // 9.1.6, observers:
      float x() const noexcept;
      float y() const noexcept;
      float dot(const basic_point_2d& other) const noexcept;
      float magnitude() const noexcept;
      float magnitude_squared() const noexcept;
      float angular_direction() const noexcept;
      basic_point_2d to_unit() const noexcept;

      // 9.1.7, member operators:
      basic_point_2d& operator+=(const basic_point_2d& rhs) noexcept;
      basic_point_2d& operator+=(float rhs) noexcept;
      basic_point_2d& operator-=(const basic_point_2d& rhs) noexcept;
      basic_point_2d& operator-=(float rhs) noexcept;
      basic_point_2d& operator*=(const basic_point_2d& rhs) noexcept;
      basic_point_2d& operator*=(float rhs) noexcept;
      basic_point_2d& operator/=(const basic_point_2d& rhs) noexcept;
      basic_point_2d& operator/=(float rhs) noexcept;
    }

  };
template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator+(const basic_point_2d<GraphicsMath>& val) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator+(const basic_point_2d<GraphicsMath>& lhs,
const basic_point_2d<GraphicsMath>& rhs) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator-(const basic_point_2d<GraphicsMath>& val) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator-(const basic_point_2d<GraphicsMath>& lhs,
const basic_point_2d<GraphicsMath>& rhs) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator*(const basic_point_2d<GraphicsMath>& lhs,
float rhs) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator*(float lhs,
const basic_point_2d<GraphicsMath>& rhs) noexcept;

template <class GraphicsMath>
basic_point_2d<GraphicsMath> operator*(const basic_point_2d<GraphicsMath>& lhs,
const basic_point_2d<GraphicsMath>& rhs) noexcept;

} 9.1.3 basic_point_2d constructors [io2d.point2d.cons]

basic_point_2d() noexcept;

Effects: Constructs an object of type basic_point_2d.

Postconditions: data() == GraphicsMath::create_point_2d().

basic_point_2d(float x, float y) noexcept;

Effects: Constructs an object of type basic_point_2d.

Postconditions: data() == GraphicsMath::create_point_2d(x, y).

basic_point_2d(const data_type& d) noexcept;

Effects: Constructs an object of type basic_point_2d.

Postconditions: data() == d.

9.1.4 basic_point_2d accessors [io2d.point2d.accessors]

const data_type& data() const noexcept;

data_type& data() noexcept;

Returns: A reference to the basic_point_2d object’s data object (See: 9.1.1).
9.1.5 basic_point_2d modifiers

```cpp
void x(float val) noexcept;

Effects: Equivalent to GraphicsMath::x(data(), val);
```

```cpp
void y(float val) noexcept;

Effects: Equivalent to GraphicsMath::y(data(), val);
```

9.1.6 basic_point_2d observers

```cpp
float x() const noexcept;

Returns: GraphicsMath::x(data()).
```

```cpp
float y() const noexcept;

Returns: GraphicsMath::y(data()).
```

```cpp
float dot(const basic_point_2d& other) const noexcept;

Returns: GraphicsMath::dot(data(), other).
```

```cpp
float magnitude() const noexcept;

Returns: GraphicsMath::magnitude(data()).
```

```cpp
float magnitude_squared() const noexcept;

Returns: GraphicsMath::magnitude_squared(data()).
```

```cpp
float angular_direction() const noexcept;

Returns: GraphicsMath::angular_direction(data()).
```

```cpp
basic_point_2d to_unit() const noexcept;

Returns: basic_point_2d(GraphicsMath::to_unit(data())).
```

9.1.7 basic_point_2d member operators

```cpp
basic_point_2d& operator+=(const basic_point_2d& rhs) noexcept;

Effects: Equivalent to data() = GraphicsMath::add(data(), rhs.data());

Returns: *this.
```

```cpp
basic_point_2d& operator-=(const basic_point_2d& rhs) noexcept;

Effects: Equivalent to data() = GraphicsMath::subtract(data(), rhs.data());

Returns: *this.
```

```cpp
basic_point_2d& operator*=(float rhs) noexcept;

Effects: Equivalent to data() = GraphicsMath::multiply(data(), rhs);

Returns: *this.
```

```cpp
basic_point_2d& operator*=(const basic_point_2d& rhs) noexcept;

Effects: Equivalent to data() = GraphicsMath::multiply(data(), rhs.data());

Returns: *this.
```

```cpp
basic_point_2d& operator/=(float rhs) noexcept;

Effects: Equivalent to: data() = GraphicsMath::divide(data(), rhs);

Returns: *this.
```

```cpp
basic_point_2d& operator/=(const basic_point_2d& rhs) noexcept;

Effects: Equivalent to: data() = GraphicsMath::divide(data(), rhs.data());

Returns: *this.
```
9.1.8 basic_point_2d non-member operators

bool operator==(const basic_point_2d& lhs, const basic_point_2d& rhs) noexcept;
Returns: GraphicsMath::equal(lhs.data(), rhs.data()).

bool operator!=(const basic_point_2d& lhs, const basic_point_2d& rhs) noexcept;
Returns: GraphicsMath::not_equal(lhs.data(), rhs.data()).

basic_point_2d operator+(const basic_point_2d& val) noexcept;
Returns: val.

basic_point_2d operator+(const basic_point_2d& lhs, const basic_point_2d& rhs) noexcept;
Returns: basic_point_2d(GraphicsMath::add(lhs.data(), rhs.data())).

basic_point_2d operator-(const basic_point_2d& val) noexcept;
Returns: basic_point_2d(GraphicsMath::negate(val.data())).

basic_point_2d operator-(const basic_point_2d& lhs, const basic_point_2d& rhs) noexcept;
Returns: basic_point_2d(GraphicsMath::subtract(lhs.data(), rhs.data())).

basic_point_2d operator*(const basic_point_2d& lhs, const basic_point_2d& rhs) noexcept;
Returns: basic_point_2d(GraphicsMath::multiply(lhs.data(), rhs.data())).

basic_point_2d operator*(const basic_point_2d& lhs, float rhs) noexcept;
Returns: basic_point_2d(GraphicsMath::multiply(lhs.data(), rhs)).

basic_point_2d operator*(float lhs, const basic_point_2d& rhs) noexcept;
Returns: basic_point_2d(GraphicsMath::multiply(lhs, rhs.data())).

basic_point_2d operator/(const basic_point_2d& lhs, const basic_point_2d& rhs) noexcept;
Requires: rhs.x() != 0.0f and rhs.y() != 0.0f.
Returns: basic_point_2d(GraphicsMath::divide(lhs.data(), rhs.data())).

basic_point_2d operator/(const basic_point_2d& lhs, float rhs) noexcept;
Requires: rhs != 0.0f.
Returns: basic_point_2d(GraphicsMath::divide(lhs, rhs.data())).

basic_point_2d operator/(float lhs, const basic_point_2d& rhs) noexcept;
Requires: rhs.x() != 0.0f and rhs.y() != 0.0f.
Returns: basic_point_2d(GraphicsMath::divide(lhs, rhs.data())).

9.2 Class basic_matrix_2d

9.2.1 basic_matrix_2d description

The class template basic_matrix_2d represents a three row by three column matrix. Its purpose is to perform affine transformations.

The matrix is composed of nine float values: \( m00, m01, m02, m10, m11, m12, m20, m21, \) and \( m22. \) The ordering of these float values in the basic_matrix_2d class is unspecified.

The specification of the basic_matrix_2d class, as described in this subclause, uses the following ordering:
\[
\begin{bmatrix}
  m00 & m01 & m02 \\
  m10 & m11 & m12 \\
  m20 & m21 & m22 \\
\end{bmatrix}
\]

[Note: The naming convention and the layout shown above are consistent with a row-major layout. Though the naming convention is fixed, the unspecified layout allows for a column-major layout (or any other layout, though row-major and column-major are the only layouts typically used). — end note]

The performance of any mathematical operation upon a basic_matrix_2d shall be carried out as-if the omitted third column data members were present with the values prescribed in the previous paragraph. 

§ 9.2.1
The data are stored in an object of type `typename GraphicsMath::matrix_2d_data_type`. It is accessible using the `data` member functions.

### 9.2.2 basic_matrix_2d synopsis

```cpp
namespace std::experimental::io2d::v1 {
    template <class GraphicsMath>
    class basic_matrix_2d {
        public:
            using data_type = typename GraphicsMath::matrix_2d_data_type;

            // 9.2.3, constructors:
            basic_matrix_2d() noexcept;
            basic_matrix_2d(float v00, float v01, float v10, float v11, float v20, float v21) noexcept;
            basic_matrix_2d(const typename GraphicsMath::matrix_2d_data_type& v) noexcept;

            // 9.2.4, accessors:
            const data_type& data() const noexcept;
            data_type& data() noexcept;

            // 9.2.5, static factory functions:
            static basic_matrix_2d init_translate(const basic_point_2d<GraphicsMath>& v) noexcept;
            static basic_matrix_2d init_scale(const basic_point_2d<GraphicsMath>& v) noexcept;
            static basic_matrix_2d init_rotate(float radians) noexcept;
            static basic_matrix_2d init_rotate(float radians, const basic_point_2d<GraphicsMath>& origin) noexcept;
            static basic_matrix_2d init_reflect(float radians) noexcept;
            static basic_matrix_2d init_shear_x(float factor) noexcept;
            static basic_matrix_2d init_shear_y(float factor) noexcept;

            // 9.2.6, modifiers:
            void m00(float v) noexcept;
            void m01(float v) noexcept;
            void m10(float v) noexcept;
            void m11(float v) noexcept;
            void m20(float v) noexcept;
            void m21(float v) noexcept;
            basic_matrix_2d& translate(const basic_point_2d<GraphicsMath>& v) noexcept;
            basic_matrix_2d& scale(const basic_point_2d<GraphicsMath>& v) noexcept;
            basic_matrix_2d& rotate(float radians) noexcept;
            basic_matrix_2d& rotate(float radians, const basic_point_2d<GraphicsMath>& origin) noexcept;
            basic_matrix_2d& reflect(float radians) noexcept;
            basic_matrix_2d& shear_x(float factor) noexcept;
            basic_matrix_2d& shear_y(float factor) noexcept;

            // 9.2.7, observers:
            float m00() const noexcept;
            float m01() const noexcept;
            float m10() const noexcept;
            float m11() const noexcept;
            float m20() const noexcept;
            float m21() const noexcept;
            bool is_finite() const noexcept;
            bool is_invertible() const noexcept;
            float determinant() const noexcept;
            basic_matrix_2d inverse() const noexcept;
            basic_point_2d<GraphicsMath> transform_pt(const basic_point_2d<GraphicsMath>& pt) const noexcept;

            // 9.2.8, member operators:
            basic_matrix_2d& operator*=(const basic_matrix_2d& other) noexcept;
    };
}[
```
9.2.3 basic_matrix_2d constructors

basic_matrix_2d() noexcept;

Effects: Constructs an object of type basic_matrix_2d.

Postconditions: data() == GraphicsMath::create_matrix_2d().

basic_matrix_2d(float m00, float m01, float m10, float m11,
float m20, float m21) noexcept;

Effects: Constructs an object of type basic_matrix_2d.

Postconditions: data() == GraphicsMath::create_matrix_2d(m00, m01, m10, m11, m20, m21).

basic_matrix_2d(const data_type& v) noexcept;

Effects: Constructs an object of type basic_matrix_2d.

Postconditions: data() == v.

9.2.4 basic_matrix_2d accessors

const data_type& data() const noexcept;
data_type& data() noexcept;

Returns: A reference to the basic_matrix_2d object’s data object (See: 9.2.1).

9.2.5 basic_matrix_2d static factory functions

static basic_matrix_2d init_translate(basic_point_2d<GraphicsMath> v) noexcept;

Returns: basic_matrix_2d(GraphicsMath::create_translate(v.data())).

static basic_matrix_2d init_scale(basic_point_2d<GraphicsMath> v) noexcept;

Returns: basic_matrix_2d(GraphicsMath::create_scale(v.data())).

static basic_matrix_2d init_rotate(float radians) noexcept;

Returns: basic_matrix_2d(GraphicsMath::create_rotate(radians)).

static basic_matrix_2d init_rotate(float radians, basic_point_2d<GraphicsMath> origin) noexcept;

Returns: basic_matrix_2d(GraphicsMath::create_rotate(radians, origin.data())).

static basic_matrix_2d init_reflect(float radians) noexcept;

static basic_matrix_2d init_shear_x(float factor) noexcept;

Returns: basic_matrix_2d(GraphicsMath::create_shear_x(factor)).
static basic_matrix_2d init_shear_y(float factor) noexcept;

Returns: basic_matrix_2d(GraphicsMath::create_shear_y(factor)).

9.2.6 basic_matrix_2d modifiers

void m00(float v) noexcept;
Effects: Equivalent to GraphicsMath::m00(data(), v);

void m01(float v) noexcept;
Effects: Equivalent to GraphicsMath::m01(data(), v);

void m10(float v) noexcept;
Effects: Equivalent to GraphicsMath::m10(data(), v);

void m11(float v) noexcept;
Effects: Equivalent to GraphicsMath::m11(data(), v);

void m20(float v) noexcept;
Effects: Equivalent to GraphicsMath::m20(data(), v);

void m21(float v) noexcept;
Effects: Equivalent to GraphicsMath::m21(data(), v);

basic_matrix_2d& translate(basic_point_2d<GraphicsMath> v) noexcept;
Effects: Equivalent to data() = GraphicsMath::translate(data(), v.data());
Returns: *this.

basic_matrix_2d& scale(basic_point_2d<GraphicsMath> v) noexcept;
Effects: Equivalent to data() = GraphicsMath::scale(data(), v.data());
Returns: *this.

basic_matrix_2d& rotate(float radians) noexcept;
Effects: Equivalent to data() = GraphicsMath::rotate(data(), radians);
Returns: *this.

basic_matrix_2d& rotate(float radians, basic_point_2d<GraphicsMath> origin) noexcept;
Effects: Equivalent to data() = GraphicsMath::rotate(data(), radians, origin.data());
Returns: *this.

basic_matrix_2d& reflect(float radians) noexcept;
Effects: Equivalent to data() = GraphicsMath::reflect(data(), radians);
Returns: *this.

basic_matrix_2d& shear_x(float factor) noexcept;
Effects: Equivalent to data() = GraphicsMath::shear_x(data(), factor);
Returns: *this.

basic_matrix_2d& shear_y(float factor) noexcept;
Effects: Equivalent to data() = GraphicsMath::shear_y(factor);
Returns: *this.

9.2.7 basic_matrix_2d observers

float m00() const noexcept;
Returns: GraphicsMath::m00(data()).
float m01() const noexcept;
   Returns: GraphicsMath::m01(data()).

float m10() const noexcept;
   Returns: GraphicsMath::m10(data()).

float m11() const noexcept;
   Returns: GraphicsMath::m11(data()).

float m20() const noexcept;
   Returns: GraphicsMath::m20(data()).

float m21() const noexcept;
   Returns: GraphicsMath::m21(data()).

bool is_finite() const noexcept;
   Returns: GraphicsMath::is_finite(data()).

bool is_invertible() const noexcept;
   Requires: is_finite() == true.
   Returns: GraphicsMath::is_invertible(data()).

basic_matrix_2d inverse() const noexcept;
   Requires: is_invertible() == true.
   Returns: basic_matrix_2d(GraphicsMath::inverse(data())).

float determinant() const noexcept;
   Requires: is_finite() == true.
   Returns: GraphicsMath::determinant(data()).

basic_point_2d<GraphicsMath> transform_pt(basic_point_2d<GraphicsMath> pt) const noexcept;
   Returns: basic_point_2d<GraphicsMath>(GraphicsMath::transform_pt(data(), pt.data())).

9.2.8 basic_matrix_2d member operators [io2d.matrix2d.member.ops]

basic_matrix_2d& operator*=(const basic_matrix_2d& rhs) noexcept;
   Effects: Equivalent to data() = GraphicsMath::multiply(data(), rhs.data());
   Returns: *this.

9.2.9 basic_matrix_2d non-member operators [io2d.matrix2d.ops]

basic_matrix_2d operator*(const basic_matrix_2d& lhs, const basic_matrix_2d& rhs)
   noexcept;
   Returns: basic_matrix_2d(GraphicsMath::multiply(lhs.data(), rhs.data())).

basic_point_2d<GraphicsMath> operator*(basic_point_2d<GraphicsMath> v, const basic_matrix_2d& m)
   noexcept;
   Returns: Equivalent to m.transform_pt(v).

bool operator==(const basic_matrix_2d& lhs, const basic_matrix_2d& rhs) noexcept;
   Returns: GraphicsMath::equal(lhs.data(), rhs.data()).

bool operator!=(const basic_matrix_2d& lhs, const basic_matrix_2d& rhs) noexcept;
   Returns: Equivalent to GraphicsMath::not_equal(lhs.data(), rhs.data()).
10  Geometry

10.1  Class template basic_display_point

10.1.1  basic_display_point description

1 The class template basic_display_point describes an integral point used to describe certain properties of surfaces.

2 It has an x coordinate of type int and a y coordinate of type int.

3 The data are stored in an object of type typename GraphicsMath::display_point_data_type. It is accessible using the data member functions.

10.1.2  basic_display_point synopsis

namespace std::experimental::io2d::v1 {
    template <class GraphicsMath>
    class basic_display_point {
    public:
        using data_type = typename GraphicsMath::display_point_data_type;

        // 10.1.3, constructors:
        basic_display_point() noexcept;
        basic_display_point(int x, int y) noexcept;
        basic_display_point(const data_type& val) noexcept;

        // 10.1.4, accessors:
        const data_type& data() const noexcept;
        data_type& data() noexcept;

        // 10.1.5, modifiers:
        void x(int val) noexcept;
        void y(int val) noexcept;

        // 10.1.6, observers:
        int x() const noexcept;
        int y() const noexcept;
    };

    // 10.1.7, operators:
    template <class GraphicsMath>
    bool operator==(const basic_display_point<GraphicsMath>& lhs, const basic_display_point<GraphicsMath>& rhs) noexcept;
    template <class GraphicsMath>
    bool operator!=(const basic_display_point<GraphicsMath>& lhs, const basic_display_point<GraphicsMath>& rhs) noexcept;
}

10.1.3  basic_display_point constructors

basic_display_point() noexcept;

1 Effects: Constructs an object of type basic_display_point.

2 Postconditions: data() == GraphicsMath::create_display_point().

3 Remarks: The x coordinate is 0 and the y coordinate is 0.

basic_display_point(int x, int y) noexcept;

4 Effects: Constructs an object of type basic_display_point.

5 Postconditions: data() == GraphicsMath::create_display_point(x, y).

6 Remarks: The x coordinate is x and the y coordinate is y.
basic_display_point(const data_type& val) noexcept;

Effects: Constructs an object of type basic_display_point.
Postconditions: data() == val.
Remarks: The x coordinate is GraphicsMath::x(val) and the y coordinate is GraphicsMath::y(val).

10.1.4 basic_display_point accessors

const data_type& data() const noexcept;
data_type& data() noexcept;

Returns: A reference to the basic_display_point object’s data object (See: 10.1.1).

10.1.5 basic_display_point modifiers

void x(int v) noexcept;
Effects: Equivalent to GraphicsMath::x(data(), v);
void y(int v) noexcept;
Effects: Equivalent to GraphicsMath::y(data(), v);

10.1.6 basic_display_point observers

int x() const noexcept;
Returns: GraphicsMath::x(data()).
int y() const noexcept;
Returns: GraphicsMath::y(data()).

10.1.7 basic_display_point operators

bool operator==(const basic_display_point<GraphicsMath>& lhs, const basic_display_point<GraphicsMath>& rhs) noexcept;
Returns: GraphicsMath::equal(lhs.data(), rhs.data()).
bool operator!=(const basic_display_point<GraphicsMath>& lhs, const basic_display_point<GraphicsMath>& rhs) noexcept;
Returns: GraphicsMath::not_equal(lhs.data(), rhs.data()).

10.2 Class basic_bounding_box

10.2.1 basic_bounding_box description

The class template basic_bounding_box describes a bounding_box.
It has an x coordinate of type float, a y coordinate of type float, a width of type float, and a height of type float.
The data are stored in an object of type typename GraphicsMath::bounding_box_data_type. It is accessible using the data member functions.

10.2.2 basic_bounding_box synopsis

namespace std::experimental::io2d::v1 {
template <class GraphicsMath>
class basic_bounding_box {
public:
using data_type = typename GraphicsMath::bounding_box_data_type;

// 10.2.3, constructors:
basic_bounding_box() noexcept;
basic_bounding_box(float x, float y, float width, float height) noexcept;
basic_bounding_box(const basic_point_2d<GraphicsMath>& tl, const basic_point_2d<GraphicsMath>& br) noexcept;
basic_bounding_box(const data_type& val) noexcept;

§ 10.2.2 81
// 10.2.4, accessors:
const data_type& data() const noexcept;
data_type& data() noexcept;

// 10.2.5, modifiers:
void x(float val) noexcept;
void y(float val) noexcept;
void width(float val) noexcept;
void height(float val) noexcept;
void top_left(const basic_point_2d<GraphicsMath>& val) noexcept;
void bottom_right(const basic_point_2d<GraphicsMath>& val) noexcept;

// 10.2.6, observers:
float x() const noexcept;
float y() const noexcept;
float width() const noexcept;
float height() const noexcept;
basic_point_2d<GraphicsMath> top_left() const noexcept;
basic_point_2d<GraphicsMath> bottom_right() const noexcept;
};

// 10.2.7, operators:
template <class GraphicsMath>
bool operator==(const basic_bounding_box<GraphicsMath>& lhs,
const basic_bounding_box<GraphicsMath>& rhs) noexcept;
template <class GraphicsMath>
bool operator!=(const basic_bounding_box<GraphicsMath>& lhs,
const basic_bounding_box<GraphicsMath>& rhs) noexcept;
}

10.2.3  basic_bounding_box constructors [io2d.bounding_box.cons]
basic_bounding_box() noexcept;
1   Effects: Constructs an object of type basic_bounding_box.
2   Postconditions: data() == GraphicsMath::create_bounding_box().
basic_bounding_box(float x, float y, float w, float h) noexcept;
3   Requires: w is not less than 0.0f and h is not less than 0.0f.
4   Effects: Constructs an object of type basic_bounding_box.
5   Postconditions: data() == GraphicsMath::create_bounding_box(x, y, w, h).
basic_bounding_box(const basic_point_2d<GraphicsMath>& tl,
const basic_point_2d<GraphicsMath>& br) noexcept;
6   Effects: Constructs an object of type basic_bounding_box.
7   Postconditions: data() == GraphicsMath::create_bounding_box(tl.data(), br.data()).
basic_bounding_box(const data_type& val) noexcept;
8   Effects: Constructs an object of type basic_bounding_box.
9   Postconditions: data() == val.

10.2.4  basic_bounding_box accessors [io2d.bounding_box.accessors]
const data_type& data() const noexcept;
data_type& data() noexcept;
1   Returns: A reference to the basic_bounding_box object’s data object (See: 10.2.1).

10.2.5  basic_bounding_box modifiers [io2d.bounding_box.modifiers]
void x(float v) noexcept;
1   Effects: Equivalent to GraphicsMath::x(data(), v);
void y(float v) noexcept;  
  Effects: Equivalent to GraphicsMath::y(data(), v);

void width(float v) noexcept;  
  Effects: Equivalent to GraphicsMath::width(data(), v);

void height(float val) noexcept;  
  Effects: Equivalent to GraphicsMath::height(data(), v);

void top_left(const basic_point_2d<GraphicsMath>& v) noexcept;  
  Effects: Equivalent to GraphicsMath::top_left(data(), v.data());

void bottom_right(const basic_point_2d<GraphicsMath>& v) noexcept;  
  Effects: Equivalent to GraphicsMath::bottom_right(data(), v.data());

10.2.6 basic_bounding_box observers

float x() const noexcept;  
  Returns: GraphicsMath::x(data());

float y() const noexcept;  
  Returns: GraphicsMath::y(data());

float width() const noexcept;  
  Returns: GraphicsMath::width(data());

float height() const noexcept;  
  Returns: GraphicsMath::height(data());

basic_point_2d<GraphicsMath> top_left() const noexcept;  
  Returns: basic_point_2d<GraphicsMath>(GraphicsMath::top_left(data()));

basic_point_2d<GraphicsMath> bottom_right() const noexcept;  
  Returns: basic_point_2d<GraphicsMath>(GraphicsMath::bottom_right(data()));

10.2.7 basic_bounding_box operators

bool operator==(const basic_bounding_box<GraphicsMath>& lhs,  
const basic_bounding_box<GraphicsMath>& rhs) noexcept;  
  Returns: GraphicsMath::equal(lhs.data(), rhs.data());

bool operator!=(const basic_bounding_box<GraphicsMath>& lhs,  
const basic_bounding_box<GraphicsMath>& rhs) noexcept;  
  Returns: GraphicsMath::not_equal(lhs.data(), rhs.data());

10.3 Class basic_circle

10.3.1 basic_circle description

The class template basic_circle describes a circle.  
It has a center of type basic_point_2d<GraphicsMath> and a radius of type float.  
The data are stored in an object of type typename GraphicsMath::circle_data_type. It is accessible using the data member functions.

10.3.2 basic_circle synopsis

namespace std::experimental::io2d::v1 {
  template <class GraphicsMath>
  class basic_circle {
    public:
      using data_type = typename GraphicsMath::circle_data_type;

§ 10.3.2
// 10.3.3, constructors:
basic_circle() noexcept;
basic_circle(const basic_point_2d<GraphicsMath>& ctr, float rad) noexcept;
basic_circle(const typename GraphicsMath::circle_data_type& val) noexcept;

// 10.3.4, accessors:
const data_type& data() const noexcept;
data_type& data() noexcept;

// 10.3.5, modifiers:
void center(const basic_point_2d<GraphicsMath>& ctr) noexcept;
void radius(float r) noexcept;

// 10.3.6, observers:
basic_point_2d<GraphicsMath> center() const noexcept;
float radius() const noexcept;
};

// 10.3.7, operators:
template <class GraphicsMath>
bool operator==(const basic_circle<GraphicsMath>& lhs, const basic_circle<GraphicsMath>& rhs) noexcept;
template <class GraphicsMath>
bool operator!=(const basic_circle<GraphicsMath>& lhs, const basic_circle<GraphicsMath>& rhs) noexcept;

10.3.3 basic_circle constructors
basic_circle() noexcept;
  Effects: Constructs an object of type basic_circle.
  Postconditions: data() == GraphicsMath::create_circle().

basic_circle(const basic_point_2d<GraphicsMath>& ctr, float rad) noexcept;
  Requires: r >= 0.0f.
  Effects: Constructs an object of type basic_circle.
  Postconditions: data() == GraphicsMath::create_circle(ctr, r).

10.3.4 basic_circle accessors
const data_type& data() const noexcept;
data_type& data() noexcept;
  Returns: A reference to the basic_circle object’s data object (See: 10.3.1).

10.3.5 basic_circle modifiers
void center(const basic_point_2d<GraphicsMath>& ctr) noexcept;
  Effects: Equivalent to GraphicsMath::center(data(), ctr.data());

void radius(float r) noexcept;
  Requires: r >= 0.0f.
  Effects: Equivalent to GraphicsMath::radius(data(), r);

10.3.6 basic_circle observers
basic_point_2d<GraphicsMath> center() const noexcept;
  Returns: (basic_point_2d<GraphicsMath>(GraphicsMath::center(data()))).

float radius() const noexcept;
  Returns: GraphicsMath::radius(data()).
### 10.3.7 basic_circle operators

#### bool operator==(const basic_circle<GraphicsMath>& lhs, const basic_circle<GraphicsMath>& rhs) noexcept;

1. **Returns:** GraphicsMath::equal(lhs.data(), rhs.data()).

#### bool operator!=(const basic_circle<GraphicsMath>& lhs, const basic_circle<GraphicsMath>& rhs) noexcept;

2. **Returns:** GraphicsMath::not_equal(lhs.data(), rhs.data()).
11 Text rendering and display [io2d.text]

1 [Note: Text rendering and matters related to it, such as font support, will be added at a later date. This section is a placeholder. The integration of text rendering is expected to result in the addition of member functions to the surface class templates and changes to other parts of the text. — end note]
12 Paths

12.1 Overview of paths

Paths define geometric objects which can be stroked (Table 33), filled, and used to define a clip area (See: 14.11.1).

A path contains zero or more figures.

A figure is composed of at least one segment.

A figure may contain degenerate segments. When a path is interpreted (12.3.16), degenerate segments are removed from figures. [Note: If a path command exists or is inserted between segments, it’s possible that points which might have compared equal will no longer compare equal as a result of interpretation (12.3.16). —end note]

Paths provide vector graphics functionality. As such they are particularly useful in situations where an application is intended to run on a variety of platforms whose output devices (??) span a large gamut of sizes, both in terms of measurement units and in terms of a horizontal and vertical pixel count, in that order.

A basic_interpreted_path object is an immutable resource wrapper containing a path (12.4). A basic_interpreted_path object is created by interpreting the path contained in a basic_path_builder object. It can also be default constructed, in which case the basic_interpreted_path object contains no figures. [Note: basic_interpreted_path objects provide significant optimization opportunities for implementations. Because they are immutable and opaque, they are intended to be used to store a path in the most efficient representation available. —end note]

12.2 Path examples (Informative)

12.2.1 Overview

Paths are composed of zero or more figures. The following examples show the basics of how paths work in practice.

Every example is placed within the following code at the indicated spot. This code is shown here once to avoid repetition:

```cpp
#include <experimental/io2d>
using namespace std;
using namespace std::experimental::io2d;

int main() {
    auto imgSfc = make_image_surface(format::argb32, 300, 200);
    brush backBrush{ rgba_color::black };
    brush foreBrush{ rgba_color::white };
    render_props aliased{ antialias::none };
    path_builder pb{};
    imgSfc.paint(backBrush);

    // Example code goes here.

    imgSfc.save(filesystem::path("example.png"), image_file_format::png);
    return 0;
}
```

12.2.2 Example 1

Example 1 consists of a single figure, forming a trapezoid:

```cpp
pb.new_figure({ 80.0f, 20.0f }); // Begins the figure.
pb.line({ 220.0f, 20.0f }); // Creates a line from the [80, 20] to [220, 20].
pb.rel_line({ 60.0f, 160.0f }); // Line from [220, 20] to
```
12.2.3 Example 2

Example 2 consists of two figures. The first is a rectangular open figure (on the left) and the second is a rectangular closed figure (on the right):

```cpp
pb.new_figure({ 20.0f, 20.0f }); // Begin the first figure.
pb.rel_line({ 100.0f, 0.0f });
pb.rel_line({ 0.0f, 160.0f });
pb.rel_line({ -100.0f, 0.0f });
pb.rel_line({ 0.0f, -160.0f });
pb.new_figure({ 180.0f, 20.0f }); // End the first figure and begin the second figure.
pb.rel_line({ 100.0f, 0.0f });
pb.rel_line({ 0.0f, 160.0f });
pb.rel_line({ -100.0f, 0.0f });
pb.close_figure(); // End the second figure.
imgSfc.stroke(foreBrush, pb, nullopt, stroke_props{ 10.0f }, nullopt, aliased);
```
The resulting image from example 2 shows the difference between an open figure and a closed figure. Each figure begins and ends at the same point. The difference is that with the closed figure, that the rendering of the point where the initial segment and final segment meet is controlled by the `line_join` value in the `stroke_props` class, which in this case is the default value of `line_join::miter`. In the open figure, the rendering of that point receives no special treatment such that each segment at that point is rendered using the `line_cap` value in the `stroke_props` class, which in this case is the default value of `line_cap::none`.

That difference between rendering as a `line_join` versus rendering as two `line_caps` is what causes the notch to appear in the open segment. Segments are rendered such that half of the stroke width is rendered on each side of the point being evaluated. With no line cap, each segment begins and ends exactly at the point specified.

So for the open figure, the first line begins at `point_2d{ 20.0f, 20.0f }` and the last line ends there. Given the stroke width of 10.0f, the visible result for the first line is a rectangle with an upper left corner of `point_2d{ 20.0f, 15.0f }` and a lower right corner of `point_2d{ 120.0f, 25.0f }`. The last line appears as a rectangle with an upper left corner of `point_2d{ 15.0f, 20.0f }` and a lower right corner of `point_2d{ 25.0f, 180.0f }`. This produces the appearance of a square gap between `point_2d{ 15.0f, 15.0f }` and `point_2d{ 20.0f, 20.0f }`.

For the closed figure, adjusting for the coordinate differences, the rendering facts are the same as for the open figure except for one key difference: the point where the first line and last line meet is rendered as a line join rather than two line caps, which, given the default value of `line_join::miter`, produces a miter, adding that square area to the rendering result.

### 12.2.4 Example 3

Example 3 demonstrates open and closed figures each containing either a quadratic curve or a cubic curve.

```cpp
pb.new_figure({ 20.0f, 20.0f });
pb.rel_quadratic_curve({ 60.0f, 120.0f }, { 60.0f, -120.0f });
pb.rel_new_figure({ 20.0f, 0.0f });
pb.rel_quadratic_curve({ 60.0f, 120.0f }, { 60.0f, -120.0f });
pb.close_figure();
pb.new_figure({ 20.0f, 150.0f });
pb.rel_cubic_curve({ 40.0f, -120.0f }, { 40.0f, 120.0f * 2.0f },
{ 40.0f, -120.0f });
pb.rel_new_figure({ 20.0f, 0.0f });
pb.rel_cubic_curve({ 40.0f, -120.0f }, { 40.0f, 120.0f * 2.0f },
{ 40.0f, -120.0f });
pb.close_figure();
imgSfc.stroke(foreBrush, pb, nullopt, nullopt, nullopt, aliased);
```
12.2.5 Example 4

Example 4 shows how to draw 'C++' using figures.

For the 'C', it is created using an arc. A scaling matrix is used to make it slightly elliptical. It is also desirable that the arc has a fixed center point, point_2d{ 85.0f, 100.0f }. The inverse of the scaling matrix is used in combination with the point_for_angle function to determine the point at which the arc should begin in order to get achieve this fixed center point. The 'C' is then stroked.

Unlike the 'C', which is created using an open figure that is stroked, each '+' is created using a closed figure that is filled. To avoid filling the 'C', pb.clear(); is called to empty the container. The first '+' is created using a series of lines and is then filled.

Taking advantage of the fact that path_builder is a container, rather than create a brand new figure for the second '+', a translation matrix is applied by inserting a figure_items::change_matrix figure item before the figure_items::new_figure object in the existing plus, reverting back to the old matrix immediately after the and then filling it again.

```cpp
// Create the 'C'.
const matrix_2d scl = matrix_2d::init_scale({ 0.9f, 1.1f });
auto pt = scl.inverse().transform_pt({ 85.0f, 100.0f }) +
    point_for_angle(half_pi<float> / 2.0f, 50.0f);
pb.matrix(scl);
pb.new_figure(pt);
pb.arc({ 50.0f, 50.0f }, three_pi_over_two<float>, half_pi<float> / 2.0f);
imgSfc.stroke(foreBrush, pb, nullopt, stroke_props{ 10.0f });
// Create the first '+'.
pb.clear();
pb.new_figure({ 130.0f, 105.0f });
pb.rel_line({ 0.0f, -10.0f });
pb.rel_line({ 25.0f, 0.0f });
pb.rel_line({ 0.0f, -25.0f });
pb.rel_line({ 10.0f, 0.0f });
pb.rel_line({ 0.0f, 25.0f });
pb.rel_line({ 25.0f, 0.0f });
pb.rel_line({ 0.0f, 10.0f });
pb.rel_line({ -25.0f, 0.0f });
pb.rel_line({ -10.0f, 0.0f });
pb.rel_line({ 0.0f, -25.0f });
```
pb.close_figure();
imgSfc.fill(foreBrush, pb);
// Create the second '+'.
pb.insert(pb.begin(), figure_items::change_matrix(
    matrix_2d::init_translate({ 80.0f, 0.0f }));)
imgSfc.fill(foreBrush, pb);

Figure 4 — Path example 4

12.3 Class template basic_figure_items

12.3.1 Introduction

1 The nested classes within the class template basic_figure_items describe figure items.
2 A figure begins with an abs_new_figure or rel_new_figure object. A figure ends when:
   (2.1) — a close_figure object is encountered;
   (2.2) — a abs_new_figure or rel_new_figure object is encountered; or
   (2.3) — there are no more figure items in the path.
3 The basic_path_builder class is a sequential container that contains a path. It provides a simple interface for building a path. A path can also be created using any container that stores basic_figure_items<GraphicsSurfaces>::figure_item objects.

12.3.2 Synopsis

namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    class basic_figure_items {
public:
    class abs_new_figure;
    class rel_new_figure;
    class close_figure;
    class abs_matrix;
    class rel_matrix;
    class revert_matrix;
    class abs_cubic_curve;
    class rel_cubic_curve;
    class abs_line;
    class rel_line;
    class abs_quadratic_curve;
    class rel_quadratic_curve;
    class arc;
}
using figure_item = variant<abs_new_figure, rel_new_figure, 
close_figure, abs_matrix, rel_matrix, revert_matrix, abs_cubic_curve, 
rel_cubic_curve, abs_line, rel_line, abs_quadratic_curve, 
rel_quadratic_curve, arc>;
};

12.3.3 Class template basic_figure_items<GraphicsSurfaces>::abs_new_figure [io2d.absnewfigure]

12.3.3.1 Overview [io2d.absnewfigure.intro]
1 The class template basic_figure_items<GraphicsSurfaces>::abs_new_figure describes a figure item that is a new figure command.
2 It has an at point of type basic_point_2d<GraphicsSurfaces::graphics_math_type>.
3 The data are stored in an object of type typename GraphicsSurfaces::paths::abs_new_figure_data_type. It is accessible using the data member functions.

12.3.3.2 Synopsis [io2d.absnewfigure.synopsis]

namespace std::experimental::io2d::v1 {

template <class GraphicsSurfaces>
class basic_figure_items<GraphicsSurfaces>::abs_new_figure {

public:

using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
using data_type = typename GraphicsSurfaces::paths::abs_new_figure_data_type;

// 12.3.3.3, construct:
abs_new_figure();
explicit abs_new_figure(const basic_point_2d<graphics_math_type>& pt);
abs_new_figure(const abs_new_figure& other) = default;
abs_new_figure(abs_new_figure&& other) noexcept = default;

// assign:
abs_new_figure& operator=(const abs_new_figure& other) = default;
abs_new_figure& operator=(abs_new_figure&& other) noexcept = default;

// 12.3.3.4, accessors:
const data_type& data() const noexcept;
data_type& data() noexcept;

// 12.3.3.5, modifiers:
void at(const basic_point_2d<graphics_math_type>& pt) noexcept;

// 12.3.3.6, observers:
basic_point_2d<graphics_math_type> at() const noexcept;
};

// 12.3.3.7, equality operators:
template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::abs_new_figure& lhs, 
const typename basic_figure_items<GraphicsSurfaces>::abs_new_figure& rhs) noexcept;
template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::abs_new_figure& lhs, 
const typename basic_figure_items<GraphicsSurfaces>::abs_new_figure& rhs) noexcept;
};

12.3.3.3 Constructors [io2d.absnewfigure.ctor]

abs_new_figure();

Effects: Constructs an object of type abs_new_figure.
explicit abs_new_figure(const basic_point_2d<graphics_math_type>& pt);

Effects: Constructs an object of type abs_new_figure.

Postconditions: data() == GraphicsSurfaces::paths::create_abs_new_figure(pt).

Remarks: The at point is pt.
// 12.3.4.3, construct:
rel_new_figure();
explicit rel_new_figure(const basic_point_2d<graphics_math_type>& pt);
rel_new_figure(const rel_new_figure& other) = default;
rel_new_figure(rel_new_figure&& other) noexcept = default;

// assign:
rel_new_figure& operator=(const rel_new_figure& other) = default;
rel_new_figure& operator=(rel_new_figure&& other) noexcept = default;

// 12.3.4.4, accessors:
const data_type& data() const noexcept;
data_type& data() noexcept;

// 12.3.4.5, modifiers:
void at(const basic_point_2d<graphics_math_type>& pt) noexcept;

// 12.3.4.6, observers:
basic_point_2d<graphics_math_type> at() const noexcept;

// 12.3.4.7, equality operators:
template <class GraphicsSurfaces>
bool operator==(const basic_figure_items<GraphicsSurfaces>::rel_new_figure& lhs,
const basic_figure_items<GraphicsSurfaces>::rel_new_figure& rhs) noexcept;

template <class GraphicsSurfaces>
bool operator!=(const basic_figure_items<GraphicsSurfaces>::rel_new_figure& lhs,
const basic_figure_items<GraphicsSurfaces>::rel_new_figure& rhs) noexcept;

12.3.4.3 Constructors

rel_new_figure() noexcept;

**Effects:** Constructs an object of type rel_new_figure.

**Postconditions:** data() == GraphicsSurfaces::paths::create_rel_new_figure().

**Remarks:** The at point is basic_point_2d<graphics_math_type>().

explicit rel_new_figure(const basic_point_2d<graphics_math_type>& pt) noexcept;

**Effects:** Constructs an object of type rel_new_figure.

**Postconditions:** data() == GraphicsSurfaces::paths::create_rel_new_figure(pt).

**Remarks:** The at point is pt.

12.3.4.4 Accessors

const data_type& data() const noexcept;
data_type& data() noexcept;

**Returns:** A reference to the rel_new_figure object’s data object (See: 12.3.4.1).

12.3.4.5 Modifiers

void at(const basic_point_2d<graphics_math_type>& pt) noexcept;

**Effects:** Calls GraphicsSurfaces::paths::at(data(), pt).

**Remarks:** The at point is pt.
12.3.4.6 Observers

basic_point_2d<graphics_math_type> at() const noexcept;

1 Returns: GraphicsSurfaces::paths::at(data()).

2 Remarks: The returned value is the at point.

12.3.4.7 Equality operators

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::rel_new_figure& lhs, const typename basic_figure_items<GraphicsSurfaces>::rel_new_figure& rhs) noexcept;

1 Returns: GraphicsSurfaces::paths::equal(lhs, rhs).

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::rel_new_figure& lhs, const typename basic_figure_items<GraphicsSurfaces>::rel_new_figure& rhs) noexcept;

2 Returns: GraphicsSurfaces::paths::not_equal(lhs, rhs).

12.3.5 Class template basic_figure_items<GraphicsSurfaces>::close_figure

12.3.5.1 Overview

The class template basic_figure_items<GraphicsSurfaces>::close_figure describes a figure item that is a close figure command.

12.3.5.2 Synopsis

namespace std::experimental::io2d::v1 {
  template <class GraphicsSurfaces>
  class basic_figure_items<GraphicsSurfaces>::close_figure {
    public:
      // construct:
      close_figure() = default;
      close_figure(const close_figure& other) = default;
      close_figure(close_figure&& other) noexcept = default;

      // assign:
      close_figure& operator=(const close_figure& other) = default;
      close_figure& operator=(close_figure&& other) noexcept = default;
    }

    // 12.3.5.3, equality operators:
    template <class GraphicsSurfaces>
    bool operator==(const typename basic_figure_items<GraphicsSurfaces>::close_figure& lhs, const typename basic_figure_items<GraphicsSurfaces>::close_figure& rhs) noexcept;

    template <class GraphicsSurfaces>
    bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::close_figure& lhs, const typename basic_figure_items<GraphicsSurfaces>::close_figure& rhs) noexcept;
  }

12.3.5.3 Equality operators

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::close_figure& lhs, const typename basic_figure_items<GraphicsSurfaces>::close_figure& rhs)
12.3.6  Class template basic_figure_items<GraphicsSurfaces>::abs_matrix
[io2d.absmatrix]

12.3.6.1  Overview
[io2d.absmatrix.intro]
The class template basic_figure_items<GraphicsSurfaces>::abs_matrix describes a figure item that is a path command. It has a transform matrix of type basic_matrix_2d<GraphicsSurfaces::graphics_math_type>. The data are stored in an object of type typename GraphicsSurfaces::paths::abs_matrix_data_type. It is accessible using the data member functions.

12.3.6.2  Synopsis
[io2d.absmatrix.synopsis]
12.3.6.3  Constructors

abs_matrix() noexcept;

Effects: Constructs an abs_matrix object.

Postconditions: data() == GraphicsSurfaces::paths::create_abs_matrix().

Remarks: The transform matrix is basic_matrix_2d<graphics_math_type>.

explicit abs_matrix(const basic_matrix_2d<graphics_math_type>& m) noexcept;

Requires: m.is_invertible() is true.

Effects: Constructs an abs_matrix object.

Postconditions: data() == GraphicsSurfaces::paths::create_abs_matrix(m).

Remarks: The transform matrix is m.

12.3.6.4  Accessors

const data_type& data() const noexcept;

data_type& data() noexcept;

Returns: A reference to the abs_matrix object’s data object (See: 12.3.6.1).

12.3.6.5  Modifiers

void matrix(const basic_matrix_2d<graphics_math_type>& m) noexcept;

Requires: m.is_invertible() is true.

Effects: The transform matrix is m.

12.3.6.6  Observers

basic_matrix_2d<typename GraphicsSurfaces::graphics_math_type> matrix() const noexcept;

Returns: The transform matrix.

12.3.6.7  Equality operators

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::abs_matrix& lhs,
const typename basic_figure_items<GraphicsSurfaces>::abs_matrix& rhs) noexcept;

Returns: lhs.matrix() == rhs.matrix().

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::abs_matrix& lhs,
const typename basic_figure_items<GraphicsSurfaces>::abs_matrix& rhs) noexcept;

Returns: lhs.matrix() != rhs.matrix().

12.3.7  Class template basic_figure_items<GraphicsSurfaces>::rel_matrix

12.3.7.1  Overview

The class template basic_figure_items<GraphicsSurfaces>::rel_matrix describes a figure item that is a path command.

It has a transform matrix of type basic_matrix_2d<GraphicsSurfaces::graphics_math_type>.

The data are stored in an object of type typename GraphicsSurfaces::paths::rel_matrix_data_type. It is accessible using the data member functions.
### 12.3.7.2 Synopsis

```cpp
namespace std::experimental::io2d::v1 {
  template <class GraphicsSurfaces>
  class basic_figure_items<GraphicsSurfaces>::rel_matrix {
    public:
      using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
      using data_type =
        typename GraphicsSurfaces::paths::rel_matrix_data_type;

      // 12.3.7.3, construct:
      rel_matrix();
      explicit rel_matrix(const basic_matrix_2d<graphics_math_type>& m) noexcept;
      rel_matrix(const rel_matrix& other) = default;
      rel_matrix(rel_matrix&& other) noexcept = default;

      // assign:
      rel_matrix& operator=(const rel_matrix& other) = default;
      rel_matrix& operator=(rel_matrix&& other) noexcept = default;

      // 12.3.7.4, accessors:
      const data_type& data() const noexcept;
      data_type& data() noexcept;

      // 12.3.7.5, modifiers:
      void matrix(const basic_matrix_2d<graphics_math_type>& m) noexcept;

      // 12.3.7.6, observers:
      basic_matrix_2d<graphics_math_type> matrix() const noexcept;
    }

    // 12.3.7.7, equality operators:
    template <class GraphicsSurfaces>
    bool operator==(const typename basic_figure_items<GraphicsSurfaces>::rel_matrix& lhs,
                  const typename basic_figure_items<GraphicsSurfaces>::rel_matrix& rhs)
      noexcept;
    template <class GraphicsSurfaces>
    bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::rel_matrix& lhs,
                  const typename basic_figure_items<GraphicsSurfaces>::rel_matrix& rhs)
      noexcept;
  }
}

### 12.3.7.3 Constructors

- **rel_matrix()**

  ```cpp
  rel_matrix() noexcept;
  ```

  **Effects:** Equivalent to: `rel_matrix{ basic_matrix_2d() };`

  **Postconditions:** `data() == GraphicsSurfaces::paths::create_rel_matrix()`.

- **explicit rel_matrix(const basic_matrix_2d<graphics_math_type>& m) noexcept**

  **Requires:** `m.is_invertible()` is true.

  **Effects:** Constructs an object of type `rel_matrix`.

  **Remarks:** The transform matrix is `m`.

### 12.3.7.4 Accessors

- **const data_type& data() const noexcept**

  ```cpp
  const data_type& data() const noexcept;
  ```

  **Returns:** A reference to the `rel_matrix` object’s data object (See: 12.3.7.1).
12.3.7.5 Modifiers

void matrix(const basic_matrix_2d&lt;tgraphics_math_type&gt;&amp; m) noexcept;

Requires: m.is_invertible() is true.

Effects: The transform matrix is m.

12.3.7.6 Observers

basic_matrix_2d&lt;typename GraphicsSurfaces::graphics_math_type&gt; matrix() const noexcept;

Returns: The transform matrix.

12.3.7.7 Equality operators

template &lt;class GraphicsSurfaces&gt;
bool operator==(const typename basic_figure_items&lt;GraphicsSurfaces&gt;::rel_matrix&amp; lhs,
const typename basic_figure_items&lt;GraphicsSurfaces&gt;::rel_matrix&amp; rhs)
noexcept;

Returns: lhs.matrix() == rhs.matrix().

template &lt;class GraphicsSurfaces&gt;
bool operator!=(const typename basic_figure_items&lt;GraphicsSurfaces&gt;::rel_matrix&amp; lhs,
const typename basic_figure_items&lt;GraphicsSurfaces&gt;::rel_matrix&amp; rhs)
noexcept;

Returns: lhs.matrix() != rhs.matrix().

12.3.8 Class template basic_figure_items&lt;GraphicsSurfaces&gt;::revert_matrix

12.3.8.1 Overview

The class template basic_figure_items&lt;GraphicsSurfaces&gt;::revert_matrix describes a figure item that is a path command.

12.3.8.2 Synopsis

namespace std::experimental::io2d::v1 {
class basic_figure_items&lt;GraphicsSurfaces&gt;::revert_matrix {
    public:
    // construct:
    revert_matrix() = default;
    revert_matrix(const revert_matrix&amp; other) = default;
    revert_matrix(revert_matrix&amp;&amp; other) noexcept = default;
    
    // assign:
    revert_matrix&amp; operator=(const revert_matrix&amp; other) = default;
    revert_matrix&amp; operator=(revert_matrix&amp;&amp; other) noexcept = default;
};

// 12.3.8.3, equality operators:
template &lt;class GraphicsSurfaces&gt;
bool operator==(const typename basic_figure_items&lt;GraphicsSurfaces&gt;::revert_matrix&amp; lhs,
const typename basic_figure_items&lt;GraphicsSurfaces&gt;::revert_matrix&amp; rhs)
noexcept;
template &lt;class GraphicsSurfaces&gt;
bool operator!=(const typename basic_figure_items&lt;GraphicsSurfaces&gt;::revert_matrix&amp; lhs,
const typename basic_figure_items&lt;GraphicsSurfaces&gt;::revert_matrix&amp; rhs)
noexcept;
}
12.3.8.3 Equality operators

template <class GraphicsSurfaces>
bool operator==(  
    const typename basic_figure_items<GraphicsSurfaces>::revert_matrix& lhs,  
    const typename basic_figure_items<GraphicsSurfaces>::revert_matrix& rhs)  
noexcept;

Returns: true.

template <class GraphicsSurfaces>
bool operator!=(  
    const typename basic_figure_items<GraphicsSurfaces>::revert_matrix& lhs,  
    const typename basic_figure_items<GraphicsSurfaces>::revert_matrix& rhs)  
noexcept;

Returns: false.

12.3.9 Class template basic_figure_items<GraphicsSurfaces>::abs_cubic_curve

12.3.9.1 Overview

The class basic_figure_items<GraphicsSurfaces>::abs_cubic_curve describes a figure item that is a segment.

It has a first control point of type basic_point_2d<GraphicsSurfaces::graphics_math_type>, a second control point of type basic_point_2d<GraphicsSurfaces::graphics_math_type>, and an end point of type basic_point_2d<GraphicsSurfaces::graphics_math_type>.

The data are stored in an object of type typename GraphicsSurfaces::paths::abs_cubic_curve_data_type. It is accessible using the data member functions.

12.3.9.2 Synopsis

namespace std::experimental::io2d:v1 {
    template <class GraphicsSurfaces>
    class basic_figure_items<GraphicsSurfaces>::abs_cubic_curve {
        public:
            using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
            using data_type = typename GraphicsSurfaces::paths::abs_cubic_curve_data_type;

            // 12.3.9.3, construct:
            abs_cubic_curve();
            abs_cubic_curve(const basic_point_2d<graphics_math_type>& cpt1,  
                const basic_point_2d<graphics_math_type>& cpt2,  
                const basic_point_2d<graphics_math_type>& ept) noexcept;
            abs_cubic_curve(const abs_cubic_curve& other) = default;
            abs_cubic_curve(abs_cubic_curve&& other) noexcept = default;

            // assign:
            abs_cubic_curve& operator=(const abs_cubic_curve& other) = default;
            abs_cubic_curve& operator=(abs_cubic_curve&& other) noexcept = default;

            // 12.3.9.4, accessors:
            const data_type& data() const noexcept;
            data_type& data() noexcept;

            // 12.3.9.5, modifiers:
            void control_pt1(const basic_point_2d<graphics_math_type>& cpt) noexcept;
            void control_pt2(const basic_point_2d<graphics_math_type>& cpt) noexcept;
            void end_pt(const basic_point_2d<graphics_math_type>& ept) noexcept;

            // 12.3.9.6, observers:
            basic_point_2d<graphics_math_type> control_pt1() const noexcept;
            basic_point_2d<graphics_math_type> control_pt2() const noexcept;
            basic_point_2d<graphics_math_type> end_pt() const noexcept;
    }
}
12.3.9.7, equality operators:

```cpp
template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::abs_cubic_curve& lhs, const typename basic_figure_items<GraphicsSurfaces>::abs_cubic_curve& rhs) noexcept;
```

```cpp
template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::abs_cubic_curve& lhs, const typename basic_figure_items<GraphicsSurfaces>::abs_cubic_curve& rhs) noexcept;
```

12.3.9.3 Constructors

```cpp
abs_cubic_curve() noexcept;
```

**Effects:** Equivalent to `abs_cubic_curve{ basic_point_2d(), basic_point_2d(), basic_point_2d() }`.

```cpp
abs_cubic_curve(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& cpt1, const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& cpt2, const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& ept) noexcept;
```

**Effects:** Constructs an object of type `abs_cubic_curve`.

**Remarks:** The first control point is `cpt1`.

**Remarks:** The second control point is `cpt2`.

**Remarks:** The end point is `ept`.

12.3.9.4 Accessors

```cpp
const data_type& data() const noexcept;
```

**Returns:** A reference to the `rel_matrix` object's data object (See: 12.3.9.1).

```cpp
data_type& data() noexcept;
```

12.3.9.5 Modifiers

```cpp
void control_pt1(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& cpt) noexcept;
```

**Effects:** The first control point is `cpt`.

```cpp
void control_pt2(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& cpt) noexcept;
```

**Effects:** The second control point is `cpt`.

```cpp
void end_pt(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& ept) noexcept;
```

**Effects:** The end point is `ept`.

12.3.9.6 Observers

```cpp
basic_point_2d<typename GraphicsSurfaces::graphics_math_type> control_pt1() const noexcept;
```

**Returns:** The first control point.

```cpp
basic_point_2d<typename GraphicsSurfaces::graphics_math_type> control_pt2() const noexcept;
```

**Returns:** The second control point.

```cpp
basic_point_2d<typename GraphicsSurfaces::graphics_math_type> end_pt() const noexcept;
```

**Returns:** The end point.
12.3.9.7 Equality operators

```cpp
template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::abs_cubic_curve& lhs, const typename basic_figure_items<GraphicsSurfaces>::abs_cubic_curve& rhs) noexcept;
```

Returns:

```cpp
lhs.control_pt1() == rhs.control_pt1() &&
lhs.control_pt2() == rhs.control_pt2() &&
lhs.end_pt() == rhs.end_pt();
```

```cpp
template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::abs_cubic_curve& lhs, const typename basic_figure_items<GraphicsSurfaces>::abs_cubic_curve& rhs) noexcept;
```

Returns:

```cpp
lhs.control_pt1() != rhs.control_pt1() ||
lhs.control_pt2() != rhs.control_pt2() ||
lhs.end_pt() != rhs.end_pt();
```

12.3.10 Class template basic_figure_items<GraphicsSurfaces>::rel_cubic_curve

12.3.10.1 Overview

The class `basic_figure_items<GraphicsSurfaces>::rel_cubic_curve` describes a figure item that is a segment. It has a first control point of type `basic_point_2d<GraphicsSurfaces::graphics_math_type>`, a second control point of type `basic_point_2d<GraphicsSurfaces::graphics_math_type>`, and an end point of type `basic_point_2d<GraphicsSurfaces::graphics_math_type>`. The data are stored in an object of type `typename GraphicsSurfaces::paths::rel_cubic_curve_data_type`. It is accessible using the data member functions.

12.3.10.2 Synopsis

```cpp
namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    class basic_figure_items<GraphicsSurfaces>::rel_cubic_curve {
        public:
            using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
            using data_type = typename GraphicsSurfaces::paths::rel_cubic_curve_data_type;

            // 12.3.10.3, construct:
            rel_cubic_curve();
            rel_cubic_curve(const basic_point_2d<graphics_math_type>& cpt1,
                            const basic_point_2d<graphics_math_type>& cpt2,
                            const basic_point_2d<graphics_math_type>& ept) noexcept;
            rel_cubic_curve(const rel_cubic_curve& other) = default;
            rel_cubic_curve(rel_cubic_curve&& other) noexcept = default;

            // assign:
            rel_cubic_curve& operator=(const rel_cubic_curve& other) = default;
            rel_cubic_curve& operator=(rel_cubic_curve&& other) noexcept = default;

            // 12.3.10.4, accessors:
            const data_type& data() const noexcept;
            data_type& data() noexcept;

            // 12.3.10.5, modifiers:
            void control_pt1(const basic_point_2d<graphics_math_type>& cpt) noexcept;
            void control_pt2(const basic_point_2d<graphics_math_type>& cpt) noexcept;
            void end_pt(const basic_point_2d<graphics_math_type>& ept) noexcept;
    }
}
```
12.3.10.6  Observers:

```cpp
basic_point_2d<graphics_math_type> control_pt1() const noexcept;
basic_point_2d<graphics_math_type> control_pt2() const noexcept;
basic_point_2d<graphics_math_type> end_pt() const noexcept;
};
```

12.3.10.7  Equality Operators:

```cpp
template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::rel_cubic_curve& lhs,
               const typename basic_figure_items<GraphicsSurfaces>::rel_cubic_curve& rhs)
noexcept;
```

```cpp
template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::rel_cubic_curve& lhs,
               const typename basic_figure_items<GraphicsSurfaces>::rel_cubic_curve& rhs)
noexcept;
```

12.3.10.3  Constructors

```cpp
rel_cubic_curve() noexcept;
```

**Effects:** Equivalent to `rel_cubic_curve{ basic_point_2d(), basic_point_2d(), basic_point_2d() }`.

```cpp
rel_cubic_curve(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& cpt1,
                const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& cpt2,
                const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& ept) noexcept;
```

**Effects:** Constructs an object of type `rel_cubic_curve`.

**Remarks:** The first control point is `cpt1`.

**Remarks:** The second control point is `cpt2`.

**Remarks:** The end point is `ept`.

12.3.10.4  Accessors

```cpp
const data_type& data() const noexcept;
data_type& data() noexcept;
```

**Returns:** A reference to the `rel_matrix` object’s data object (See: 12.3.10.1).

12.3.10.5  Modifiers

```cpp
void control_pt1(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& cpt) noexcept;
```

**Effects:** The first control point is `cpt`.

```cpp
void control_pt2(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& cpt) noexcept;
```

**Effects:** The second control point is `cpt`.

```cpp
void end_pt(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& ept) noexcept;
```

**Effects:** The end point is `ept`.

12.3.10.6  Observers

```cpp
basic_point_2d<graphics_math_type> control_pt1() const noexcept;
```

**Returns:** The first control point.

```cpp
basic_point_2d<graphics_math_type> control_pt2() const noexcept;
```

**Returns:** The second control point.
basic_point_2d<graphics_math_type> end_pt() const noexcept;

Returns: The end point.

12.3.10.7 Equality operators

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::rel_cubic_curve& lhs,
const typename basic_figure_items<GraphicsSurfaces>::rel_cubic_curve& rhs)
noexcept;

Returns: lhs.control_pt1() == rhs.control_pt1() && lhs.control_pt2() == rhs.control_pt2() &&
lhs.end_pt() == rhs.end_pt().

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::rel_cubic_curve& lhs,
const typename basic_figure_items<GraphicsSurfaces>::rel_cubic_curve& rhs)
noexcept;

Returns: lhs.control_pt1() != rhs.control pt1() || lhs.control pt2() != rhs.control pt2() ||
lhs.end_pt() != rhs.end pt().

12.3.11 Class template basic_figure_items<GraphicsSurfaces>::abs line

12.3.11.1 Overview

The class basic_figure_items<GraphicsSurfaces>::abs line describes a figure item that is a segment.

It has an end point of type basic_point_2d<GraphicsSurfaces::graphics_math_type>.

The data are stored in an object of type typename GraphicsSurfaces::paths::abs line_data_type. It
is accessible using the data member functions.

12.3.11.2 Synopsis

namespace std::experimemtal::io2d::v1 {

template <class GraphicsSurfaces>
class basic_figure_items<GraphicsSurfaces>::abs line {

public:

    using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
    using data_type = typename GraphicsSurfaces::paths::abs line_data_type;

    // 12.3.11.3, construct:
    abs line();
    explicit abs line(const basic_point_2d<graphics_math_type>& pt);
    abs line(const abs line& other) = default;
    abs line(abs line&& other) noexcept = default;

    // assign:
    abs line& operator=(const abs line& other) = default;
    abs line& operator=(abs line&& other) noexcept = default;

    // 12.3.11.4, accessors:
    const data_type& data() const noexcept;
    data_type& data() noexcept;

    // 12.3.11.5, modifiers:
    void at(const basic_point_2d<graphics_math_type>& pt) noexcept;

    // 12.3.11.6, observers:
    basic_point_2d<graphics_math_type> at() const noexcept;
};
12.3.11.7 Equality operators

```cpp
template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::abs_line& lhs,
               const typename basic_figure_items<GraphicsSurfaces>::abs_line& rhs)
    noexcept;
```

```cpp
template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::abs_line& lhs,
               const typename basic_figure_items<GraphicsSurfaces>::abs_line& rhs)
    noexcept;
```

12.3.11.3 Constructors

```cpp
abs_line() noexcept;
```

**Effects:** Equivalent to: `abs_line{ basic_point_2d() }`;

```cpp
explicit abs_line(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& pt) noexcept;
```

**Effects:** Constructs an object of type `abs_line`.

**Remarks:** The end point is `pt`.

12.3.11.4 Accessors

```cpp
const data_type& data() const noexcept;
```

**Returns:** A reference to the `abs_line` object's data object (See: 12.3.11.1).

```cpp
data_type& data() noexcept;
```

12.3.11.5 Modifiers

```cpp
void to(const basic_point_2d<graphics_math_type>& pt) noexcept;
```

**Effects:** The end point is `pt`.

12.3.11.6 Observers

```cpp
basic_point_2d<graphics_math_type> to() const noexcept;
```

**Returns:** The end point.

12.3.11.7 Equality operators

```cpp
template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::abs_line& lhs,
               const typename basic_figure_items<GraphicsSurfaces>::abs_line& rhs)
    noexcept;
```

**Returns:** `lhs.to() == rhs.to()`.

```cpp
template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::abs_line& lhs,
               const typename basic_figure_items<GraphicsSurfaces>::abs_line& rhs)
    noexcept;
```

**Returns:** `lhs.to() != rhs.to()`.

12.3.12 Class rel_line

12.3.12.1 Overview

The class `basic_figure_items<GraphicsSurfaces>::rel_line` describes a figure item that is a segment. It has an `end point` of type `basic_point_2d<GraphicsSurfaces::graphics_math_type>`. The data are stored in an object of type `typename GraphicsSurfaces::paths::rel_line_data_type`. It is accessible using the `data` member functions.
12.3.12.2 Synopsis

```cpp
namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    class basic_figure_items<GraphicsSurfaces>::rel_line {
        public:
            using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
            using data_type =
                typename GraphicsSurfaces::paths::rel_line_data_type;

            // 12.3.12.3, construct:
            rel_line();
            explicit rel_line(const basic_point_2d<graphics_math_type>& pt);
            rel_line(const rel_line& other) = default;
            rel_line(rel_line&& other) noexcept = default;

            // assign:
            rel_line& operator=(const rel_line& other) = default;
            rel_line& operator=(rel_line&& other) noexcept = default;

            // 12.3.12.4, accessors:
            const data_type& data() const noexcept;
            data_type& data() noexcept;

            // 12.3.12.5, modifiers:
            void at(const basic_point_2d<graphics_math_type>& pt) noexcept;

            // 12.3.12.6, observers:
            basic_point_2d<graphics_math_type> at() const noexcept;
        }
    }
}
```

12.3.12.3 Constructors

```cpp
rel_line() noexcept;
```

1 Effects: Equivalent to: `rel_line{ basic_point_2d() }`;

```cpp
explicit rel_line(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& pt) noexcept;
```

2 Effects: Constructs an object of type `rel_line`.

3 Remarks: The end point is pt.

12.3.12.4 Accessors

```cpp
const data_type& data() const noexcept;
```

1 Returns: A reference to the `rel_line` object’s data object (See: 12.3.12.1).

12.3.12.5 Modifiers

```cpp
void at(const basic_point_2d<graphics_math_type>& pt) noexcept;
```

1 Effects: The end point is pt.
12.3.12.6 Observers

basic_point_2d<graphics_math_type> to() const noexcept;

Returns: The end point.

12.3.12.7 Equality operators

template <class GraphicsSurfaces>
bool operator==(
    const typename basic_figure_items<GraphicsSurfaces>::rel_line& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::rel_line& rhs)
noexcept;

Returns: lhs.to() == rhs.to().

12.3.13 Class template basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve

12.3.13.1 Overview

The class basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve describes a figure item that is a segment. It has a control point of type basic_point_2d<GraphicsSurfaces::graphics_math_type> and an end point of type basic_point_2d<GraphicsSurfaces::graphics_math_type>.

The data are stored in an object of type typename GraphicsSurfaces::paths::abs_quadratic_curve_data_type. It is accessible using the data member functions.

12.3.13.2 Synopsis

namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    class basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve {
        public:
            using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
            using data_type = typename GraphicsSurfaces::paths::abs_quadratic_curve_data_type;

            // 12.3.13.3, construct:
            abs_quadratic_curve();
            abs_quadratic_curve(const basic_point_2d<graphics_math_type>& cpt,
                const basic_point_2d<graphics_math_type>& ept);
            abs_quadratic_curve(const abs_quadratic_curve& other) = default;
            abs_quadratic_curve(abs_quadratic_curve&& other) noexcept = default;

            // assign:
            abs_quadratic_curve& operator=(const abs_quadratic_curve& other) = default;
            abs_quadratic_curve& operator=(abs_quadratic_curve&& other) noexcept = default;

            // 12.3.13.4, accessors:
            const data_type& data() const noexcept;
            data_type& data() noexcept;

            // 12.3.13.5, modifiers:
            void control_pt(const basic_point_2d<graphics_math_type>& cpt) noexcept;
            void end_pt(const basic_point_2d<graphics_math_type>& ept) noexcept;
        };
    };
};
// 12.3.13.6, observers:
  basic_point_2d<graphics_math_type> control_pt() const noexcept;
  basic_point_2d<graphics_math_type> end_pt() const noexcept;
};

// 12.3.13.7, equality operators:
template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve& lhs,
                const typename basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve& rhs)
  noexcept;
template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve& lhs,
                const typename basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve& rhs)
  noexcept;

12.3.13.3 Constructors

abs_quadratic_curve() noexcept;
  Effects: Equivalent to: abs_quadratic_curve{ basic_point_2d(), basic_point_2d() };

abs_quadratic_curve(const basic_point_2d<graphics_math_type>& cpt,
                     const basic_point_2d<graphics_math_type>& ept) noexcept;
  Effects: Constructs an object of type abs_quadratic_curve.
  Remarks: The control point is cpt.
  Remarks: The end point is ept.

12.3.13.4 Accessors

const data_type& data() const noexcept;
  Returns: A reference to the abs_quadratic_curve object’s data object (See: 12.3.13.1).

data_type& data() noexcept;

12.3.13.5 Modifiers

void control_pt(const basic_point_2d<graphics_math_type>& cpt) noexcept;
  Effects: The control point is cpt.

void end_pt(const basic_point_2d<graphics_math_type>& ept) noexcept;
  Effects: The end point is ept.

12.3.13.6 Observers

basic_point_2d<graphics_math_type> control_pt() const noexcept;
  Returns: The control point.

basic_point_2d<graphics_math_type> end_pt() const noexcept;
  Returns: The end point.

12.3.13.7 Equality operators

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve& lhs,
                const typename basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve& rhs)
  noexcept;
  Returns: lhs.control_pt() == rhs.control_pt() && lhs.end_pt() == rhs.end_pt().
template <class GraphicsSurfaces>
bool operator!=(
    const typename basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve& rhs)
noexcept;

Returns: lhs.control_pt() != rhs.control_pt() || lhs.end_pt() != rhs.end_pt().

12.3.14 Class template basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve
[io2d.relquadraticcurve]

12.3.14.1 Overview
[io2d.relquadraticcurve.intro]
The class basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve describes a figure item that is a segment.

It has a control point of type basic_point_2d<GraphicsSurfaces::graphics_math_type> and an end point of type basic_point_2d<GraphicsSurfaces::graphics_math_type>.

The data are stored in an object of type typename GraphicsSurfaces::paths::rel_quadratic_curve_data_type. It is accessible using the data member functions.

12.3.14.2 Synopsis
[io2d.relquadraticcurve.synopsis]
```cpp
namespace std::experimental::io2d::v1 {
  template <class GraphicsSurfaces>
  class basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve {
    public:
      using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
      using data_type = typename GraphicsSurfaces::paths::rel_quadratic_curve_data_type;

      // 12.3.14.3, construct:
      rel_quadratic_curve();
      rel_quadratic_curve(const basic_point_2d<graphics_math_type>& cpt,
                          const basic_point_2d<graphics_math_type>& ept);
      rel_quadratic_curve(const rel_quadratic_curve& other) = default;
      rel_quadratic_curve(rel_quadratic_curve&& other) noexcept = default;

      // assign:
      rel_quadratic_curve& operator=(const rel_quadratic_curve& other) = default;
      rel_quadratic_curve& operator=(rel_quadratic_curve&& other) noexcept = default;

      // 12.3.14.4, accessors:
      const data_type& data() const noexcept;
      data_type& data() noexcept;

      // 12.3.14.5, modifiers:
      void control_pt(const basic_point_2d<graphics_math_type>& cpt) noexcept;
      void end_pt(const basic_point_2d<graphics_math_type>& ept) noexcept;

      // 12.3.14.6, observers:
      basic_point_2d<graphics_math_type> control_pt() const noexcept;
      basic_point_2d<graphics_math_type> end_pt() const noexcept;
    }
  };
```

// 12.3.14.7, equality operators:
```cpp
template <class GraphicsSurfaces>
bool operator==(  
    const typename basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve& rhs)
noexcept;

template <class GraphicsSurfaces>
bool operator!=(  
    const typename basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve& rhs)
noexcept;
```

§ 12.3.14.2
12.3.14.3 Constructors

rel_quadratic_curve() noexcept;

Effects: Equivalent to: rel_quadratic_curve( basic_point_2d(), basic_point_2d() );

rel_quadratic_curve(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& cpt,
                    const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& ept) noexcept;

Effects: Constructs an object of type rel_quadratic_curve.
Remarks: The control point is cpt.
Remarks: The end point is ept.

12.3.14.4 Accessors

const data_type& data() const noexcept;
data_type& data() noexcept;

Returns: A reference to the rel_quadratic_curve object’s data object (See: 12.3.14.1).

12.3.14.5 Modifiers

void control_pt(const basic_point_2d<graphics_math_type>& cpt) noexcept;

Effects: The control point is cpt.

void end_pt(const basic_point_2d<graphics_math_type>& ept) noexcept;

Effects: The end point is ept.

12.3.14.6 Observers

basic_point_2d<graphics_math_type> control_pt() const noexcept;

Returns: The control point.

basic_point_2d<graphics_math_type> end_pt() const noexcept;

Returns: The end point.

12.3.14.7 Equality operators

template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve& lhs,
                      const typename basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve& rhs)
                      noexcept;

Returns: lhs.control_pt() == rhs.control_pt() && lhs.end_pt() == rhs.end_pt().

template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve& lhs,
                      const typename basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve& rhs)
                      noexcept;

Returns: lhs.control_pt() != rhs.control_pt() || lhs.end_pt() != rhs.end_pt().

12.3.15 Class template basic_figure_items<GraphicsSurfaces>::arc

12.3.15.1 Overview

The class basic_figure_items<GraphicsSurfaces>::arc describes a figure item that is a segment.

It has a radius of type basic_point_2d<GraphicsSurfaces::graphics_math_type>, a rotation of type float, and a start angle of type float.

It forms a portion of the circumference of a circle. The centre of the circle is implied by the start point, the radius and the start angle of the arc.
The data are stored in an object of type `typename GraphicsSurfaces::paths::arc_data_type`. It is accessible using the `data` member functions.

### 12.3.15.2 Synopsis

```cpp	namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    class basic_figure_items<GraphicsSurfaces>::arc {
        public:
            using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
            using data_type = typename GraphicsSurfaces::paths::arc_data_type;

            // 12.3.15.3, construct:
            arc();
            arc(const basic_point_2d<graphics_math_type>& rad, float rot, float sang) noexcept;
            arc(const arc& other) = default;
            arc(arc&& other) noexcept = default;

            // assign:
            arc& operator=(const arc& other) = default;
            arc& operator=(arc&& other) noexcept = default;

            // 12.3.15.4, accessor:
            const data_type& data() const noexcept;
            data_type& data() noexcept;

            // 12.3.15.5, modifier:
            void radius(const basic_point_2d<graphics_math_type>& rad) noexcept;
            void rotation(float rot) noexcept;
            void start_angle(float radians) noexcept;

            // 12.3.15.6, observer:
            basic_point_2d<typename GraphicsSurfaces::graphics_math_type> radius() const noexcept;
            float rotation() const noexcept;
            float start_angle() const noexcept;
            basic_point_2d<typename GraphicsSurfaces::graphics_math_type> center(const basic_point_2d<graphics_math_type>& cpt, const basic_matrix_2d<graphics_math_type>& m = basic_matrix_2d<graphics_math_type>{}) const noexcept;
            basic_point_2d<typename GraphicsSurfaces::graphics_math_type> end_pt(const basic_point_2d<graphics_math_type>& cpt, const basic_matrix_2d<graphics_math_type>& m = basic_matrix_2d<graphics_math_type>{}) const noexcept;
    };

    // 12.3.15.7, equality operators:
    template <class GraphicsSurfaces>
    bool operator==(
        const typename basic_figure_items<GraphicsSurfaces>::arc& lhs,
        const typename basic_figure_items<GraphicsSurfaces>::arc& rhs) noexcept;
    template <class GraphicsSurfaces>
    bool operator!=(
        const typename basic_figure_items<GraphicsSurfaces>::arc& lhs,
        const typename basic_figure_items<GraphicsSurfaces>::arc& rhs) noexcept;
};
```

### 12.3.15.3 Constructors

```cpp
class arc():
    Effects: Equivalent to: arc{ basic_point_2d(10.0f, 10.0f), pi<float>, pi<float> };
```
arc(const basic_point_2d<graphics_math_type>& rad,
    float rot, float sang) noexcept;

Effects: Constructs an object of type arc.

The radius is rad.
The rotation is rot.
The start angle is sang.

12.3.15.4 Accessors

const data_type& data() const noexcept;
data_type& data() noexcept;

Returns: A reference to the arc object’s data object (See: 12.3.15.1).

12.3.15.5 Modifiers

void radius(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& rad) noexcept;

Effects: The radius is rad.

constexpr void rotation(float rot) noexcept;
Effects: The rotation is rot.

void start_angle(float sang) noexcept;
Effects: The start angle is sang.

12.3.15.6 Observers

basic_point_2d<typename GraphicsSurfaces::graphics_math_type> radius() const noexcept;

Returns: The radius.

float rotation() const noexcept;
Returns: The rotation.

float start_angle() const noexcept;
Returns: The start angle.

basic_point_2d<graphics_math_type> center(
    const basic_point_2d<graphics_math_type>& cpt,
    const basic_matrix_2d<graphics_math_type>& m =
    basic_matrix_2d<graphics_math_type>{}) const noexcept;

Returns: As-if:
    auto lmtx = m;
lmtx.m20 = 0.0f;
lmtx.m21 = 0.0f;
    auto centerOffset = point_for_angle(two_pi<float> - start_angle(), radius());
    centerOffset.y = -centerOffset.y;
    return cpt - centerOffset * lmtx;

basic_point_2d<graphics_math_type> end_pt(
    const basic_point_2d<graphics_math_type>& cpt,
    const basic_matrix_2d<graphics_math_type>& m =
    basic_matrix_2d<graphics_math_type>{}) const noexcept;

Returns: As-if:
    auto lmtx = m;
    auto tfrm = matrix_2d::init_rotate(start_angle() + rotation());
lmtx.m20 = 0.0f;
lmtx.m21 = 0.0f;
    auto pt = (radius() * tfrm);
    pt.y = -pt.y;
    return cpt + pt * lmtx;
12.3.15.7 Equality operators

```cpp
template <class GraphicsSurfaces>
bool operator==(const typename basic_figure_items<GraphicsSurfaces>::arc& lhs,
               const typename basic_figure_items<GraphicsSurfaces>::arc& rhs);
```

Returns:

\[
lhs.\text{radius}() == rhs.\text{radius}() \land lhs.\text{rotation}() == rhs.\text{rotation}() \land
lhs.\text{start\_angle}() == rhs.\text{start\_angle}()
\]

```cpp
template <class GraphicsSurfaces>
bool operator!=(const typename basic_figure_items<GraphicsSurfaces>::arc& lhs,
                const typename basic_figure_items<GraphicsSurfaces>::arc& rhs);
```

Returns:

\[
lhs.\text{radius}() != rhs.\text{radius}() \lor lhs.\text{rotation}() != rhs.\text{rotation}() \lor
lhs.\text{start\_angle}() != rhs.\text{start\_angle}()
\]

12.3.16 Path interpretation

This subclause describes how to interpret a path for use in a rendering and composing operation.

Interpreting a path consists of sequentially evaluating the figure items contained in the figures in the path and transforming them into zero or more figures as-if in the manner specified in this subclause.

The interpretation of a path requires the state data specified in Table 17.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtx</td>
<td>Path transformation matrix</td>
<td>matrix_2d</td>
<td>matrix_2d{}</td>
</tr>
<tr>
<td>currPt</td>
<td>Current point</td>
<td>point_2d</td>
<td>unspecified</td>
</tr>
<tr>
<td>lnfPt</td>
<td>Last new figure point</td>
<td>point_2d</td>
<td>unspecified</td>
</tr>
<tr>
<td>mtxStk</td>
<td>Matrix stack</td>
<td>stack&lt;matrix_2d&gt;</td>
<td>stack&lt;matrix_2d&gt;{}</td>
</tr>
</tbody>
</table>

When interpreting a path, until a `figure_items::abs_new_figure` figure item is reached, a path shall only contain path command figure items; no diagnostic is required. If a figure is a degenerate figure, none of its figure items have any effects, with two exceptions:

1. the path’s `figure_items::abs_new_figure` or `figure_items::rel_new_figure` figure item sets the value of `currPt` as-if the figure item was interpreted; and,
2. any path command figure items are evaluated with full effect.

The effects of a figure item contained in a `figure_items::figure_item` object when that object is being evaluated during path interpretation are described in Table 18.

If evaluation of a figure item contained in a `figure_items::figure_item` during path interpretation results in the figure item becoming a degenerate segment, its effects are ignored and interpretation continues as-if that figure item did not exist.

<table>
<thead>
<tr>
<th>Figure item</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>figure_items::abs_new_figure</code></td>
<td>Creates a new figure. Sets <code>currPt</code> to <code>p.at()</code> * <code>mtx</code>. Sets <code>lnfPt</code> to <code>currPt</code>.</td>
</tr>
<tr>
<td><code>figure_items::rel_new_figure</code></td>
<td>Let <code>mm</code> equal <code>mtx</code>. Let <code>mm.m20</code> equal 0.0f. Let <code>mm.m21</code> equal 0.0f. Creates a new figure. Sets <code>currPt</code> to <code>currPt</code> + <code>p.at()</code> * <code>mm</code>. Sets <code>lnfPt</code> to <code>currPt</code>.</td>
</tr>
<tr>
<td>Figure item</td>
<td>Effects</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td><code>figure_items::close_figure</code> p</td>
<td>Creates a line from <code>currPt</code> to <code>lnfPt</code>. Makes the current figure a closed figure. Creates a new figure. Sets <code>currPt</code> to <code>lnfPt</code>.</td>
</tr>
<tr>
<td><code>figure_items::abs_matrix</code> p</td>
<td>Calls <code>mtxStk.push(mtx)</code>. Sets <code>mtx</code> to <code>p.matrix()</code>.</td>
</tr>
<tr>
<td><code>figure_items::rel_matrix</code> p</td>
<td>Calls <code>mtxStk.push(mtx)</code>. Sets <code>mtx</code> to <code>p.matrix() * mtx</code>.</td>
</tr>
<tr>
<td><code>figure_items::revert_matrix</code> p</td>
<td>If <code>mtxStk.empty()</code> is false, sets <code>mtx</code> to <code>mtxStk.top()</code> then calls <code>mtxStk.pop()</code>. Otherwise sets <code>mtx</code> to its initial value as specified in Table 17.</td>
</tr>
<tr>
<td><code>figure_items::abs_line</code> p</td>
<td>Let <code>pt</code> equal <code>p.to() * mtx</code>. Creates a line from <code>currPt</code> to <code>pt</code>. Sets <code>currPt</code> to <code>pt</code>.</td>
</tr>
<tr>
<td><code>figure_items::rel_line</code> p</td>
<td>Let <code>mm</code> equal <code>mtx</code>. Let <code>mm.m20</code> equal 0.0f. Let <code>mm.m21</code> equal 0.0f. Let <code>pt</code> equal <code>currPt + p.to() * mm</code>. Creates a line from <code>currPt</code> to <code>pt</code>. Sets <code>currPt</code> to <code>pt</code>.</td>
</tr>
<tr>
<td><code>figure_items::abs_quadratic_curve</code> p</td>
<td>Let <code>cpt</code> equal <code>p.control_pt() * mtx</code>. Let <code>ept</code> equal <code>p.end_pt() * mtx</code>. Creates a quadratic Bézier curve from <code>currPt</code> to <code>ept</code> using <code>cpt</code> as the curve's control point. Sets <code>currPt</code> to <code>ept</code>.</td>
</tr>
<tr>
<td><code>figure_items::rel_quadratic_curve</code> p</td>
<td>Let <code>mm</code> equal <code>mtx</code>. Let <code>mm.m20</code> equal 0.0f. Let <code>mm.m21</code> equal 0.0f. Let <code>cpt</code> equal <code>currPt + p.control_pt() * mm</code>. Let <code>ept</code> equal <code>currPt + p.control_pt() * mm + p.end_pt() * mm</code>. Creates a quadratic Bézier curve from <code>currPt</code> to <code>ept</code> using <code>cpt</code> as the curve's control point. Sets <code>currPt</code> to <code>ept</code>.</td>
</tr>
<tr>
<td><code>figure_items::abs_cubic_curve</code> p</td>
<td>Let <code>cpt1</code> equal <code>p.control_pt1() * mtx</code>. Let <code>cpt2</code> equal <code>p.control_pt2() * mtx</code>. Creates a cubic Bézier curve from <code>currPt</code> to <code>ept</code> using <code>cpt1</code> as the curve's first control point and <code>cpt2</code> as the curve's second control point. Sets <code>currPt</code> to <code>ept</code>.</td>
</tr>
<tr>
<td><code>figure_items::rel_quadratic_curve</code> p</td>
<td>Let <code>mm</code> equal <code>mtx</code>. Let <code>mm.m20</code> equal 0.0f. Let <code>mm.m21</code> equal 0.0f. Let <code>cpt1</code> equal <code>currPt + p.control_pt1() * mm</code>. Let <code>cpt2</code> equal <code>currPt + p.control_pt1() * mm + p.control_pt2() * mm</code>. Creates a cubic Bézier curve from <code>currPt</code> to <code>ept</code> using <code>cpt1</code> as the curve's first control point and <code>cpt2</code> as the curve's second control point. Sets <code>currPt</code> to <code>ept</code>.</td>
</tr>
<tr>
<td><code>figure_items::arc</code> p</td>
<td>Let <code>mm</code> equal <code>mtx</code>. Let <code>mm.m20</code> equal 0.0f. Let <code>mm.m21</code> equal 0.0f. Creates an arc. It begins at <code>currPt</code>, which is at <code>p.start_angle()</code> radians on the arc and rotates <code>p.rotation()</code> radians. If <code>p.rotation()</code> is positive, rotation is counterclockwise, otherwise it is clockwise. The center of the arc is located at <code>p.center(currPt, mm)</code>. The arc ends at <code>p.end_pt(currPt, mm)</code>. Sets <code>currPt</code> to <code>p.end_pt(currPt, mm)</code>.</td>
</tr>
</tbody>
</table>

### 12.4 Class `basic_interpreted_path` [io2d.pathgroup]

1. The class `basic_interpreted_path` contains the data that result from interpreting (12.3.16) a sequence of `figure_items::figure_item` objects.

2. A `basic_interpreted_path` object is used by most rendering and composing operations.

#### 12.4.1 `basic_interpreted_path` synopsis [io2d.pathgroup.synopsis]

```cpp
namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    class basic_interpreted_path {
```
public:
  // 12.4.2, construct:
  basic_interpreted_path() noexcept;
  explicit basic_interpreted_path(
    const basic_bounding_box<GraphicsSurfaces>& bb);
  template <class Allocator>
  explicit basic_interpreted_path(
    const basic_path_builder<GraphicsSurfaces, Allocator>& pb);
  template <class InputIterator>
  basic_interpreted_path(InputIterator first, InputIterator last);
  explicit basic_interpreted_path(initializer_list<typename
    basic_figure_items<GraphicsSurfaces>::figure_item>> il);
};

12.4.2 basic_interpreted_path constructors

basic_interpreted_path() noexcept;

Effects: Constructs a basic_interpreted_path that contains an empty path.

template <class Allocator>
explicit basic_interpreted_path(const basic_path_builder<GraphicsSurfaces, Allocator>& pb);

Effects: Equivalent to: basic_interpreted_path{ begin(pb), end(pb) }.

template <class InputIterator>
basic_interpreted_path(InputIterator first, InputIterator last);

Effects: Constructs an object of type interpreted_path.

The contained path is as-if it was the result of interpreting a path containing the values of the elements
from first to the last element before last.

explicit basic_interpreted_path(initializer_list<typename
  basic_figure_items<GraphicsSurfaces>::figure_item> il);

Effects: <TODO>

12.5 Class basic_path_builder

The class basic_path_builder is a container that stores and manipulates objects of type figure_items:figure_item from which interpreted_path objects are created.

A basic_path_builder is a contiguous container. (See [container.requirements.general] in C++ 2017.)
The collection of figure_items:figure_item objects in a path builder is referred to as its path.

12.5.1 basic_path_builder synopsis

namespace std::experimental::io2d::v1 {
  template <class GraphicsSurfaces,
    class Allocator = ::std::allocator<typename
    basic_figure_items<GraphicsSurfaces>::figure_item>>
  class basic_path_builder {
  public:
    using value_type = typename basic_figure_items<GraphicsSurfaces>::figure_item;
    using allocator_type = Allocator;
    using reference = value_type&;
    using const_reference = const value_type&;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
}
// 12.5.3, construct, copy, move, destroy:
basic_path_builder() noexcept(noexcept(Allocator()));
explicit basic_path_builder(const Allocator&) noexcept;
explicit basic_path_builder(size_type n, const Allocator& = Allocator());
basic_path_builder(size_type n, const value_type& value, const Allocator& = Allocator());
template <class InputIterator>
basic_path_builder(InputIterator first, InputIterator last, const Allocator& = Allocator());
basic_path_builder(const basic_path_builder& x);
basic_path_builder(basic_path_builder&&) noexcept;
basic_path_builder(const basic_path_builder&, const Allocator&);
basic_path_builder(basic_path_builder&&, const Allocator&);
basic_path_builder(initializer_list<value_type>, const Allocator& = Allocator());
~basic_path_builder();
basic_path_builder& operator=(const basic_path_builder& x);
basic_path_builder& operator=(basic_path_builder&& x) noexcept(
    allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
    allocator_traits<Allocator>::is_always_equal::value);
basic_path_builder& operator=(initializer_list<value_type>);
~basic_path_builder();

// 12.5.4, capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
size_type capacity() const noexcept;
void resize(size_type sz);
void resize(size_type sz, const value_type& c);
void reserve(size_type n);
void shrink_to_fit();

// element access:
reference operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference at(size_type n);
reference front();
const_reference front() const;
reference back();
const_reference back() const;

// 12.5.5, modifiers:
void new_figure(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& pt) noexcept;
void rel_new_figure(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& pt) noexcept;
void close_figure() noexcept;
void matrix(const basic_matrix_2d<typename GraphicsSurfaces::graphics_math_type>& m) noexcept;
void rel_matrix(const basic_matrix_2d<typename GraphicsSurfaces::graphics_math_type>& m) noexcept;
void revert_matrix() noexcept;
void line(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& pt) noexcept;
void rel_line(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& dpt) noexcept;
void quadratic_curve(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& pt0, const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& pt2) noexcept;
void rel_quadratic_curve(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& pt0, const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& pt2) noexcept;
void cubic_curve(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& pt0, const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& pt1, const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& pt2) noexcept;

void rel_cubic_curve(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& dpt0, const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& dpt1, const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& dpt2) noexcept;

void arc(const basic_point_2d<typename GraphicsSurfaces::graphics_math_type>& rad, float rot, float sang = pi<float>) noexcept;

template <class... Args>
reference emplace_back(Args&&... args);
void push_back(const value_type& x);
void push_back(value_type&& x);
void pop_back();

template <class... Args>
iterator emplace(const_iterator position, Args&&... args);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
iterator insert(const_iterator position, size_type n, const value_type& x);

template <class InputIterator>
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void swap(basic_path_builder&) noexcept;

// 12.5.6, iterators:
iterator begin() noexcept;
const_iterator begin() const noexcept;
const_iterator cbegin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cend() const noexcept;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
const_reverse_iterator crbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
const_reverse_iterator crend() const noexcept;

12.5.2 basic_path_builder container requirements

[io2d.pathbuilder.containerrequirements]

This class is a sequence container, as defined in [containers] in C++ 2017, and all sequence container requirements that apply specifically to vector shall also apply to this class.
12.5.3 basic_path_builder constructors, copy, and assignment

basic_path_builder() noexcept(noexcept(Allocator()));

   Effects: Constructs an empty basic_path_builder.

explicit basic_path_builder(const Allocator&);

   Effects: Constructs an empty basic_path_builder, using the specified allocator.

   Complexity: Constant.

explicit basic_path_builder(size_type n, const Allocator& = Allocator());

   Effects: Constructs a basic_path_builder with n default-inserted elements using the specified allocator.

   Complexity: Linear in n.

basic_path_builder(size_type n, const value_type& value, const Allocator& = Allocator());

   Requires: value_type shall be CopyInsertable into *this.

   Effects: Constructs a basic_path_builder with n copies of value, using the specified allocator.

   Complexity: Linear in n.

template <class InputIterator>
basic_path_builder(InputIterator first, InputIterator last, const Allocator& = Allocator());

   Effects: Constructs a basic_path_builder equal to the range [first, last), using the specified allocator.

   Complexity: Makes only N calls to the copy constructor of value_type (where N is the distance between first and last) and no reallocations if iterators first and last are of forward, bidirectional, or random access categories. It makes order \( N \) calls to the copy constructor of value_type and order \( \log(N) \) reallocations if they are just input iterators.

12.5.4 basic_path_builder capacity

size_type capacity() const noexcept;

   Returns: The total number of elements that the path builder can hold without requiring reallocation.

void reserve(size_type n);

   Requires: value_type shall be MoveInsertable into *this.

   Effects: A directive that informs a path builder of a planned change in size, so that it can manage the storage allocation accordingly. After reserve(), capacity() is greater or equal to the argument of reserve if reallocation happens; and equal to the previous value of capacity() otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of reserve(). If an exception is thrown other than by the move constructor of a non-CopyInsertable type, there are no effects.

   Complexity: It does not change the size of the sequence and takes at most linear time in the size of the sequence.

   Throws: length_error if \( n > \max\text{-}size() \).

Remarks: Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. No reallocation shall take place during insertions that happen after a call to reserve() until the time when an insertion would make the size of the vector greater than the value of capacity().

void shrink_to_fit();

   Requires: value_type shall be MoveInsertable into *this.

   Effects: shrink_to_fit is a non-binding request to reduce capacity() to size(). [ Note: The request is non-binding to allow latitude for implementation-specific optimizations. — end note ] It does not
increase capacity(), but may reduce capacity() by causing reallocation. If an exception is thrown other than by the move constructor of a non-CopyInsertable value_type there are no effects.

**Complexity:** Linear in the size of the sequence.

**Remarks:** Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. If no reallocation happens, they remain valid.

```cpp
void swap(basic_path_builder&)
  noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
           allocator_traits<Allocator>::is_always_equal::value);
Effects: Exchanges the contents and capacity() of *this with that of x.
Complexity: Constant time.
```

```cpp
void resize(size_type sz);
Effects: If sz < size(), erases the last size() - sz elements from the sequence. Otherwise, appends sz - size() default-inserted elements to the sequence.
Requires: value_type shall be MoveInsertable and DefaultInsertable into *this.
Remarks: If an exception is thrown other than by the move constructor of a non-CopyInsertable value_type there are no effects.
```

```cpp
void resize(size_type sz, const value_type& c);
Effects: If sz < size(), erases the last size() - sz elements from the sequence. Otherwise, appends sz - size() copies of c to the sequence.
Requires: value_type shall be CopyInsertable into *this.
Remarks: If an exception is thrown there are no effects.
```

### 12.5.5 basic_path_builder modifiers

**new_figure(point_2d pt) noexcept;**
- **Effects:** Adds a figure_items::figure_item object constructed from figure_items::abs_new_figure(pt) to the end of the path.

**rel_new_figure(point_2d pt) noexcept;**
- **Effects:** Adds a figure_items::figure_item object constructed from figure_items::rel_new_figure(pt) to the end of the path.

**close_figure() noexcept;**
- **Requires:** The current point contains a value.
- **Effects:** Adds a figure_items::figure_item object constructed from figure_items::close_figure() to the end of the path.

**matrix(const matrix_2d& m) noexcept;**
- **Requires:** The matrix m shall be invertible.
- **Effects:** Adds a figure_items::figure_item object constructed from (figure_items::abs_matrix(m)) to the end of the path.

**rel_matrix(const matrix_2d& m) noexcept;**
- **Requires:** The matrix m shall be invertible.
- **Effects:** Adds a figure_items::figure_item object constructed from (figure_items::rel_matrix(m)) to the end of the path.

**revert_matrix() noexcept;**
- **Effects:** Adds a figure_items::figure_item object constructed from (figure_items::revert_matrix()) to the end of the path.
void line(point_2d pt) noexcept;

   Adds a figure_items::figure_item object constructed from figure_items::abs_line(pt) to the end of the path.

void rel_line(point_2d dpt) noexcept;

   Effects: Adds a figure_items::figure_item object constructed from figure_items::rel_line(pt) to the end of the path.

void quadratic_curve(point_2d pt0, point_2d pt1) noexcept;

   Effects: Adds a figure_items::figure_item object constructed from figure_items::abs_quadratic_curve(pt0, pt1) to the end of the path.

void rel_quadratic_curve(point_2d dpt0, point_2d dpt1) noexcept;

   Effects: Adds a figure_items::figure_item object constructed from figure_items::rel_quadratic_curve(dpt0, dpt1) to the end of the path.

void cubic_curve(point_2d pt0, point_2d pt1, point_2d pt2) noexcept;

   Effects: Adds a figure_items::figure_item object constructed from figure_items::abs_cubic_curve(pt0, pt1, pt2) to the end of the path.

void rel_cubic_curve(point_2d dpt0, point_2d dpt1, point_2d dpt2) noexcept;

   Effects: Adds a figure_items::figure_item object constructed from figure_items::rel_cubic_curve(dpt0, dpt1, dpt2) to the end of the path.

void arc(point_2d rad, float rot, float sang) noexcept;

   Effects: Adds a figure_items::figure_item object constructed from figure_items::arc(rad, rot, sang) to the end of the path.


 Remarks: Causes reallocation if the new size is greater than the old capacity. Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. If no reallocation happens, all the iterators and references before the insertion point remain valid. If an exception is thrown other than by the copy constructor, move constructor, assignment operator, or move assignment operator of value_type or by any InputIterator operation there are no effects. If an exception is thrown while inserting a single element at the end and value_type is CopyInsertable or is_nothrow_move_constructible_v<value_type> is true, there are no effects. Otherwise, if an exception is thrown by the move constructor of a non-CopyInsertable value_type, the effects are unspecified.

 Complexity: The complexity is linear in the number of elements inserted plus the distance to the end of the path builder.

 iterator erase(const_iterator position);
 iterator erase(const_iterator first, const_iterator last);
 void pop_back();

   Effects: Invalidates iterators and references at or after the point of the erase.

§ 12.5.5
Complexity: The destructor of `value_type` is called the number of times equal to the number of the elements erased, but the assignment operator of `value_type` is called the number of times equal to the number of elements in the path builder after the erased elements.

Throws: Nothing unless an exception is thrown by the copy constructor, move constructor, assignment operator, or move assignment operator of `value_type`.

### 12.5.6 basic_path_builder iterators

- `iterator begin() noexcept;
  const_iterator begin() const noexcept;
  const_iterator cbegin() const noexcept;

  Returns: An iterator referring to the first `figure_items::figure_item` item in the path.

- `iterator end() noexcept;
  const_iterator end() const noexcept;
  const_iterator cend() const noexcept;

  Returns: An iterator which is the past-the-end value.

- `reverse_iterator rbegin() noexcept;
  const_reverse_iterator rbegin() const noexcept;
  const_reverse_iterator crbegin() const noexcept;

  Returns: An iterator which is semantically equivalent to `reverse_iterator(end)`.

- `reverse_iterator rend() noexcept;
  const_reverse_iterator rend() const noexcept;
  const_reverse_iterator crend() const noexcept;

  Returns: An iterator which is semantically equivalent to `reverse_iterator(begin)`.

### 12.5.7 basic_path_builder specialized algorithms

```cpp
template <class Allocator>
void swap(basic_path_builder<Allocator>& lhs, basic_path_builder<Allocator>& rhs)
  noexcept(noexcept(lhs.swap(rhs))))
{
  Effects: As if by `lhs.swap(rhs)`.
}
```
13  Brushes

13.1  Overview of brushes

1 Brushes contain visual data and serve as sources of visual data for rendering and composing operations.
2 There are four types of brushes:

(2.1) — solid color;
(2.2) — linear gradient;
(2.3) — radial gradient; and,
(2.4) — surface.
3 Once a brush is created, its visual data is immutable.
4 [Note: While copy and move operations along with a swap operation can change the visual data that a brush contains, the visual data itself is not modified. —end note]
5 A brush is used either as a source brush or a mask brush (15.3.2.2).
6 When a brush is used in a rendering and composing operation, if it is used as a source brush, it has a brush_props object that describes how the brush is interpreted for purposes of sampling. If it is used as a mask brush, it has a mask_props object that describes how the brush is interpreted for purposes of sampling.

7 The basic_brush_props (14.10.1) and basic_mask_props (14.14.1) classes each have a wrap mode and a filter. The basic_brush_props class also has a brush matrix and a fill rule. The basic_mask_props class also has a mask matrix. Where possible, the terms that are common between the two classes are referenced without regard to whether the brush is being used as a source brush or a mask brush.
8 Solid color brushes are unbounded and as such always produce the same visual data when sampled from, regardless of the requested point.
9 Linear gradient and radial gradient brushes share similarities with each other that are not shared by the other types of brushes. This is discussed in more detail elsewhere (13.2).
10 Surface brushes are constructed from a basic_image_surface object. Their visual data is raster graphics data, which has implications on sampling from the brush that are not present in the other brush types.

13.2  Gradient brushes

13.2.1  Common properties of gradients

1 Gradients are formed, in part, from a collection of gradient_stop objects.
2 The collection of gradient_stop objects contribute to defining a brush which, when sampled from, returns a value that is interpolated based on those gradient stops.

13.2.2  Linear gradients

1 A linear gradient is a type of gradient.
2 A linear gradient has a begin point and an end point, each of which are objects of type basic_point_2d.
3 A linear gradient for which the distance between its begin point and its end point is <TODO>basic_point_2d::zero() is a degenerate linear gradient.
4 All attempts to sample from a a degenerate linear gradient return the color rgba_color::transparent_black. The remainder of 13.2 is inapplicable to degenerate linear gradients. [Note: Because a point has no width and this case is only met when the distance is between the begin point and the end point is zero (such that it collapses to a single point), the existence of one or more gradient stops is irrelevant. A linear gradient requires a line segment to define its color(s). Without a line segment, it is not a linear gradient. —end note]
5 The begin point and end point of a linear gradient define a line segment, with a gradient stop offset value of 0.0f corresponding to the begin point and a gradient stop offset value of 1.0f corresponding to the end point.
6 Gradient stop offset values in the range [0.0f, 1.0f] linearly correspond to points on the line segment.

§ 13.2.2  122
Example: Given a linear gradient with a begin point of \( <\text{TODO}>\text{basic_point}_2d(0.0f, 0.0f) \) and an end point of \( <\text{TODO}>\text{basic_point}_2d(10.0f, 5.0f) \), a gradient stop offset value of 0.6f would correspond to the point \( <\text{TODO}>\text{basic_point}_2d(6.0f, 3.0f) \). —end example

To determine the offset value of a point \( p \) for a linear gradient, perform the following steps:

a) Create a line at the begin point of the linear gradient, the begin line, and another line at the end point of the linear gradient, the end line, with each line being perpendicular to the gradient line segment, which is the line segment delineated by the begin point and the end point.

b) Using the begin line, \( p \), and the end line, create a line, the \( p \) line, which is parallel to the gradient line segment.

c) Defining \( dp \) as the distance between \( p \) and the point where the \( p \) line intersects the begin line and \( dt \) as the distance between the point where the \( p \) line intersects the begin line and the point where the \( p \) line intersects the end line, the offset value of \( p \) is \( dp \div dt \).

d) The offset value shall be negative if

(8.1) \( p \) is not on the line segment delineated by the point where the \( p \) line intersects the begin line and the point where the \( p \) line intersects the end line; and,

(8.2) the distance between \( p \) and the point where the \( p \) line intersects the begin line is less than the distance between \( p \) and the point where the \( p \) line intersects the end line.

### 13.2.3 Radial gradients

A radial gradient is a type of gradient.

A radial gradient has a start circle and an end circle, each of which is defined by a basic_circle object.

A radial gradient is a degenerate radial gradient if:

(3.1) its start circle has a negative radius; or,

(3.2) its end circle has a negative radius; or,

(3.3) the distance between the center point of its start circle and the center point of its end circle is \( <\text{TODO}>\text{basic_point}_2d::\text{zero}() \); or,

(3.4) its start circle has a radius of 0.0f and its end circle has a radius of 0.0f.

All attempts to sample from a brush object created using a degenerate radial gradient return the color rgba_color::transparent_black. The remainder of 13.2 is inapplicable to degenerate radial gradients.

A gradient stop offset of 0.0f corresponds to all points along the diameter of the start circle or to its center point if it has a radius value of 0.0f.

A gradient stop offset of 1.0f corresponds to all points along the diameter of the end circle or to its center point if it has a radius value of 0.0f.

A radial gradient shall be rendered as a continuous series of interpolated circles defined by the following equations:

\[
\begin{align*}
  x(o) &= x_{\text{start}} + o \times (x_{\text{end}} - x_{\text{start}}) \\
  y(o) &= y_{\text{start}} + o \times (y_{\text{end}} - y_{\text{start}}) \\
  \text{radius}(o) &= \text{radius}_{\text{start}} + o \times (\text{radius}_{\text{end}} - \text{radius}_{\text{start}})
\end{align*}
\]

where \( o \) is a gradient stop offset value.

The range of potential values for \( o \) shall be determined by the wrap mode (13.1):

(8.1) For wrap_mode::none, the range of potential values for \( o \) is \([0, 1]\).

(8.2) For all other wrap_mode values, the range of potential values for \( o \) is \([\text{numeric_limits<float>::lowest()},\text{numeric_limits<float>::max()}]\).

The interpolated circles shall be rendered starting from the smallest potential value of \( o \).

An interpolated circle shall not be rendered if its value for \( o \) results in \( \text{radius}(o) \) evaluating to a negative value.

### 13.2.4 Sampling from gradients

For any offset value \( o \), its color value shall be determined according to the following rules:
a) If there are less than two gradient stops or if all gradient stops have the same offset value, then the color value of every offset value shall be \texttt{rgba\_color::transparent\_black} and the remainder of these rules are inapplicable.

b) If exactly one gradient stop has an offset value equal to \( a \), \( a \)'s color value shall be the color value of that gradient stop and the remainder of these rules are inapplicable.

c) If two or more gradient stops have an offset value equal to \( a \), \( a \)'s color value shall be the color value of the gradient stop which has the lowest index value among the set of gradient stops that have an offset value equal to \( a \) and the remainder of \( 13.2.4 \) is inapplicable.

d) When no gradient stop has the offset value of \( 0.0f \), then, defining \( n \) to be the offset value that is nearest to \( 0.0f \) among the offset values in the set of all gradient stops, if \( o \) is in the offset range \([0, n)\), \( o \)'s color value shall be \texttt{rgba\_color::transparent\_black} and the remainder of these rules are inapplicable. [Note: Since the range described does not include \( n \), it does not matter how many gradient stops have \( n \) as their offset value for purposes of this rule. — end note]

e) When no gradient stop has the offset value of \( 1.0f \), then, defining \( n \) to be the offset value that is nearest to \( 1.0f \) among the offset values in the set of all gradient stops, if \( o \) is in the offset range \((n, 1]\), \( o \)'s color value shall be \texttt{rgba\_color::transparent\_black} and the remainder of these rules are inapplicable. [Note: Since the range described does not include \( n \), it does not matter how many gradient stops have \( n \) as their offset value for purposes of this rule. — end note]

f) Each gradient stop has, at most, two adjacent gradient stops: one to its left and one to its right.

g) Adjacency of gradient stops is initially determined by offset values. If two or more gradient stops have the same offset value then index values are used to determine adjacency as described below.

h) For each gradient stop \( a \), the \textit{set of gradient stops to its left} are those gradient stops which have an offset value which is closer to \( 0.0f \) than \( a \)'s offset value. [Note: This includes any gradient stops with an offset value of \( 0.0f \) provided that \( a \)'s offset value is not \( 0.0f \). — end note]

i) For each gradient stop \( b \), the \textit{set of gradient stops to its right} are those gradient stops which have an offset value which is closer to \( 1.0f \) than \( b \)'s offset value. [Note: This includes any gradient stops with an offset value of \( 1.0f \) provided that \( b \)'s offset value is not \( 1.0f \). — end note]

j) A gradient stop which has an offset value of \( 0.0f \) does not have an adjacent gradient stop to its left.

k) A gradient stop which has an offset value of \( 1.0f \) does not have an adjacent gradient stop to its right.

l) If a gradient stop \( a \)'s set of gradient stops to its left consists of exactly one gradient stop, that gradient stop is the gradient stop that is adjacent to \( a \) on its left.

m) If a gradient stop \( b \)'s set of gradient stops to its right consists of exactly one gradient stop, that gradient stop is the gradient stop that is adjacent to \( b \) on its right.

n) If two or more gradient stops have the same offset value then the gradient stop with the lowest index value is the only gradient stop from that set of gradient stops which can have a gradient stop that is adjacent to it on its left and the gradient stop with the highest index value is the only gradient stop from that set of gradient stops which can have a gradient stop that is adjacent to it on its right. This rule takes precedence over all of the remaining rules.

o) If a gradient stop can have an adjacent gradient stop to its left, then the gradient stop which is adjacent to it to its left is the gradient stop from the set of gradient stops to its left which has an offset value which is closest to its offset value. If two or more gradient stops meet that criteria, then the gradient stop which is adjacent to it to its left is the gradient stop which has the highest index value from the set of gradient stops to its left which are tied for being closest to its offset value.

p) If a gradient stop can have an adjacent gradient stop to its right, then the gradient stop which is adjacent to it to its right is the gradient stop from the set of gradient stops to its right which has an offset value which is closest to its offset value. If two or more gradient stops meet that criteria, then the gradient stop which is adjacent to it to its right is the gradient stop which has the lowest index value from the set of gradient stops to its right which are tied for being closest to its offset value.

q) Where the value of \( o \) is in the range \([0, 1]\), its color value shall be determined by interpolating between the gradient stop, \( r \), which is the gradient stop whose offset value is closest to \( o \) without being less than \( o \) and which can have an adjacent gradient stop to its left, and the gradient stop that is adjacent to \( r \) on \( r \)'s left. The acceptable forms of interpolating between color values is set forth later in this section.
r) Where the value of \( o \) is outside the range \([0, 1]\), its color value depends on the value of wrap mode:

1. If wrap mode is `wrap_mode::none`, the color value of \( o \) shall be `rgba_color::transparent_black`.  
2. If wrap mode is `wrap_mode::pad`, if \( o \) is negative then the color value of \( o \) shall be the same as-if the value of \( o \) was 0.0f, otherwise the color value of \( o \) shall be the same as-if the value of \( o \) was 1.0f.
3. If wrap mode is `wrap_mode::repeat`, then 1.0f shall be added to or subtracted from \( o \) until \( o \) is in the range \([0, 1]\), at which point its color value is the color value for the modified value of \( o \) as determined by these rules. [Example: Given \( o == 2.1 \), after application of this rule \( o == 0.1 \) and the color value of \( o \) shall be the same value as-if the initial value of \( o \) was 0.1. Given \( o == -0.3 \), after application of this rule \( o == 0.7 \) and the color value of \( o \) shall be the same as-if the initial value of \( o \) was 0.7. —end example]
4. If wrap mode is `wrap_mode::reflect`, \( o \) shall be set to the absolute value of \( o \), then 2.0f shall be subtracted from \( o \) until \( o \) is in the range \([0, 1]\) then if \( o \) is in the range \((1, 2)\) then \( o \) shall be set to \( 1.0f - (o - 1.0f) \), at which point its color value is the color value for the modified value of \( o \) as determined by these rules. [Example: Given \( o == 2.8 \), after application of this rule \( o == 0.8 \) and the color value of \( o \) shall be the same value as-if the initial value of \( o \) was 0.8. Given \( o == 3.6 \), after application of this rule \( o == 0.4 \) and the color value of \( o \) shall be the same value as-if the initial value of \( o \) was 0.4. Given \( o == -0.3 \), after application of this rule \( o == 0.3 \) and the color value of \( o \) shall be the same as-if the initial value of \( o \) was 0.3. Given \( o == -5.8 \), after application of this rule \( o == 0.2 \) and the color value of \( o \) shall be the same as-if the initial value of \( o \) was 0.2. —end example]

Interpolation between the color values of two adjacent gradient stops is performed linearly on each color channel.

13.3 Enum class `wrap_mode`  

13.3.1 `wrap_mode` summary  

The `wrap_mode` enum class describes how a point’s visual data is determined if it is outside the bounds of the source brush (15.3.2.2) when sampling. Depending on the source brush’s `filter` value, the visual data of several points may be required to determine the appropriate visual data value for the point that is being sampled. In this case, each point is sampled according to the source brush’s `wrap_mode` value with two exceptions:

1. If the point to be sampled is within the bounds of the source brush and the source brush’s `wrap_mode` value is `wrap_mode::none`, then if the source brush’s `filter` value requires that one or more points which are outside of the bounds of the source brush be sampled, each of those points is sampled as-if the source brush’s `wrap_mode` value is `wrap_mode::pad` rather than `wrap_mode::none`.  
2. If the point to be sampled is within the bounds of the source brush and the source brush’s `wrap_mode` value is `wrap_mode::none`, then if the source brush’s `filter` value requires that one or more points which are inside of the bounds of the source brush be sampled, each of those points is sampled such that the visual data that is returned is the equivalent of `rgba_color::transparent_black`.  

If a point to be sampled does not have a defined visual data element and the search for the nearest point with defined visual data produces two or more points with defined visual data that are equidistant from the point to be sampled, the returned visual data shall be an unspecified value which is the visual data of one of those equidistant points. Where possible, implementations should choose the among the equidistant points that have an \( x \) axis value and a \( y \) axis value that is nearest to 0.0f.

See Table 19 for the meaning of each `wrap_mode` enumerator.

namespace std::experimental::io2d::v1 {
  enum class `wrap_mode` {
    none, repeat, reflect,
  }
13.3.3 \textit{wrap\_mode} enumerators

Table 19 — \textit{wrap\_mode} enumerator meanings

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>If the point to be sampled is outside of the bounds of the source brush, the visual data that is returned is the equivalent of \texttt{rgba_color::transparent_black}.</td>
</tr>
<tr>
<td>repeat</td>
<td>If the point to be sampled is outside of the bounds of the source brush, the visual data that is returned is the visual data that would have been returned if the source brush was infinitely large and repeated itself in a left-to-right-left-to-right and top-to-bottom-top-to-bottom fashion.</td>
</tr>
<tr>
<td>reflect</td>
<td>If the point to be sampled is outside of the bounds of the source brush, the visual data that is returned is the visual data that would have been returned if the source brush was infinitely large and repeated itself in a left-to-right-to-left-to-right and top-to-bottom-to-top-to-bottom fashion.</td>
</tr>
<tr>
<td>pad</td>
<td>If the point to be sampled is outside of the bounds of the source brush, the visual data that is returned is the visual data that would have been returned for the nearest defined point that is in inside the bounds of the source brush.</td>
</tr>
</tbody>
</table>

13.4 \textit{Enum} class \textit{filter}

13.4.1 \textit{filter} summary

1 The \textit{filter} enum class specifies the type of filter to use when sampling from raster graphics data.

2 Three of the \textit{filter} enumerators, \texttt{filter::fast}, \texttt{filter::good}, and \texttt{filter::best}, specify desired characteristics of the filter, leaving the choice of a specific filter to the implementation. The other two, \texttt{filter::nearest} and \texttt{filter::bilinear}, each specify a particular filter that shall be used.

3 \textit{Note:} The only type of brush that has raster graphics data as its visual data is a brush with a brush type of \texttt{brush\_type::surface}. \textit{— end note}

4 See Table 20 for the meaning of each \textit{filter} enumerator.

13.4.2 \textit{filter} synopsis

```cpp
namespace std::experimental::io2d::v1 {

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

13.4.3 \textit{filter} enumerators
Table 20 — filter enumerator meanings

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>fast</td>
<td>The filter that corresponds to this value is implementation-defined. The implementation shall ensure that the time complexity of the chosen filter is not greater than the time complexity of the filter that corresponds to filter::good. [Note: By choosing this value, the user is hinting that performance is more important than quality. —end note]</td>
</tr>
<tr>
<td>good</td>
<td>The filter that corresponds to this value is implementation-defined. The implementation shall ensure that the time complexity of the chosen formula is not greater than the time complexity of the formula for filter::best. [Note: By choosing this value, the user is hinting that quality and performance are equally important. —end note]</td>
</tr>
<tr>
<td>best</td>
<td>The filter that corresponds to this value is implementation-defined. [Note: By choosing this value, the user is hinting that quality is more important than performance. —end note]</td>
</tr>
<tr>
<td>nearest</td>
<td>Nearest-neighbor interpolation filtering</td>
</tr>
<tr>
<td>bilinear</td>
<td>Bilinear interpolation filtering</td>
</tr>
</tbody>
</table>

13.5 Enum class brush_type

13.5.1 brush_type summary

The brush_type enum class denotes the type of a brush object.

See Table 21 for the meaning of each brush_type enumerator.

13.5.2 brush_type synopsis

```cpp
namespace std::experimental::io2d::v1 {
    enum class brush_type {
        solid_color,
        surface,
        linear,
        radial
    };
}
```

13.5.3 brush_type enumerators

Table 21 — brush_type enumerator meanings

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>solid_color</td>
<td>The brush object is a solid color brush.</td>
</tr>
<tr>
<td>surface</td>
<td>The brush object is a surface brush.</td>
</tr>
<tr>
<td>linear</td>
<td>The brush object is a linear gradient brush.</td>
</tr>
<tr>
<td>radial</td>
<td>The brush object is a radial gradient brush.</td>
</tr>
</tbody>
</table>

13.6 Class gradient_stop

13.6.1 Overview

The class gradient_stop describes a gradient stop that is used by gradient brushes.

It has an offset of type float and an offset color of type rgba_color.
13.6.2 gradient_stop synopsis

namespace std::experimental::io2d::v1 {
    class gradient_stop {
    public:
        // 13.6.3, construct:
        constexpr gradient_stop() noexcept;
        constexpr gradient_stop(float o, rgba_color c) noexcept;

        // 13.6.4, modifiers:
        constexpr void offset(float o) noexcept;
        constexpr void color(rgba_color c) noexcept;

        // 13.6.5, observers:
        constexpr float offset() const noexcept;
        constexpr rgba_color color() const noexcept;
    };
    // 13.6.6, operators:
    constexpr bool operator==(const gradient_stop& lhs, const gradient_stop& rhs)
    noexcept;
    constexpr bool operator!=(const gradient_stop& lhs, const gradient_stop& rhs)
    noexcept;
};

13.6.3 gradient_stop constructors

constexpr gradient_stop() noexcept;

Effects: Equivalent to: gradient_stop(0.0f, rgba_color::transparent_black).

constexpr gradient_stop(float o, rgba_color c) noexcept;

Requires: o >= 0.0f and o <= 1.0f.

Effects: Constructs a gradient_stop object.

The offset is o rounded to the nearest multiple of 0.00001f. The offset color is c.

13.6.4 gradient_stop modifiers

constexpr void offset(float o) noexcept;

Requires: o >= 0.0f and o <= 1.0f.

Effects: The offset is o rounded to the nearest multiple of 0.00001f.

constexpr void color(rgba_color c) noexcept;

Effects: The offset color is c.

13.6.5 gradient_stop observers

constexpr float offset() const noexcept;

Returns: The offset.

constexpr rgba_color color() const noexcept;

Returns: The offset color.

13.6.6 gradient_stop operators

constexpr bool operator==(const gradient_stop& lhs, const gradient_stop& rhs)
    noexcept;

Returns: lhs.offset() == rhs.offset() && lhs.color() == rhs.color();

13.7 Class template basic_brush

13.7.1 Summary

The class template basic_brush describes an opaque wrapper for visual data. It takes one type parameter, which is a GraphicsSurfaces.
A basic_brush object is usable with any basic_image_surface basic_output_surface, and basic_unmanaged_output_surface object provided that they have the same GraphicsSurfaces as the basic_brush object.

A basic_brush object’s visual data is immutable. It is observable only by the effect that it produces when the brush is used as a source brush or as a mask brush (15.3.2.2).

A basic_brush object has a brush type of brush_type, which indicates which type of brush it is (Table 21).

As a result of technological limitations, a basic_brush object’s visual data may have less precision than the data from which it was created.

The data are stored in an object of type typename GraphicsMath::brushes::brush_data_type.

### 13.7.2 Synopsis

```cpp
namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    class basic_brush {
        public:
            using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
            using data_type = typename GraphicsSurfaces::brushes::brush_data_type;

            // 13.7.4, constructors:
            explicit basic_brush(const rgba_color& c);
            template <class InputIterator>
            basic_brush(const basic_point_2d<graphics_math_type>& begin,
                         const basic_point_2d<graphics_math_type>& end,
                         InputIterator first, InputIterator last);
            basic_brush(const basic_point_2d<graphics_math_type>& begin,
                         const basic_point_2d<graphics_math_type>& end,
                         ::std::initializer_list<gradient_stop> il);
            template <class InputIterator>
            basic_brush(const basic_circle<graphics_math_type>& start,
                         const basic_circle<graphics_math_type>& end,
                         InputIterator first, InputIterator last);
            basic_brush(const basic_circle<graphics_math_type>& start,
                         const basic_circle<graphics_math_type>& end,
                         ::std::initializer_list<gradient_stop> il);
            basic_brush(basic_image_surface<GraphicsSurfaces>&& img);

            // 13.7.5, accessors:
            const data_type& data() const noexcept;
            brush_type type() const noexcept;
    };
}
```

### 13.7.3 Sampling from a basic_brush object

1. A basic_brush object is sampled from either as a source brush (15.3.2.2) or a mask brush (15.3.2.2).
2. If it is being sampled from as a source brush, its wrap mode, filter, and brush matrix are defined by a basic_brush_props object (15.3.2.4 and 15.3.2.6).
3. If it is being sampled from as a mask brush, its wrap mode, filter, and mask matrix are defined by a basic_mask_props object (15.3.2.5 and 15.3.2.6).
4. When sampling from a basic_brush object b, the brush_type returned by calling b.type() determines how the results of sampling are determined:
   1. If the result of b.type() is brush_type::solid_color then b is a solid color brush.
   2. If the result of b.type() is brush_type::surface then b is a surface brush.
   3. If the result of b.type() is brush_type::linear then b is a linear gradient brush.
   4. If the result of b.type() is brush_type::radial then b is a radial gradient brush.
13.7.3.1 Sampling from a solid color brush

When \( b \) is a solid color brush, then when sampling from \( b \), the visual data returned is always the visual data used to construct \( b \), regardless of the point which is to be sampled and regardless of the return values of wrap mode, filter, and brush matrix or mask matrix.

13.7.3.2 Sampling from a linear gradient brush

When \( b \) is a linear gradient brush, when sampling point \( pt \), where \( pt \) is the return value of calling the transform_pt member function of brush matrix or mask matrix using the requested point, from \( b \), the visual data returned are as specified by 13.2.2 and 13.2.4.

13.7.3.3 Sampling from a radial gradient brush

When \( b \) is a radial gradient brush, when sampling point \( pt \), where \( pt \) is the return value of calling the transform_pt member function of brush matrix or mask matrix using the requested point, from \( b \), the visual data are as specified by 13.2.3 and 13.2.4.

13.7.3.4 Sampling from a surface brush

When \( b \) is a surface brush, when sampling point \( pt \) from \( b \), where \( pt \) is the return value of calling the transform_pt member function of the brush matrix or mask matrix using the requested point, the visual data returned are from the point \( pt \) in the raster graphics data of the brush, as modified by the values of wrap mode (13.3) and filter (13.4).

13.7.4 Constructors

**explicit basic_brush(const rgba_color& c);**

Effects: Constructs an object of type basic_brush.

Postconditions: \( \text{data()} == \text{GraphicsSurfaces::brushes::create_brush}(c) \).

Remarks: The visual data format of the visual data are as-if it is that specified by format::argb32.

Sampling from the brush produces the results specified in 13.7.3.1.

**template <class InputIterator> basic_brush(const basic_point_2d<graphics_math_type>& begin, const basic_point_2d<graphics_math_type>& end, InputIterator first, InputIterator last);**

Effects: Constructs an object of type basic_brush.

Postconditions: \( \text{data()} == \text{GraphicsSurfaces::brushes::create_brush}(\text{begin, end, first, last}) \).

Remarks: Sampling from this brush produces the results specified in 13.7.3.2.

**basic_brush(const basic_point_2d<graphics_math_type>& begin, const basic_point_2d<graphics_math_type>& end, initializer_list<gradient_stop> il);**

Effects: Constructs an object of type basic_brush.

Postconditions: \( \text{data()} == \text{GraphicsSurfaces::brushes::create_brush}(\text{begin, end, il}) \).

Remarks: Sampling from this brush produces the results specified in 13.7.3.2.

**template <class InputIterator> basic_brush(const basic_circle<graphics_math_type>& start, const basic_circle<graphics_math_type>& end, InputIterator first, InputIterator last);**

Effects: Constructs an object of type basic_brush.

Postconditions: \( \text{data()} == \text{GraphicsSurfaces::brushes::create_brush}(\text{start, end, first, last}) \).

Remarks: Sampling from this brush produces the results specified in 13.7.3.3.

**basic_brush(const basic_circle<graphics_math_type>& start, const basic_circle<graphics_math_type>& end, initializer_list<gradient_stop> il);**

Effects: Constructs an object of type basic_brush.
Postconditions: \texttt{data()} == \texttt{GraphicsSurfaces::brushes::create_brush(start, end, il)}.

Remarks: Sampling from this brush produces the results specified in \texttt{13.7.3.3}.

\begin{verbatim}
basic_brush(basic_image_surface<GraphicsSurfaces>&& img);
\end{verbatim}

Effects: Constructs an object of type \texttt{basic_brush}.

Postconditions: \texttt{data()} == \texttt{GraphicsSurfaces::brushes::create_brush(move(img))}.

Sampling from this brush produces the results specified in \texttt{13.7.3.4}.

\subsection*{13.7.5 Accessors} \[io2d.brush.acc\]

\begin{verbatim}
const data_type& data() const noexcept;
\end{verbatim}

Returns: A reference to the \texttt{basic_brush} object's data object (See \texttt{13.7.1}).

\begin{verbatim}
brush_type type() const noexcept;
\end{verbatim}

Returns: \texttt{GraphicsSurfaces::brushes::get_brush_type(data())}. 

\section*{§ 13.7.5}
14 Surface state props [io2d.surfacestate]

14.1 Overview [io2d.surfacestate.general]
1 In order to produce effects beyond simply drawing raster graphics data or a path to a surface, graphics state data is supplied when performing rendering and composing operations (15.3.2) on surfaces.

2 Surface state types group together related graphics state data. Objects of those types are then supplied as arguments to the functions that carry out rendering and composing operations on surfaces. [Note: This allows surfaces to be stateless, which typically provides significant performance gains on modern graphics acceleration hardware. — end note]

3 The enum class types and surface state class templates that define and provide the graphics state data are described below.

14.2 Enum class antialias [io2d.antialias]

14.2.1 antialias summary [io2d.antialias.summary]
1 The antialias enum class specifies the type of anti-aliasing that the rendering system uses for rendering paths. See Table 22 for the meaning of each antialias enumerator.

14.2.2 antialias synopsis [io2d.antialias.synopsis]
namespace std::experimental::io2d::v1 {
    enum class antialias {
        none,
        fast,
        good,
        best
    };
}

14.2.3 antialias enumerators [io2d.antialias.enumerators]

Table 22 — antialias enumerator meanings

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>No anti-aliasing is performed when performing a rendering operation.</td>
</tr>
<tr>
<td>fast</td>
<td>Some form of anti-aliasing should be used when performing a rendering operation but performance is more important than the quality of the results. The technique used is implementation-defined.</td>
</tr>
<tr>
<td>good</td>
<td>Some form of anti-aliasing should be used when performing a rendering operation and the sacrificing some performance to obtain better anti-aliasing results than would likely be obtained from antialias::fast is acceptable. The technique used is implementation-defined.</td>
</tr>
<tr>
<td>best</td>
<td>Some form of anti-aliasing should be used when performing a rendering operation and better anti-aliasing results than would likely be obtained from antialias::fast and antialias::good are desired even if performance degrades significantly. The technique used is implementation-defined. [Note: This might commonly be chosen when a user is going to render something once and cache the results for repeated use or when a user is rendering something that does not necessarily need performance suitable for real-time computer graphics applications. — end note]</td>
</tr>
</tbody>
</table>

§ 14.2.3
14.3 Enum class `fill_rule`

14.3.1 `fill_rule` summary

The `fill_rule` enum class determines how the filling operation (15.3.5) is performed on a path.

For each point, draw a ray from that point to infinity which does not pass through the start point or end point of any non-degenerate segment in the path, is not tangent to any non-degenerate segment in the path, and is not coincident with any non-degenerate segment in the path.

See Table 23 for the meaning of each `fill_rule` enumerator.

14.3.2 `fill_rule` synopsis

namespace std::experimental::io2d::v1 {
enum class fill_rule {
    winding,
    even_odd
};
}

14.3.3 `fill_rule` enumerators

Table 23 — `fill_rule` enumerator meanings

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>winding</td>
<td>If the <code>fill rule</code> (14.10.1) is <code>fill_rule::winding</code>, then using the ray described above and beginning with a count of zero, add one to the count each time a non-degenerate segment crosses the ray going left-to-right from its begin point to its end point, and subtract one each time a non-degenerate segment crosses the ray going from right-to-left from its begin point to its end point. If the resulting count is zero after all non-degenerate segments that cross the ray have been evaluated, the point shall not be filled; otherwise the point shall be filled.</td>
</tr>
<tr>
<td>even_odd</td>
<td>If the fill rule is <code>fill_rule::even_odd</code>, then using the ray described above and beginning with a count of zero, add one to the count each time a non-degenerate segment crosses the ray. If the resulting count is an odd number after all non-degenerate segments that cross the ray have been evaluated, the point shall be filled; otherwise the point shall not be filled. [Note: Mathematically, zero is an even number, not an odd number. — end note]</td>
</tr>
</tbody>
</table>

14.4 Enum class `line_cap`

14.4.1 `line_cap` summary

The `line_cap` enum class specifies how the ends of lines should be rendered when a `interpreted_path` object is stroked. See Table 24 for the meaning of each `line_cap` enumerator.

14.4.2 `line_cap` synopsis

namespace std::experimental::io2d::v1 {
enum class line_cap {
    none,
    round,
    square
};
}

14.4.3 `line_cap` enumerators
Table 24 — line_cap enumerator meanings

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>The line has no cap. It terminates exactly at the end point.</td>
</tr>
<tr>
<td>round</td>
<td>The line has a circular cap, with the end point serving as the center of the circle and the line width serving as its diameter.</td>
</tr>
<tr>
<td>square</td>
<td>The line has a square cap, with the end point serving as the center of the square and the line width serving as the length of each side.</td>
</tr>
</tbody>
</table>

14.5 Enum class line_join

14.5.1 line_join summary

The line_join enum class specifies how the junction of two line segments should be rendered when a interpreted_path is stroked. See Table 25 for the meaning of each enumerator.

14.5.2 line_join synopsis

namespace std::experimental::io2d::v1 {
    enum class line_join {
        miter,
        round,
        bevel
    };
}

14.5.3 line_join enumerators

Table 25 — line_join enumerator meanings

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>miter</td>
<td>Joins will be mitered or beveled, depending on the miter limit (see: 14.12.1).</td>
</tr>
<tr>
<td>round</td>
<td>Joins will be rounded, with the center of the circle being the join point.</td>
</tr>
<tr>
<td>bevel</td>
<td>Joins will be beveled, with the join cut off at half the line width from the join point. Implementations may vary the cut off distance by an amount that is less than one pixel at each join for aesthetic or technical reasons.</td>
</tr>
</tbody>
</table>

14.6 Enum class compositing_op

14.6.1 compositing_op Summary

The compositing_op enum class specifies composition algorithms. See Table 26, Table 27 and Table 28 for the meaning of each compositing_op enumerator.

14.6.2 compositing_op Synopsis

namespace std::experimental::io2d::v1 {
    enum class compositing_op {
        // basic
        over,
        clear,
        source,
        in,
        out,
        atop,
        dest_over,
        dest_in,
        dest_out,
    };
}
dest_atop,
xor_op,
add,
saturate,
// blend
multiply,
screen,
overlay,
darken,
lighten,
color_dodge,
color_burn,
hard_light,
soft_light,
difference,
exclusion,
// hsl
hsl_hue,
hsl_saturation,
hsl_color,
hsl_luminosity
};

14.6.3 compositing_op Enumerators

The tables below specifies the mathematical formula for each enumerator's composition algorithm. The formulas differentiate between three color channels (red, green, and blue) and an alpha channel (transparency). For all channels, valid channel values are in the range [0.0, 1.0].

1 Where a visual data format for a visual data element has no alpha channel, the visual data format shall be treated as though it had an alpha channel with a value of 1.0 for purposes of evaluating the formulas.

2 Where a visual data format for a visual data element has no color channels, the visual data format shall be treated as though it had a value of 0.0 for all color channels for purposes of evaluating the formulas.

3 The following symbols and specifiers are used:
   The $R$ symbol means the result color value
   The $S$ symbol means the source color value
   The $D$ symbol means the destination color value
   The $c$ specifier means the color channels of the value it follows
   The $a$ specifier means the alpha channel of the value it follows

4 The color symbols $R$, $S$, and $D$ may appear with or without any specifiers.

5 If a color symbol appears alone, it designates the entire color as a tuple in the unsigned normalized form (red, green, blue, alpha).

6 The specifiers $c$ and $a$ may appear alone or together after any of the three color symbols.

7 The presence of the $c$ specifier alone means the three color channels of the color as a tuple in the unsigned normalized form (red, green, blue).

8 The presence of the $a$ specifier alone means the alpha channel of the color in unsigned normalized form.

9 The presence of the $a$ specifier alone means the alpha channel of the color in unsigned normalized form.

10 The presence of the specifiers together in the form $ca$ means the value of the color as a tuple in the unsigned normalized form (red, green, blue, alpha), where the value of each color channel is the product of each color channel and the alpha channel and the value of the alpha channel is the original value of the alpha channel. [Example: When it appears in a formula, $Sca$ means $((Sc \times Sa), Sa)$, such that, given a source color $Sc = (1.0, 0.5, 0.0)$ and an source alpha $Sa = (0.5)$, the value of $Sca$ when specified in one of the formulas would be $Sca = (1.0 \times 0.5, 0.5 \times 0.5, 0.0 \times 0.5, 0.5) = (0.5, 0.25, 0.0, 0.5)$. The same is true for $Dca$ and $Rca$. — end example]

11 No space is left between a value and its channel specifiers. Channel specifiers will be preceded by exactly one value symbol.

12 When performing an operation that involves evaluating the color channels, each color channel should be evaluated individually to produce its own value.

§ 14.6.3

135
The basic enumerators specify a value for `bound`. This value may be 'Yes', 'No', or 'N/A'.

If the bound value is 'Yes', then the source is treated as though it is also a mask. As such, only areas of the surface where the source would affect the surface are altered. The remaining areas of the surface have the same color value as before the compositing operation.

If the bound value is 'No', then every area of the surface that is not affected by the source will become transparent black. In effect, it is as though the source was treated as being the same size as the destination surface with every part of the source that does not already have a color value assigned to it being treated as though it were transparent black. Application of the formula with this precondition results in those areas evaluating to transparent black such that evaluation can be bypassed due to the predetermined outcome.

If the bound value is 'N/A', the operation would have the same effect regardless of whether it was treated as 'Yes' or 'No' such that those bound values are not applicable to the operation. A 'N/A' formula when applied to an area where the source does not provide a value will evaluate to the original value of the destination even if the source is treated as having a value there of transparent black. As such the result is the same as-if the source were treated as being a mask, i.e. 'Yes' and 'No' treatment each produce the same result in areas where the source does not have a value.

If a clip is set and the bound value is 'Yes' or 'N/A', then only those areas of the surface that are within the clip will be affected by the compositing operation.

If a clip is set and the bound value is 'No', then only those areas of the surface that are within the clip will be affected by the compositing operation. Even if no part of the source is within the clip, the operation will still set every area within the clip to transparent black. Areas outside the clip are not modified.

### Table 26 — compositing\_op basic enumerator meanings

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Bound</th>
<th>Color</th>
<th>Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>Yes</td>
<td>(R_c = 0)</td>
<td>(R_a = 0)</td>
</tr>
<tr>
<td>source</td>
<td>Yes</td>
<td>(R_c = C_s)</td>
<td>(R_a = S_a)</td>
</tr>
<tr>
<td>over</td>
<td>N/A</td>
<td>(R_c = \frac{(S_{ca} + D_{ca} \times (1 - S_a))}{R_a})</td>
<td>(R_a = S_a + D_a \times (1 - S_a))</td>
</tr>
<tr>
<td>in</td>
<td>No</td>
<td>(R_c = S_c)</td>
<td>(R_a = S_a \times D_a)</td>
</tr>
<tr>
<td>out</td>
<td>No</td>
<td>(R_c = S_c)</td>
<td>(R_a = S_a \times (1 - D_a))</td>
</tr>
<tr>
<td>atop</td>
<td>N/A</td>
<td>(R_c = S_{ca} + D_c \times (1 - S_a))</td>
<td>(R_a = D_a)</td>
</tr>
<tr>
<td>dest_over</td>
<td>N/A</td>
<td>(R_c = \frac{(S_a \times (1 - D_a) + D_{ca})}{R_a})</td>
<td>(R_a = (1 - D_a) \times S_a + D_a)</td>
</tr>
<tr>
<td>dest_in</td>
<td>No</td>
<td>(R_c = D_c)</td>
<td>(R_a = S_a \times D_a)</td>
</tr>
<tr>
<td>dest_out</td>
<td>N/A</td>
<td>(R_c = D_c)</td>
<td>(R_a = (1 - S_a) \times D_a)</td>
</tr>
<tr>
<td>dest_atop</td>
<td>No</td>
<td>(R_c = S_c \times (1 - D_a) + D_{ca})</td>
<td>(R_a = S_a)</td>
</tr>
<tr>
<td>xor_op</td>
<td>N/A</td>
<td>(R_c = \frac{(S_{ca} \times (1 - D_a) + D_{ca} \times (1 - S_a))}{R_a})</td>
<td>(R_a = S_a + D_a - 2 \times S_a \times D_a)</td>
</tr>
<tr>
<td>add</td>
<td>N/A</td>
<td>(R_c = \frac{(S_{ca} + D_{ca})}{R_a})</td>
<td>(R_a = \min(1, S_a + D_a))</td>
</tr>
<tr>
<td>saturate</td>
<td>N/A</td>
<td>(R_c = \frac{\min(S_a, 1 - D_a) \times S_c + D_{ca}}{R_a})</td>
<td>(R_a = \min(1, S_a + D_a))</td>
</tr>
</tbody>
</table>

The blend enumerators and hsl enumerators share a common formula for the result color’s color channel, with only one part of it changing depending on the enumerator. The result color’s color channel value formula is as follows: \(R_c = \frac{1}{R_a} \times ((1 - D_a) \times S_{ca} + (1 - S_a) \times D_{ca} + S_a \times D_a \times f(S_c, D_c))\). The function \(f(S_c, D_c)\) is the component of the formula that is enumerator dependent.

For the blend enumerators, the color channels shall be treated as separable, meaning that the color formula shall be evaluated separately for each color channel: red, green, and blue.
The color formula divides 1 by the result color’s alpha channel value. As a result, if the result color’s alpha channel is zero then a division by zero would normally occur. Implementations shall not throw an exception nor otherwise produce any observable error condition if the result color’s alpha channel is zero. Instead, implementations shall bypass the division by zero and produce the result color (0, 0, 0, 0), i.e. transparent black, if the result color alpha channel formula evaluates to zero. [Note: The simplest way to comply with this requirement is to bypass evaluation of the color channel formula in the event that the result alpha is zero. However, in order to allow implementations the greatest latitude possible, only the result is specified. —end note]

For the enumerators in Table 27 and Table 28 the result color’s alpha channel value formula is as follows: \( Ra = Sa + Da \times (1 - Sa) \). [Note: Since it is the same formula for all enumerators in those tables, the formula is not included in those tables. —end note]

All of the blend enumerators and hsl enumerators have a bound value of 'N/A'.

Table 27 — compositing_op blend enumerator meanings

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiply</td>
<td>( f(Sc, Dc) = Sc \times Dc )</td>
</tr>
<tr>
<td>screen</td>
<td>( f(Sc, Dc) = Sc + Dc - Sc \times Dc )</td>
</tr>
</tbody>
</table>
| overlay          | \( \) if(\( Dc \leq 0.5 \)) \{ \)
|                  | \( f(Sc, Dc) = 2 \times Sc \times Dc \) \}      |
|                  | \{ else \{ \)
|                  | \( f(Sc, Dc) = 1 - 2 \times (1 - Sc) \times (1 - Dc) \) \} |
|                  | \}                                               |
|                  | [Note: The difference between this enumerator and hard_light is that this tests the destination color (Dc) whereas hard_light tests the source color (Sc). —end note] |
| darken           | \( f(Sc, Dc) = \min(Sc, Dc) \)                   |
| lighten          | \( f(Sc, Dc) = \max(Sc, Dc) \)                   |
| color_dodge      | \( \) if(\( Dc < 1 \)) \{ \)
|                  | \( f(Sc, Dc) = \min(1 - \frac{Dc}{1 - Sc}) \) \} |
|                  | \}                                               |
|                  | \{ else \{ \)
|                  | \( f(Sc, Dc) = 1 \) \}                           |
| color_burn       | \( \) if(\( Dc > 0 \)) \{ \)
|                  | \( f(Sc, Dc) = 1 - \min(1, \frac{1 - Dc}{Sc}) \) \} |
|                  | \}                                               |
|                  | \{ else \{ \)
|                  | \( f(Sc, Dc) = 0 \) \}                           |
|                  | \}                                               |
| hard_light       | \( \) if(\( Sc \leq 0.5f \)) \{ \)
|                  | \( f(Sc, Dc) = 2 \times Sc \times Dc \) \}       |
|                  | \}                                               |
|                  | \{ else \{ \)
|                  | \( f(Sc, Dc) = 1 - 2 \times (1 - Sc) \times (1 - Dc) \) \} |
|                  | \}                                               |
|                  | [Note: The difference between this enumerator and overlay is that this tests the source color (Sc) whereas overlay tests the destination color (Dc). —end note] |
Table 27 — compositing_op blend enumerator meanings (continued)

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Color</th>
</tr>
</thead>
</table>
| **soft_light**   | *if* $(Sc \leq 0.5)$  
                  $f(Sc, Dc) =$  
                  $Dc - (1 - 2 \times Sc) \times Dc \times$  
                  $(1 - Dc)$  
                  
                  *else*  
                  $f(Sc, Dc) =$  
                  $Dc + (2 \times Sc - 1) \times$  
                  $(g(Dc) - Sc)$  
                  |

$g(Dc)$ is defined as follows:

*if* $(Dc \leq 0.25)$  
$g(Dc) =$  
$((16 \times Dc - 12) \times Dc + 4) \times Dc$  

*else*  
$g(Dc) = \sqrt{Dc}$

**difference**  
$f(Sc, Dc) = \text{abs}(Dc - Sc)$

**exclusion**  
$f(Sc, Dc) = Sc + Dc - 2 \times Sc \times Dc$

For the hsl enumerators, the color channels shall be treated as nonseparable, meaning that the color formula shall be evaluated once, with the colors being passed in as tuples in the form (red, green, blue).

The following additional functions are used to define the hsl enumerator formulas:

$\text{min}(x, y, z) = \text{min}(x, \text{min}(y, z))$

$max(x, y, z) = \text{max}(x, \text{max}(y, z))$

$\text{sat}(C) = \text{max}(Cr, Cg, Cb) - \text{min}(Cr, Cg, Cb)$

$lum(C) = Cr \times 0.3 + Cg \times 0.59 + Cb \times 0.11$

$\text{clip}_{\_color}(C) =$  
$L = \text{lum}(C)$  
$N = \text{min}(Cr, Cg, Cb)$  
$X = \text{max}(Cr, Cg, Cb)$  
*if* $(N < 0.0)$  
$Cr = L + \left(\frac{(Cr - L) \times L}{(L - N)}\right)$  
$Cg = L + \left(\frac{(Cg - L) \times L}{(L - N)}\right)$  
$Cb = L + \left(\frac{(Cb - L) \times L}{(L - N)}\right)$  
*else*  
$Cr = L + \left(\frac{(Cr - L) \times (1 - L)}{(X - L)}\right)$  
$Cg = L + \left(\frac{(Cg - L) \times (1 - L)}{(X - L)}\right)$  
$Cb = L + \left(\frac{(Cb - L) \times (1 - L)}{(X - L)}\right)$


```c
return C
}

31 set_lum(C, L) = {
    D = L - lum(C)
    Cr = Cr + D
    Cg = Cg + D
   Cb = Cb + D
    return clip_color(C)
}

32 set_sat(C, S) = {
    R = C
    auto& max = (Rr > Rg) ? ((Rr > Rb) ? Rr : Rb) : ((Rg > Rb) ? Rg : Rb)
    auto& mid = (Rr > Rg) ? ((Rr > Rb) ? (Rg > Rb) ? Rr : Rg) : Rr
    if (max > min) {
        mid = \frac{(mid - min) \times S}{max - min}
        max = S
    }
    else {
        mid = 0.0
        max = 0.0
    }
    min = 0.0
    return R
}
```

[Note: In the formula, max, mid, and min are reference variables which are bound to the highest value, second highest value, and lowest value color channels of the (red, blue, green) tuple R such that the subsequent operations modify the values of R directly. — end note]

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Color &amp; Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsl_hue</td>
<td>f(Sc, Dc) = set_lum(set_sat(Sc, sat(Dc)), lum(Dc))</td>
</tr>
<tr>
<td>hsl_saturation</td>
<td>f(Sc, Dc) = set_lum(set_sat(Dc, sat(Sc)), lum(Dc))</td>
</tr>
<tr>
<td>hsl_color</td>
<td>f(Sc, Dc) = set_lum(Sc, lum(Dc))</td>
</tr>
<tr>
<td>hsl_luminosity</td>
<td>f(Sc, Dc) = set_lum(Dc, lum(Sc))</td>
</tr>
</tbody>
</table>

14.7 Enum class format

14.7.1 Summary
The format enum class indicates a visual data format. See Table 29 for the meaning of each format enumerator.

14.7.2 Synopsis

```c
namespace std::experimental::io2d::v1 {
    enum class format {
        invalid,
        argb32,
        xrgb32,
        xrgb16,
        a8
    };
}
```

14.7.3 Enumerators
Table 29 — format enumerator meanings

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>invalid</td>
<td>A previously requested format is unsupported by the implementation.</td>
</tr>
<tr>
<td>argb32</td>
<td>A 32-bit RGB color model pixel format. There is an 8 bit alpha channel, an 8-bit red color channel, an 8-bit green color channel, and an 8-bit blue color channel. The byte order, interpretation of values within each channel, and whether or not this is a premultiplied format are implementation-defined.</td>
</tr>
<tr>
<td>xrgb32</td>
<td>A 32-bit RGB color model pixel format. There is an 8 bit channel that is not used, an 8-bit red color channel, an 8-bit green color channel, and an 8-bit blue color channel. The byte order and interpretation of values within each channel are implementation-defined.</td>
</tr>
<tr>
<td>xrgb16</td>
<td>A 16-bit RGB color model pixel format. There is a red color channel, a green color channel, and a blue color channel. The number of bits, byte order, and interpretation of values within each channel are implementation-defined.</td>
</tr>
<tr>
<td>a8</td>
<td>An 8-bit transparency data pixel format. All 8 bits are an alpha channel.</td>
</tr>
</tbody>
</table>

1 Implementations may support additional visual data formats (Sec: 8.2.3).

14.8 Enum class scaling

14.8.1 scaling summary

The scaling enum class specifies the type of scaling a display_surface will use when the size of its display buffer differs from the size of its back buffer.

2 See Table 30 for the meaning of each scaling enumerator.

14.8.2 scaling synopsis

namespace std::experimental::io2d::v1 {
    enum class scaling {
        letterbox,
        uniform,
        fill_uniform,
        fill_exact,
        none
    };
}

14.8.3 scaling enumerators

[Note: In the following table, examples will be given to help explain the meaning of each enumerator. The examples will all use a display_surface called ds.]

The back buffer of ds is 640x480 (i.e. it has a width of 640 pixels and a height of 480 pixels), giving it an aspect ratio of 1.3.

The display buffer of ds is 1280x720, giving it an aspect ratio of 1.7.

When a rectangle is defined in an example, the coordinate \((x_1, y_1)\) denotes the top left corner of the rectangle, inclusive, and the coordinate \((x_2, y_2)\) denotes the bottom right corner of the rectangle, exclusive. As such, a rectangle with \((x_1, y_1) = (10, 10), (x_2, y_2) = (20, 20)\) is 10 pixels wide and 10 pixels tall and includes the pixel \((x, y) = (19, 19)\) but does not include the pixels \((x, y) = (20, 19)\) or \((x, y) = (19, 20)\). — end note]
Table 30 — scaling enumerator meanings

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| letterbox  | Fill the display buffer with the letterbox brush (??) of the display_surface. Uniformly scale the back buffer so that one dimension of it is the same length as the same dimension of the display buffer and the second dimension of it is not longer than the second dimension of the display buffer and transfer the scaled back buffer to the display buffer using sampling such that it is centered in the display buffer.  

[Example: The display buffer of ds will be filled with the brush object returned by ds.letterbox_brush();. The back buffer of ds will be scaled so that it is 960x720, thereby retaining its original aspect ratio. The scaled back buffer will be transferred to the display buffer using sampling such that it is in the rectangle $(x1, y1) = (1280/2 - 960/2, 0) = (160, 0)$,  

$(x2, y2) = (960 + (1280/2 - 960/2) / 2, 0) = (1120, 720)$. This fulfills all of the conditions. At least one dimension of the scaled back buffer is the same length as the same dimension of the display buffer (both have a height of 720 pixels). The second dimension of the scaled back buffer is not longer than the second dimension of the display buffer (the back buffer’s scaled width is 960 pixels, which is not longer than the display buffer’s width of 1280 pixels. Lastly, the scaled back buffer is centered in the display buffer (on the x axis there are 160 pixels between each vertical side of the scaled back buffer and the nearest vertical edge of the display buffer and on the y axis there are 0 pixels between each horizontal side of the scaled back buffer and the nearest horizontal edge of the display buffer). — end example]
Table 30 — scaling enumerator meanings (continued)

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| **uniform** | Uniformly scale the back buffer so that one dimension of it is the same length as the same dimension of the display buffer and the second dimension of it is not longer than the second dimension of the display buffer and transfer the scaled back buffer to the display buffer using sampling such that it is centered in the display buffer.  
   [Example: The back buffer of ds will be scaled so that it is 960x720, thereby retaining its original aspect ratio. The scaled back buffer will be transferred to the display buffer using sampling such that it is in the rectangle 
   $(x_1, y_1) = \left( \frac{1280}{2} - \frac{960}{2}, 0 \right) = (160, 0)$, 
   $(x_2, y_2) = (960 + \frac{1280}{2} - \frac{960}{2}, 720) = (1120, 720)$. This fulfills all of the conditions. At least one dimension of the scaled back buffer is the same length as the same dimension of the display buffer (both have a height of 720 pixels). The second dimension of the scaled back buffer is not longer than the second dimension of the display buffer (the back buffer’s scaled width is 960 pixels, which is not longer than the display buffer’s width of 1280 pixels. Lastly, the scaled back buffer is centered in the display buffer (on the $x$ axis there are 160 pixels between each vertical side of the scaled back buffer and the nearest vertical edge of the display buffer and on the $y$ axis there are 0 pixels between each horizontal side of the scaled back buffer and the nearest horizontal edge of the display buffer).]  
   [Note: The difference between uniform and letterbox is that uniform does not modify the contents of the display buffer that fall outside of the rectangle into which the scaled back buffer is drawn while letterbox fills those areas with the display_surface object’s letterbox brush (see: ??).] — end example] |
| **fill_uniform** | Uniformly scale the back buffer so that one dimension of it is the same length as the same dimension of the display buffer and the second dimension of it is not shorter than the second dimension of the display buffer and transfer the scaled back buffer to the display buffer using sampling such that it is centered in the display buffer.  
   [Example: The back buffer of ds will be drawn in the rectangle $(x_1, y_1) = (0, -120)$, $(x_2, y_2) = (1280, 840)$. This fulfills all of the conditions. At least one dimension of the scaled back buffer is the same length as the same dimension of the display buffer (both have a width of 1280 pixels). The second dimension of the scaled back buffer is not shorter than the second dimension of the display buffer (the back buffer’s scaled height is 840 pixels, which is not shorter than the display buffer’s height of 720 pixels). Lastly, the scaled back buffer is centered in the display buffer (on the $x$ axis there are 0 pixels between each vertical side of the rectangle and the nearest vertical edge of the display buffer and on the $y$ axis there are 120 pixels between each horizontal side of the rectangle and the nearest horizontal edge of the display buffer).] — end example] |
Table 30 — scaling enumerator meanings (continued)

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill_exact</td>
<td>Scale the back buffer so that each dimension of it is the same length as the same dimension of the display buffer and transfer the</td>
</tr>
<tr>
<td></td>
<td>scaled back buffer to the display buffer using sampling such that its origin is at the origin of the display buffer.</td>
</tr>
<tr>
<td></td>
<td>[Example: The back buffer will be drawn in the rectangle $(x_1, y_1) = (0, 0), (x_2, y_2) = (1280, 720)$. This fulfills all of</td>
</tr>
<tr>
<td></td>
<td>the conditions. Each dimension of the scaled back buffer is the same length as the same dimension of the display buffer (both have</td>
</tr>
<tr>
<td></td>
<td>a width of 1280 pixels and a height of 720 pixels) and the origin of the scaled back buffer is at the origin of the display buffer.</td>
</tr>
<tr>
<td></td>
<td>— end example]</td>
</tr>
<tr>
<td>none</td>
<td>Do not perform any scaling. Transfer the back buffer to the display buffer using sampling such that its origin is at the origin of</td>
</tr>
<tr>
<td></td>
<td>the display buffer.</td>
</tr>
<tr>
<td></td>
<td>[Example: The back buffer of ds will be drawn in the rectangle $(x_1, y_1) = (0, 0), (x_2, y_2) = (640, 480)$ such that no scaling</td>
</tr>
<tr>
<td></td>
<td>occurs and the origin of the back buffer is at the origin of the display buffer. — end example]</td>
</tr>
</tbody>
</table>

14.9 Class template basic_render_props

14.9.1 Overview

The basic_render_props class template provides general state information that is applicable to all rendering and composing operations (15.3.2).

2 It has a filter algorithm of type filter, a surface matrix of type basic_matrix_2d, and a compositing operator of type compositing_op.

3 The data are stored in an object of type typename GraphicsSurfaces::surface_state_props::render_props_data_type. It is accessible using the data member functions.

14.9.2 Synopsis

namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    class basic_render_props {
        public:
            using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
            using data_type = typename GraphicsSurfaces::surface_state_props::render_props_data_type;

            // 14.9.3, constructors:
            basic_render_props() noexcept;
            explicit basic_render_props(filter f,
                        const basic_matrix_2d<graphics_math_type>& m = basic_matrix_2d<graphics_math_type>{},
                        compositing_op co = compositing_op::over) noexcept;

            // 14.9.4, accessors:
            const data_type& data() const noexcept;
            data_type& data() noexcept;

            // 14.9.5, modifiers:
            void filtering(filter f) noexcept;
            void compositing(compositing_op co) noexcept;
            void surface_matrix(const basic_matrix_2d<graphics_math_type>& m) noexcept;

            // 14.9.6, observers:
            filter filtering() const noexcept;
            compositing_op compositing() const noexcept;

§ 14.9.2 143
basic_matrix_2d<graphics_math_type> surface_matrix() const noexcept;
};

14.9.3 Constructors

basic_render_props() noexcept;

Effects: Constructs an object of type basic_render_props.

Postconditions: data() == X::surface_state_props::create_render_props().

explicit basic_render_props(filter f,
const basic_matrix_2d<graphics_math_type>& m = basic_matrix_2d<graphics_math_type>{},
compositing_op co = compositing_op::over) noexcept;

Requires: m.is_invertible() == true.

Effects: Constructs an object of type basic_render_props.

Postconditions: data() == GraphicsSurfaces::surface_state_props::create_render_props(f,
m, co).

14.9.4 Accessors

const data_type& data() const noexcept;

data_type& data() noexcept;

Returns: A reference to the basic_render_props object’s data object (See: 14.9.1).

14.9.5 Modifiers

void filtering(filter f) noexcept;

Effects: Calls GraphicsSurfaces::surface_state_props::filtering(data(), f)).

Remarks: The filtering algorithm is f.

void compositing(compositing_op co) noexcept;

Effects: Calls GraphicsSurfaces::surface_state_props::compositing(data(), co).

Remarks: The compositing operator is co.

void surface_matrix(const basic_matrix_2d<graphics_math_type>& m) noexcept;

Requires: m.is_invertible() == true.

Effects: Calls GraphicsSurfaces::surface_state_props::surface_matrix(data(), m).

Remarks: The surface matrix is m.

14.9.6 Observers

filter filtering() const noexcept;

Returns: GraphicsSurfaces::surface_state_props::filtering(data()).

Remarks: The returned value is the filter algorithm.

compositing_op compositing() const noexcept;

Returns: GraphicsSurfaces::surface_state_props::compositing(data()).

Remarks: The returned value is the compositing operator.

basic_matrix_2d<graphics_math_type> surface_matrix() const noexcept;

Returns: GraphicsSurfaces::surface_state_props::surface_matrix(data()).

Remarks: The returned value is the surface matrix.
14.10 Class basic_brush_props

14.10.1 basic_brush_props summary

The basic_brush_props class provides general state information that is applicable to all rendering and composing operations (15.3.2).

It has a wrap mode of type wrap_mode, a filter of type filter, a fill rule of type fill_rule, and a brush matrix of type basic_matrix_2d.

14.10.2 basic_brush_props synopsis

```cpp
namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    class basic_brush_props {
    public:
        using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;

        // 14.10.3, constructor:
        basic_brush_props(io2d::wrap_mode w = io2d::wrap_mode::none,
                          io2d::filter fi = io2d::filter::good,
                          io2d::fill_rule fr = io2d::fill_rule::winding,
                          const basic_matrix_2d<graphics_math_type>& m = basic_matrix_2d<graphics_math_type>{})
            noexcept;

        // 14.10.4, modifiers:
        void wrap_mode(io2d::wrap_mode w) noexcept;
        void filter(io2d::filter fi) noexcept;
        void fill_rule(io2d::fill_rule fr) noexcept;
        void brush_matrix(const basic_matrix_2d<graphics_math_type>& m) noexcept;

        // 14.10.5, observers:
        io2d::wrap_mode wrap_mode() const noexcept;
        io2d::filter filter() const noexcept;
        io2d::fill_rule fill_rule() const noexcept;
        basic_matrix_2d<graphics_math_type> brush_matrix() const noexcept;
    };
}
```

14.10.3 basic_brush_props constructor

```cpp
basic_brush_props(io2d::wrap_mode w = io2d::wrap_mode::none,
                   io2d::filter fi = io2d::filter::good,
                   io2d::fill_rule fr = io2d::fill_rule::winding,
                   const basic_matrix_2d<graphics_math_type>& m = basic_matrix_2d<graphics_math_type>{})
        noexcept;
```

1. Requires: m.is_invertible() == true.
2. Effects: Constructs an object of type basic_brush_props.
3. The wrap mode is w. The filter is fi. The fill rule is fr. The brush matrix is m.

14.10.4 basic_brush_props modifiers

```cpp
void wrap_mode(io2d::wrap_mode w) noexcept;

void filter(io2d::filter fi) noexcept;

void fill_rule(io2d::fill_rule fr) noexcept;

void brush_matrix(const basic_matrix_2d<graphics_math_type>& m) noexcept;
```

1. Effects: The wrap mode is w.
2. Effects: The filter is fi.
3. Effects: The fill rule is fr.
4. Requires: m.is_invertible() == true.
5. Effects: The brush matrix is m.
14.10.5 basic_brush_props observers

io2d::wrap_mode wrap_mode() const noexcept;

Returns: The wrap mode.

io2d::filter filter() const noexcept;

Returns: The filter.

io2d::fill_rule fill_rule() const noexcept;

Returns: The fill rule.

basic_matrix_2d<graphics_math_type> brush_matrix() const noexcept;

Returns: The brush matrix.

14.11 Class template basic_clip_props

14.11.1 Overview

The basic_clip_props class template provides general state information that is applicable to all rendering and compositing operations (15.3.2). It has a clip area of type optional<interpreted_path> and a fill rule of type fill_rule. If the clip area has no value, the clip area is boundless.

The data are stored in an object of type typename GraphicsSurfaces::surface_state_props::clip_props_data_type. It is accessible using the data member functions.

14.11.2 basic_clip_props synopsis

```cpp
namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    class basic_clip_props {
    public:
        using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
        using data_type = typename GraphicsSurfaces::surface_state_props::clip_props_data_type;

        // 14.11.3, constructors:
        basic_clip_props() noexcept;
        template <class Allocator>
        explicit basic_clip_props(
            const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
            io2d::fill_rule fr = io2d::fill_rule::winding);
        explicit basic_clip_props(
            const basic_interpreted_path<GraphicsSurfaces>& ip,
            io2d::fill_rule fr = io2d::fill_rule::winding) noexcept;
        explicit basic_clip_props(const basic_bounding_box<graphics_math_type>& r,
            io2d::fill_rule fr = io2d::fill_rule::winding);

        // 14.11.4, accessors:
        const data_type& data() const noexcept;
        data_type& data() noexcept;

        // 14.11.5, modifiers:
        void clip();
        template <class Allocator>
        void clip(const basic_path_builder<GraphicsSurfaces, Allocator>& pb);
        void clip(const basic_interpreted_path<GraphicsSurfaces>& ip) noexcept;
        void fill_rule(io2d::fill_rule fr) noexcept;

        // 14.11.6, observers:
        optional<basic_interpreted_path<GraphicsSurfaces>> clip() const noexcept;
        io2d::fill_rule fill_rule() const noexcept;
    };
}
```
14.11.3 basic_clip_props constructors

basic_clip_props() noexcept;
   
   Effects: Constructs an object of type basic_clip_props.

Postconditions: data() == GraphicsSurfaces::surface_state_props::create_clip_props().

Remarks: The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(). The fill rule
   is io2d::fill_rule::winding.

template <class Allocator>
explicit basic_clip_props(const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
   io2d::fill_rule fr = io2d::fill_rule::winding);
   
   Effects: Constructs an object of type basic_clip_props.

Postconditions: data() == GraphicsSurfaces::surface_state_props::create_clip_props(pb, fr).

Remarks: The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(basic_interpreted_path<GraphicsSurfaces>(pb)).
   The fill rule is fr.

template <class InputIterator>
basic_clip_props(InputIterator first, InputIterator last,
   io2d::fill_rule fr = io2d::fill_rule::winding);
   
   Effects: Constructs an object of type basic_clip_props.

Postconditions: data() == GraphicsSurfaces::surface_state_props::create_clip_props(first,
   last, fr).

Remarks: The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(basic_interpreted_path<GraphicsSurfaces>(first, last)).
   The fill rule is fr.

template <class Allocator>
explicit basic_clip_props(
   initializer_list<basic_figure_items<GraphicsSurfaces>::figure_item> il,
   io2d::fill_rule fr = io2d::fill_rule::winding);
   
   Effects: Constructs an object of type basic_clip_props.

Postconditions: data() == GraphicsSurfaces::surface_state_props::create_clip_props(il, fr).

Remarks: The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(basic_interpreted_path<GraphicsSurfaces>(il)).
   The fill rule is fr.

explicit basic_clip_props(const basic_interpreted_path<GraphicsSurfaces>& ip,
   io2d::fill_rule fr = io2d::fill_rule::winding) noexcept;
   
   Effects: Constructs an object of type basic_clip_props.

Postconditions: data() == GraphicsSurfaces::surface_state_props::create_clip_props(ip, fr).

Remarks: The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(ip).
   The fill rule is fr.

explicit basic_clip_props(const basic_bounding_box<graphics_math_type>& r,
   io2d::fill_rule fr = io2d::fill_rule::winding);
   
   Effects: Constructs an object of type basic_clip_props.

Postconditions: data() == GraphicsSurfaces::surface_state_props::create_clip_props(r, fr).

Remarks: The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(r));
   The fill rule is fr.
14.11.4 Accessors

const data_type& data() const noexcept;
data_type& data() noexcept;

Returns: A reference to the basic_clip_props object’s data object (See: 14.11.1).

14.11.5 basic_clip_props modifiers

template <class Allocator>
void clip();

Effects: The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>().

void clip(const basic_bounding_box<GraphicsSurfaces>& bb);

Effects: Calls GraphicsSurfaces::surface_state_props::clip(data(), bb);
Remarks: The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(bb).

template <class Allocator>
void clip(const basic_path_builder<GraphicsSurfaces, Allocator>& pb);

Effects: Calls GraphicsSurfaces::surface_state_props::clip(data(), pb);
Remarks: The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(pb).

template <class InputIterator>
void clip(InputIterator first, InputIterator last);

Effects: Calls GraphicsSurfaces::surface_state_props::clip(data(), first, last);
Remarks: The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(first, last).

void clip(
    const initializer_list<typename basic_figure_items<GraphicsSurfaces>::figure_item> il);

Effects: Calls GraphicsSurfaces::surface_state_props::clip(data(), il);
Remarks: The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(il).

void clip(const basic_bounding_box<GraphicsSurfaces>& bb);

Effects: Calls GraphicsSurfaces::surface_state_props::clip(data(), ip);
Remarks: The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(ip).

void fill_rule(experimental::io2d::fill_rule fr) noexcept;

Effects: Calls GraphicsSurfaces::surface_state_props::clip_fill_rule(fr).
Remarks: The fill rule is fr.

14.11.6 basic_clip_props observers

optional<basic_interpreted_path<GraphicsSurfaces>> clip() const noexcept;

Returns: GraphicsSurfaces::surface_state_props::clip(data()).
Remarks: The return value is the clip area.

io2d::fill_rule fill_rule() const noexcept;

Returns: GraphicsSurfaces::surface_state_props::clip_fill_rule(data()).
Remarks: The return value is the fill rule.
14.12 Class basic_stroke_props

14.12.1 basic_stroke_props summary

The basic_stroke_props class provides state information that is applicable to the stroking operation (see: 15.3.2 and 15.3.6).

It has a line width of type float, a line cap of type line_cap, a line join of type line_join, and a miter limit of type float.

14.12.2 basic_stroke_props synopsis

namespace std::experimental::io2d::v1 {
  template <class GraphicsSurfaces>
  class basic_stroke_props {
    public:
      using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;

      // 14.12.3, constructors:
      basic_stroke_props() noexcept;
      explicit basic_stroke_props(float w, io2d::line_cap lc = io2d::line_cap::none,
                                  io2d::line_join lj = io2d::line_join::miter, float ml = 10.0f) noexcept;

      // 14.12.4, modifiers:
      void line_width(float w) noexcept;
      void line_cap(io2d::line_cap lc) noexcept;
      void line_join(io2d::line_join lj) noexcept;
      void miter_limit(float ml) noexcept;

      // 14.12.5, observers:
      float line_width() const noexcept;
      io2d::line_cap line_cap() const noexcept;
      io2d::line_join line_join() const noexcept;
      float miter_limit() const noexcept;
    }
  }

14.12.3 basic_stroke_props constructors

basic_stroke_props() noexcept;

Effects: Equivalent to: basic_stroke_props(2.0f).

explicit basic_stroke_props(float w, io2d::line_cap lc = io2d::line_cap::none,
                            io2d::line_join lj = io2d::line_join::miter,
                            float ml = 10.0f) noexcept;

Requires: w > 0.0f. ml >= 10.0f. ml <= max_miter_limit().

Effects: The line width is w. The line cap is lc. The line join is lj. The miter limit is ml.

14.12.4 basic_stroke_props modifiers

void line_width(float w) noexcept;

Requires: w >= 0.0f.

Effects: The line width is w.

void line_cap(io2d::line_cap lc) noexcept;

Effects: The line cap is lc.

void line_join(io2d::line_join lj) noexcept;

Effects: The line join is lj.
void miter_limit(float ml) noexcept;

Requires: ml >= 1.0f and ml <= max_miter_limit.

Effects: The miter limit is ml.

14.12.5 basic_stroke_props observers

float line_width() const noexcept;

Returns: The line width.

io2d::line_cap line_cap() const noexcept;

Returns: The line cap.

io2d::line_join line_join() const noexcept;

Returns: The line join.

float miter_limit() const noexcept;

Returns: The miter limit.

float max_miter_limit() const noexcept;

Requires: This value shall be finite and greater than 10.0f.

Returns: The implementation-defined maximum value of miter limit.

14.13 Class template basic_fill_props

14.13.1 Overview

The basic_fill_props class template provides state information that is applicable to the filling rendering and composing operation (15.3.2).

It has a fill rule of type fill_rule and an antialiasing algorithm of type antialias.

The data are stored in an object of type typename GraphicsSurfaces::surface_state_props::fill_props_data_type. It is accessible using the data member functions.

14.13.2 Synopsis

namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    class basic_fill_props {
    public:
        using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
        using data_type = typename GraphicsSurfaces::surface_state_props::fill_props_data_type;

        // 14.13.3, constructors:
        basic_fill_props() noexcept;
        explicit basic_fill_props(io2d::fill_rule fr,
                                  antialias aa = antialias::good) noexcept;

        // 14.13.4, accessors:
        const data_type& data() const noexcept;
        data_type& data() noexcept;

        // 14.13.5, modifiers:
        void fill_rule(io2d::fill_rule fr) noexcept;
        void antialiasing(antialias aa) noexcept;

        // 14.13.6, observers:
        io2d::fill_rule fill_rule() const noexcept;
        antialias antialiasing() const noexcept;
    };
}
14.13.3 Constructors

basic_fill_props() noexcept;
1
   Effects: Constructs an object of type basic_fill_props.
2
   Postconditions: data() == X::surface_state_props::create_fill_props().

explicit basic_fill_props(io2d::fill_rule fr, antialias aa = antialias::good) noexcept;
3
   Effects: Constructs an object of type basic_fill_props.
4
   Postconditions: data() == GraphicsSurfaces::surface_state_props::create_fill_props(fr, aa).

14.13.4 Accessors

const data_type& data() const noexcept;
1
   Returns: A reference to the basic_fill_props object’s data object (See: 14.13.1).

data_type& data() noexcept;
2

14.13.5 Modifiers

void fill_rule(io2d::fill_rule fr) noexcept;
1
   Effects: Calls GraphicsSurfaces::surface_state_props::fill_fill_rule(data(), fr)).
   Remarks: The fill rull is fr.

void compositing(compositing_op co) noexcept;
2
   Effects: Calls GraphicsSurfaces::surface_state_props::antialiasing(data(), aa).
   Remarks: The antialiasing algorithm is aa.

14.13.6 Observers

io2d::fill_rule fill_rule() const noexcept;
1
   Returns: GraphicsSurfaces::surface_state_props::fill_fill_rule(data()).
   Remarks: The returned value is the fill rule.

antialias antialiasing() const noexcept;
3
   Returns: GraphicsSurfaces::surface_state_props::antialiasing(data()).
   Remarks: The returned value is the antialiasing algorithm.

14.14 Class basic_mask_props

14.14.1 basic_mask_props summary

The basic_mask_props class provides state information that is applicable to the mask rendering and compositing operation (15.3.2).

14.14.2 basic_mask_props synopsis

namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    class basic_mask_props {
        public:
            using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;

            // 14.14.3, constructor:
            basic_mask_props(io2d::wrap_mode w = io2d::wrap_mode::repeat,
                             io2d::filter fi = io2d::filter::good,
                             const basic_matrix_2d<graphics_math_type>& m = basic_matrix_2d<graphics_math_type>())
            noexcept;

§ 14.14.2
// 14.14.4, modifiers:
void wrap_mode(io2d::wrap_mode w) noexcept;
void filter(io2d::filter fi) noexcept;
void mask_matrix(const basic_matrix_2d<graphics_math_type>& m) noexcept;

// 14.14.5, observers:
io2d::wrap_mode wrap_mode() const noexcept;
io2d::filter filter() const noexcept;
basic_matrix_2d<graphics_math_type> mask_matrix() const noexcept;
}

14.14.3 basic_mask_props constructor
basic_mask_props(io2d::wrap_mode w = io2d::wrap_mode::repeat,
io2d::filter fi = io2d::filter::good,
const basic_matrix_2d<graphics_math_type>& m = basic_matrix_2d<graphics_math_type>{}) noexcept;

Requires: m.is_invertible() == true.

Effects: The wrap mode is w. The filter is fi. The mask matrix is m.

14.14.4 basic_mask_props modifiers
void wrap_mode(io2d::wrap_mode w) noexcept;

Effects: The wrap mode is w.

void filter(io2d::filter fi) noexcept;

Effects: The filter is fi.

void mask_matrix(const basic_matrix_2d<graphics_math_type>& m) noexcept;

Requires: m.is_invertible() == true.

Effects: The mask matrix is m.

14.14.5 basic_mask_props observers
io2d::wrap_mode wrap_mode() const noexcept;

Returns: The wrap mode.

io2d::filter filter() const noexcept;

Returns: The filter.

basic_matrix_2d<graphics_math_type> mask_matrix() const noexcept;

Returns: The mask matrix.

14.15 Class template basic_dashes

14.15.1 basic_dashes class template

The class template basic_dashes describes a pattern for determining, in conjunction with other properties,
what points on a path are included when a stroking operation is performed.

It has an offset of type float and a pattern of an unspecified type capable of sequentially storing floating-
point values.

The data are stored in an object of type typename GraphicsSurfaces::surface_props_data::dashes-
data_type. It is accessible using the data member function.

14.15.2 Synopsis

namespace std::experimental::io2d::v1 {
template <class GraphicsSurfaces>
class basic_dashes {
public:
using data_type =
typename GraphicsSurfaces::surface_state_props::dashes_data_type;
public:
    // 14.15.3, constructors:
    basic_dashes() noexcept;
    template <class InputIterator>
    basic_dashes(float o, InputIterator first, InputIterator last);
    basic_dashes(float o, initializer_list<float> il);
    // 14.15.4, observers:
    const data_type& data() const noexcept;
};
    // 14.15.5, operators:
    template <class GraphicsSurfaces>
    bool operator==(const basic_dashes<GraphicsSurfaces>& lhs, const basic_dashes<GraphicsSurfaces>& rhs) noexcept;
    template <class GraphicsSurfaces>
    bool operator!=(const basic_dashes<GraphicsSurfaces>& lhs, const basic_dashes<GraphicsSurfaces>& rhs) noexcept;

14.15.3 Constructors

basic_dashes() noexcept;

Effects: Constructs an object of type basic_dashes.

Postconditions: data() == GraphicsSurfaces::surface_state_props::create_dashes().

Remarks: The offset is 0.0f and the pattern contains no values.

template <class InputIterator>
basic_dashes(float o, InputIterator first, InputIterator last);

Requires: The value type of InputIterator is float.

Each value from first through last - 1 is greater than or equal to 0.0f.

Effects: Constructs an object of type basic_dashes.

Postconditions: data() == GraphicsSurfaces::surface_state_props::create_dashes(o, first, last).

Remarks: The offset is o and the pattern is the sequential list of value beginning at first and ending at last - 1.

basic_dashes(float o, initializer_list<float> il);

Requires: Each value in il is greater than or equal to 0.0f.

Effects: Constructs an object of type basic_dashes.

Postconditions: data() == GraphicsSurfaces::surface_state_props::create_dashes(o, il).

14.15.4 Observers

const data_type& data() const noexcept;

Returns: A reference to the basic_dashes object’s data object (See 14.15.1).

14.15.5 Operators

template <class GraphicsSurfaces>
bool operator==(const basic_dashes<GraphicsSurfaces>& lhs, const basic_dashes<GraphicsSurfaces>& rhs) noexcept;

Returns: GraphicsSurfaces::surface_state_props::equal(lhs.data(), rhs.data()).
template <class GraphicsSurfaces>
bool operator!=(const basic_dashes<GraphicsSurfaces>& lhs,
               const basic_dashes<GraphicsSurfaces>& rhs) noexcept;

Returns: GraphicsSurfaces::surface_state_props::not_equal(lhs.data(), rhs.data()).
15 Surfaces

15.1 Enum class refresh_style

15.1.1 refresh_style summary

The refresh_style enum class describes when the draw callback (Table ??) of a display_surface object shall be called. See Table 31 for the meaning of each refresh_style enumerator.

15.1.2 refresh_style synopsis

namespace std::experimental::io2d::v1 {
enum class refresh_style {
    as_needed,
    as_fast_as_possible,
    fixed
};
}

15.1.3 refresh_style enumerators

Table 31 — refresh_style value meanings

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>as_needed</td>
<td>The draw callback shall be called when the implementation needs to do so. [Note: The intention of this enumerator is that implementations will call the draw callback as little as possible in order to minimize power usage. Users can call display_surface::redraw_required to make the implementation run the draw callback whenever the user requires. — end note]</td>
</tr>
<tr>
<td>as_fast_as_possible</td>
<td>The draw callback shall be called as frequently as possible, subject to any limits of the execution environment.</td>
</tr>
</tbody>
</table>
Table 31 — refresh_style value meanings (continued)

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed</td>
<td>The draw callback shall be called as frequently as needed to maintain the desired frame rate (Table ??) as closely as possible. If more time has passed between two successive calls to the draw callback than is required, it shall be called excess time and it shall count towards the required time, which is the time that is required to pass after a call to the draw callback before the next successive call to the draw callback shall be made. If the excess time is greater than the required time, implementations shall call the draw callback and then repeatedly subtract the required time from the excess time until the excess time is less than the required time. If the implementation needs to call the draw callback for some other reason, it shall use that call as the new starting point for maintaining the desired frame rate. [Example: Given a desired frame rate of 20.0f, then as per the above, the implementation would call the draw callback at 50 millisecond intervals or as close thereto as possible. If for some reason the excess time is 51 milliseconds, the implementation would call the draw callback, subtract 50 milliseconds from the excess time, and then would wait 49 milliseconds before calling the draw callback again. If only 15 milliseconds have passed since the draw callback was last called and the implementation needs to call the draw callback again, then the implementation shall call the draw callback immediately and proceed to wait 50 milliseconds before calling the draw callback again. — end example]</td>
</tr>
</tbody>
</table>

15.2 Enum class image_file_format

15.2.1 image_file_format summary

The image_file_format enum class specifies the data format that an image_surface object is constructed from or saved to. This allows data in a format that is required to be supported to be read or written regardless of its extension.

2 It also has a value that allows implementations to support additional file formats if it recognizes them.

15.2.2 image_file_format synopsis

namespace std::experimental::io2d::v1 {
enum class image_file_format {
    unknown,    
    png,        
    jpeg,       
    tiff
};
}

15.2.3 image_file_format enumerators

Table 32 — image_fileformat enumerator meanings

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknown</td>
<td>The format is unknown because it is not an image file format that is required to be supported. It may be known and supported by the implementation.</td>
</tr>
<tr>
<td>png</td>
<td>The PNG format.</td>
</tr>
</tbody>
</table>
Table 32 — `imagefileformat` enumerator meanings (continued)

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>jpeg</td>
<td>The JPEG format.</td>
</tr>
<tr>
<td>tiff</td>
<td>The TIFF format.</td>
</tr>
</tbody>
</table>

15.3 Overview of surface classes

15.3.1 Surface class templates description

There are three surface class templates:

1. basic_image_surface
2. basic_output_surface
3. basic_unmanaged_output_surface

For ease of description, an instantiation of a surface class template will be called a surface.

A surface contains visual data and provides an interface for managing and manipulating that visual data.

Surface class templates are `MoveConstructible` and `MoveAssignable`. They are neither `CopyConstructible` nor `CopyAssignable`. [Note: On many platforms, especially those that use specialized hardware to accelerate various graphics operations, copying a surface is highly detrimental to performance and is rarely desired. The `copy_surface` function (17.2) exists for those situations where a copy is desired. — end note]

The surface class templates manipulate visual data through rendering and composing operations.

The rendering and composing operations 15.3.2 are described in terms of operating on each integral point of the visual data of a surface. The reason for that is to support the discrete nature of raster graphics data. Operating on each integral point of the surface is the coarsest granularity allowed. Implementations may perform rendering and composing operations at a finer granularity than that of each integral point. [Note: Vector graphics data, being continuous, has the finest granularity possible since it resolves at the limits imposed by the precision of the floating-point types used to determine its visual data at any particular point. — end note]

15.3.2 Rendering and composing

15.3.2.1 Operations

The surface classes provide four fundamental rendering and composing operations:

Table 33 — surface rendering and composing operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Function(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Painting</td>
<td>paint</td>
</tr>
<tr>
<td>Filling</td>
<td>fill</td>
</tr>
<tr>
<td>Stroking</td>
<td>stroke</td>
</tr>
<tr>
<td>Masking</td>
<td>mask</td>
</tr>
</tbody>
</table>

All composing operations shall happen as-if in a linear color space, regardless of the color space of the visual data that is involved.

[Note: While a color space such as sRGB helps produce expected, consistent results when visual data are viewed by people, composing operations only produce expected results when the valid values for the color channel and alpha channel data in the visual data involved are uniformly (i.e. linearly) spaced. — end note]

15.3.2.2 Rendering and composing brushes

All rendering and composing operations use a `source brush` of type `basic_brush`.

The masking operation uses a `mask brush` of type `basic_brush`.

15.3.2.3 Rendering and composing source path

In addition to brushes (15.3.2.2), all rendering and composing operation except for painting and masking use a `source path`. The source path is either a `basic_path_builder<Allocator>` object or a `basic_interpreted_
path object. If it is a `basic_path_builder<Allocator>` object, it is interpreted (12.3.16) before it is used as the source path.

### 15.3.2.4 Common state data

All rendering and composing operations use the following state data:

Table 34 — surface rendering and composing common state data

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brush properties</td>
<td><code>brush_props</code></td>
</tr>
<tr>
<td>Surface properties</td>
<td><code>render_props</code></td>
</tr>
<tr>
<td>Clip properties</td>
<td><code>clip_props</code></td>
</tr>
</tbody>
</table>

### 15.3.2.5 Specific state data

In addition to the common state data (15.3.2.4), certain rendering and composing operations use state data that is specific to each of them:

Table 35 — surface rendering and composing specific state data

<table>
<thead>
<tr>
<th>Operation</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stroking</td>
<td>Stroke properties</td>
<td><code>stroke_props</code></td>
</tr>
<tr>
<td>Stroking</td>
<td>Dashes</td>
<td><code>dashes</code></td>
</tr>
<tr>
<td>Masking</td>
<td>Mask properties</td>
<td><code>mask_props</code></td>
</tr>
</tbody>
</table>

### 15.3.2.6 State data default values

For all rendering and composing operations, the state data objects named above are provided using `optional<T>` class template arguments.

If there is no contained value for a state data object, it is interpreted as-if the `optional<T>` argument contained a default constructed object of the relevant state data object.

### 15.3.3 Standard coordinate spaces

There are four standard coordinate spaces relevant to the rendering and composing operations (15.3.2):

1. The **brush coordinate space** is the standard coordinate space of the source brush (15.3.2.2). Its transformation matrix is the brush properties’ brush matrix (14.10.1).
2. The **mask coordinate space** is the standard coordinate space of the mask brush (15.3.2.2). Its transformation matrix is the mask properties’ mask matrix (14.14.1).
3. The **user coordinate space** is the standard coordinate space of `basic_interpreted_path` objects. Its transformation matrix is a default-constructed `basic_matrix_2d`.
4. The **surface coordinate space** is the standard coordinate space of the surface object’s visual data. Its transformation matrix is the surface properties’ surface matrix (14.9.1).

Given a point `pt`, a brush coordinate space transformation matrix `bcsm`, a mask coordinate space transformation matrix `mcsm`, a user coordinate space transformation matrix `ucsm`, and a surface coordinate space transformation matrix `scsm`, the following table describes how to transform it from each of these standard coordinate spaces to the other standard coordinate spaces:
Table 36 — Point transformations

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>brush coordinate space</td>
<td>mask coordinate space</td>
<td>mcsm.transform_pt(bcsm.invert().transform_pt(pt)).</td>
</tr>
<tr>
<td>brush coordinate space</td>
<td>user coordinate space</td>
<td>bcsm.invert().transform_pt(pt).</td>
</tr>
<tr>
<td>brush coordinate space</td>
<td>surface coordinate space</td>
<td>scsm.transform_pt(bcsm.invert().transform_pt(pt)).</td>
</tr>
<tr>
<td>user coordinate space</td>
<td>brush coordinate space</td>
<td>bcsm.transform_pt(pt).</td>
</tr>
<tr>
<td>user coordinate space</td>
<td>mask coordinate space</td>
<td>mcsm.transform_pt(pt).</td>
</tr>
<tr>
<td>user coordinate space</td>
<td>surface coordinate space</td>
<td>scsm.transform_pt(pt).</td>
</tr>
<tr>
<td>surface coordinate space</td>
<td>brush coordinate space</td>
<td>bcsm.transform_pt(scsm.invert().transform_pt(pt)).</td>
</tr>
<tr>
<td>surface coordinate space</td>
<td>mask coordinate space</td>
<td>mcsm.transform_pt(scsm.invert().transform_pt(pt)).</td>
</tr>
<tr>
<td>surface coordinate space</td>
<td>user coordinate space</td>
<td>scsm.invert().transform_pt(pt).</td>
</tr>
</tbody>
</table>

15.3.4 surface painting

When a painting operation is initiated on a surface, the implementation shall produce results as-if the following steps were performed:

1. For each integral point \(sp\) of the surface’s visual data, determine if \(sp\) is within the clip area (14.11.1); if so, proceed with the remaining steps.
2. Transform \(sp\) from the surface coordinate space (15.3.3) to the brush coordinate space (Table 36), resulting in point \(bp\).
3. Sample from point \(bp\) of the source brush (15.3.2.2), combine the resulting visual data with the visual data at point \(sp\) in the surface’s visual data in the manner specified by the surface’s current compositing operator (14.9.1), and modify the visual data of the surface at point \(sp\) to reflect the result produced by application of the compositing operator.

15.3.5 surface filling

When a filling operation is initiated on a surface, the implementation shall produce results as-if the following steps were performed:

1. For each integral point \(sp\) of the surface’s visual data, determine if \(sp\) is within the clip area (14.11.1); if so, proceed with the remaining steps.
2. Transform \(sp\) from the surface coordinate space (15.3.3) to the user coordinate space (Table 36), resulting in point \(up\).
3. Using the source path (15.3.2.3) and the fill rule (14.10.1), determine whether \(up\) shall be filled; if so, proceed with the remaining steps.
4. Transform \(up\) from the user coordinate space to the brush coordinate space (15.3.3 and Table 36), resulting in point \(bp\).
5. Sample from point \(bp\) of the source brush (15.3.2.2), combine the resulting visual data with the visual data at point \(sp\) in the surface’s visual data in the manner specified by the surface’s current compositing operator (14.9.1), and modify the surface’s visual data at point \(sp\) to reflect the result produced by application of the compositing operator.

15.3.6 surface stroking

When a stroking operation is initiated on a surface, it is carried out for each figure in the source path (15.3.2).

The following rules shall apply when a stroking operation is carried out on a figure:

1. No part of the surface’s visual data that is outside of the clip area shall be modified.
2. If the figure is a closed figure, then the point where the end point of its final segment meets the start point of the initial segment shall be rendered as specified by the line join value (see: 14.12.1 and 15.3.2.5); otherwise the start point of the initial segment and end point of the final segment shall each by rendered as specified by the line cap value. The remaining meetings between successive end points and start points shall be rendered as specified by the line join value.

3. If the dash pattern (Table 35) has its default value or if its vector<float> member is empty, the segments shall be rendered as a continuous path.

4. If the dash pattern’s vector<float> member contains only one value, that value shall be used to define a repeating pattern in which the path is shown then hidden. The ends of each shown portion of the path shall be rendered as specified by the line cap value.

5. If the dash pattern’s vector<float> member contains two or more values, the values shall be used to define a pattern in which the figure is alternatively rendered then not rendered for the length specified by the value. The ends of each rendered portion of the figure shall be rendered as specified by the line cap value. If the dash pattern’s float member, which specifies an offset value, is not 0.0f, the meaning of its value is implementation-defined. If a rendered portion of the figure overlaps a not rendered portion of the figure, the rendered portion shall be rendered.

When a stroking operation is carried out on a figure, the width of each rendered portion shall be the line width (see: 14.12.1 and 15.3.2.5). Ideally this means that the diameter of the stroke at each rendered point should be equal to the line width. However, because there are an infinite number of points along each rendered portion, implementations may choose an unspecified method of determining minimum distances between points along each rendered portion and the diameter of the stroke between those points shall be the same. [Note: This concept is sometimes referred to as a tolerance. It allows for a balance between precision and performance, especially in situations where the end result is in a non-exact format such as raster graphics data. —end note]

After all figures in the path have been rendered but before the rendered result is composed to the surface’s visual data, the rendered result shall be transformed from the user coordinate space (15.3.3) to the surface coordinate space (15.3.3).

15.3.7 surface masking

When a masking operation is initiated on a surface, the implementation shall produce results as-if the following steps were performed:

1. For each integral point sp of the surface’s visual data, determine if sp is within the clip area (14.11.1); if so, proceed with the remaining steps.
2. Transform sp from the surface coordinate space (15.3.3) to the mask coordinate space (Table 36), resulting in point mp.
3. Sample the alpha channel from point mp of the mask brush and store the result in mac; if the visual data format of the mask brush does not have an alpha channel, the value of mac shall always be 1.0.
4. Transform sp from the surface coordinate space to the brush coordinate space, resulting in point bp.
5. Sample from point bp of the source brush (15.3.2.2), combine the resulting visual data with the surface’s visual data at point sp in the manner specified by the surface’s current compositing operator (14.9.1), multiply each channel of the result produced by application of the compositing operator by map if the visual data format of the surface’s visual data is a premultiplied format and if not then just multiply the alpha channel of the result by map, and modify the surface’s visual data at point sp to reflect the multiplied result.

15.3.8 output surface miscellaneous behavior

What constitutes an output device is implementation-defined, with the sole constraint being that an output device must allow the user to see the dynamically-updated contents of the display buffer. [Example: An output device might be a window in a windowing system environment or the usable screen area of a smart phone or tablet. —end example]

Implementations may allow more than one basic_output_surface object, basic_unmanaged_output_surface object, or a combination thereof to exist and be displayed co-synchronously. [Note: In windowing environments, implementations would likely support multiple objects of these types. In contrast, on a
smart phone it is unlikely that an implementation would support multiple objects of these types due to environmental and platform constraints. — end note]

3 It is not required that implementations support the existence of any basic_unmanaged_output_surface objects. See Table 16.

4 All functions that perform rendering and composing operations operate on the back buffer. <TODO> Specify on a per-function basis rather than with a blanket statement that is probably inaccurate.

15.3.9 output surface state

Table 37 specifies the name, type, function, and default value for each item of a display surface’s observable state.

Table 37 — Output surface observable state

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letterbox brush</td>
<td>brush</td>
<td>This is the brush that shall be used as specified by scaling::letterbox (Table 30)</td>
<td>brush{ { rgba_color::black } }</td>
</tr>
<tr>
<td>Letterbox brush props</td>
<td>brush_props</td>
<td>This is the brush properties for the letterbox brush</td>
<td>brush_props{ }</td>
</tr>
<tr>
<td>Scaling type</td>
<td>scaling</td>
<td>When the user scaling callback is equal to its default value, this is the type of scaling that shall be used when transferring the back buffer to the display buffer</td>
<td>scaling::letterbox</td>
</tr>
<tr>
<td>Draw width</td>
<td>int</td>
<td>The width in pixels of the back buffer. The minimum value is 1. The maximum value is unspecified. Because users can only request a preferred value for the draw width when setting and altering it, the maximum value may be a run-time determined value. If the preferred draw width exceeds the maximum value, then if a preferred draw height has also been supplied then implementations should provide a back buffer with the largest dimensions possible that maintain as nearly as possible the aspect ratio between the preferred draw width and the preferred draw height otherwise implementations should provide a back buffer with the largest dimensions possible that maintain as nearly as possible the aspect ratio between the preferred draw width and the current draw height.</td>
<td>N/A [Note: It is impossible to create an output surface object without providing a preferred draw width value; as such a default value cannot exist. — end note]</td>
</tr>
</tbody>
</table>
Table 37 — Output surface observable state (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Function</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draw height</td>
<td>int</td>
<td>The height in pixels of the back buffer. The minimum value is 1. The maximum value is unspecified. Because users can only request a preferred value for the draw height when setting and altering it, the maximum value may be a run-time determined value. If the preferred draw height exceeds the maximum value, then if a preferred draw width has also been supplied then implementations should provide a back buffer with the largest dimensions possible that maintain as nearly as possible the aspect ratio between the preferred draw width and the preferred draw height otherwise implementations should provide a back buffer with the largest dimensions possible that maintain as nearly as possible the aspect ratio between the current draw width and the preferred draw height</td>
<td>N/A [Note: It is impossible to create an output surface object without providing a preferred draw height value; as such a default value cannot exist. — end note]</td>
</tr>
<tr>
<td>Draw format</td>
<td>format</td>
<td>The pixel format of the back buffer. When an output surface object is created, a preferred pixel format value is provided. If the implementation does not support the preferred pixel format value as the value of draw format, the resulting value of draw format is implementation-defined</td>
<td>N/A [Note: It is impossible to create an output surface object without providing a preferred draw format value; as such a default value cannot exist. — end note]</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Function</td>
<td>Default value</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>--------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Display width</td>
<td>int</td>
<td>The width in pixels of the display buffer. The minimum value is unspecified. The maximum value is unspecified. Because users can only request a preferred value for the display width when setting and altering it, both the minimum value and the maximum value may be run-time determined values. If the preferred display width is not within the range between the minimum value and the maximum value, inclusive, then if a preferred display height has also been supplied then implementations should provide a display buffer with the largest dimensions possible that maintain as nearly as possible the aspect ratio between the preferred display width and the preferred display height otherwise implementations should provide a display buffer with the largest dimensions possible that maintain as nearly as possible the aspect ratio between the preferred display width and the current display height.</td>
<td>N/A [Note: It is impossible to create an output surface object without providing a preferred display width value since in the absence of an explicit display width argument the mandatory preferred draw width argument is used as the preferred display width; as such a default value cannot exist. — end note]</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Function</td>
<td>Default value</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------</td>
<td>---------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Display height</td>
<td>int</td>
<td>The height in pixels of the display buffer. The minimum value is unspecified. The maximum value is unspecified. Because users can only request a preferred value for the display height when setting and altering it, both the minimum value and the maximum value may be run-time determined values. If the preferred display height is not within the range between the minimum value and the maximum value, inclusive, then if a preferred display width has also been supplied then implementations should provide a display buffer with the largest dimensions possible that maintain as nearly as possible the aspect ratio between the preferred display width and the preferred display height. If no preferred display width has been supplied then implementations should provide a display buffer with the largest dimensions possible that maintain as nearly as possible the aspect ratio between the current display width and the preferred display height.</td>
<td>N/A [Note: It is impossible to create an output surface object without providing a preferred display height value since in the absence of an explicit display height argument the mandatory preferred draw height argument is used as the preferred display height; as such a default value cannot exist. — end note]</td>
</tr>
<tr>
<td>Auto clear</td>
<td>bool</td>
<td>If true the implementation shall call clear, which shall clear the back buffer, immediately before it executes the draw callback.</td>
<td>false</td>
</tr>
<tr>
<td>Refresh style</td>
<td>refresh_style</td>
<td>The refresh_style value that determines when the draw callback shall be called while basic_output_surface&lt;T&gt;::begin_show is being executed.</td>
<td>refresh_style::as_fast_as_possible</td>
</tr>
<tr>
<td>Desired frame rate</td>
<td>float</td>
<td>This value is the number of times the draw callback shall be called per second while basic_output_surface&lt;T&gt;::begin_show is being executed when the value of refresh style is refresh_style::fixed, subject to the additional requirements documented in the meaning of refresh_style::fixed (See: Table 31).</td>
<td>30.0f</td>
</tr>
</tbody>
</table>
15.4 Class basic_image_surface

15.4.1 basic_image_surface summary

The class basic_image_surface provides an interface to raster graphics data.

It has a pixel format of type format, a width of type int, and a height of type int.

[Note: Because of the functionality it provides and what it can be used for, it is expected that developers familiar with other graphics technologies will think of the basic_image_surface class as being a form of render target. This is intentional, though this Technical Specification does not formally define or use that term to avoid any minor ambiguities and differences in its meaning between the various graphics technologies that do use the term render target. — end note]

15.4.2 basic_image_surface synopsis

namespace std::experimental::io2d::v1 {

template <class GraphicsSurfaces>
class basic_image_surface {

public:
  using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;

  // 15.4.3, construct/copy/move/destroy:
  basic_image_surface(io2d::format fmt, int width, int height);
  basic_image_surface(filesystem::path f, io2d::image_file_format iff, io2d::format fmt);
  basic_image_surface(filesystem::path f, io2d::image_file_format iff, io2d::format fmt,
                      error_code& ec) noexcept;
  basic_image_surface(basic_image_surface&&) noexcept;
  basic_image_surface& operator=(basic_image_surface&&) noexcept;

  // 15.4.4, members:
  void save(filesystem::path p, image_file_format i);
  void save(filesystem::path p, image_file_format i, error_code& ec) noexcept;

  // 15.4.5, static members:
  static basic_display_point<graphics_math_type> max_dimensions() noexcept;

  // 15.4.6, observers:
  io2d::format format() const noexcept;
  basic_display_point<graphics_math_type> dimensions() const noexcept;

  // 15.4.7, modifiers:
  void clear();
  void paint(const basic_brush<GraphicsSurfaces>& b,
             const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
             const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
             const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
  template <class Allocator>
  void stroke(const basic_brush<GraphicsSurfaces>& b,
              const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
              const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
              const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
              const optional<basic_dashes<GraphicsSurfaces>>& d = nullopt,
              const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
  void stroke(const basic_brush<GraphicsSurfaces>& b,
              const basic_interpreted_path<GraphicsSurfaces>& ip,
              const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
              const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
              const optional<basic_dashes<GraphicsSurfaces>>& d = nullopt,
              const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
  template <class Allocator>
  void fill(const basic_brush<GraphicsSurfaces>& b,
            const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
            const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
void fill(const basic_brush<GraphicsSurfaces>& b,
const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
void mask(const basic_brush<GraphicsSurfaces>& b,
const basic_brush<GraphicsSurfaces>& mb,
const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
const optional<basic_mask_props<GraphicsSurfaces>>& mp = nullopt,
const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
};
}

15.4.3 basic_image_surface constructors and assignment operators

basic_image_surface(io2d::format fmt, int w, int h);

1 Requires: \( w \) is greater than 0 and not greater than \( \text{basic_image_surface::max_width()} \).
2 \( h \) is greater than 0 and not greater than \( \text{basic_image_surface::max_height()} \).
3 fmt is not io2d::format::invalid.
4 Effects: Constructs an object of type basic_image_surface.
5 The pixel format is fmt, the width is \( w \), and the height is \( h \).

basic_image_surface(filesystem::path f, io2d::image_file_format i, io2d::format fmt);
basic_image_surface(filesystem::path f, io2d::image_file_format i, io2d::format fmt,
error_code& ec) noexcept;

6 Requires: \( f \) is a file and its contents are data in either JPEG format, TIFF format or PNG format.
7 fmt is not io2d::format::invalid.
8 Effects: Constructs an object of type basic_image_surface.
9 The raster graphics data is the result of processing \( f \) into uncompressed raster graphics in the manner
specified by the standard that describes how to transform the contents of data contained in \( f \) into raster
graphics data and then transforming that transformed raster graphics data into the format specified by
fmt.
10 The data of \( f \) is processed into uncompressed raster graphics data as specified by the value of \( i \).
11 If \( i \) is image_file_format::unknown, implementations may attempt to process the data of \( f \) into
uncompressed raster graphics data. The manner in which it does so is unspecified. If no uncompressed
raster graphics data is produced, the error specified below occurs.
12 [ Note: The intent of image_file_format::unknown is to allow implementations to support image file
formats that are not required to be supported. — end note ]
13 If the width of the uncompressed raster graphics data would be less than 1 or greater than \( \text{basic_-}
image_surface::max_width() \) or if the height of the uncompressed raster graphics data would be less
than 1 or greater than \( \text{basic_image_surface::max_height()} \), the error specified below occurs.
14 The resulting uncompressed raster graphics data is then transformed into the data format specified by
fmt. If the format specified by fmt only contains an alpha channel, the values of the color channels, if
any, of the surface’s visual data are unspecified. If the format specified by fmt only contains color
channels and the resulting uncompressed raster graphics data is in a premultiplied format, then the
value of each color channel for each pixel is be divided by the value of the alpha channel for that pixel.
The visual data is then set as the visual data of the surface.
15 The width is the width of the uncompressed raster graphics data. The height is the height of the
uncompressed raster graphics data.
16 Throws: As specified in Error reporting (Clause 4).
**Error conditions:** Any error that could result from trying to access \( f \), open \( f \) for reading, or reading data from \( f \).

```
errc::not_supported if image_file_format::unknown is passed as an argument and the implementation is unable to determine the file format or does not support saving in the image file format it determined.
```

```
errc::invalid_argument if fmt is io2d::format::invalid.
```

```
errc::argument_out_of_domain if the width would be less than 1, the width would be greater than basic_image_surface::max_width(), the height would be less than 1, or the height would be greater than basic_image_surface::max_height().
```

### 15.4.4 basic_image_surface members

[io2d.imagesurface.members]

```
void save(filesystem::path p, image_file_format i);
void save(filesystem::path p, image_file_format i, error_code& ec) noexcept;
```

1. **Requires:** \( p \) shall be a valid path to a file. The file need not exist provided that the other components of the path are valid.
2. If the file exists, it shall be writable. If the file does not exist, it shall be possible to create the file at the specified path and then the created file shall be writable.
3. **Effects:** Any pending rendering and composing operations (15.3.2) are performed.
4. The surface’s visual data is written to \( p \) in the data format specified by \( i \).
5. If \( i \) is image_file_format::unknown, it is implementation-defined whether the surface is saved in the image file format, if any, that the implementation associates with \( p.extension() \) provided that \( p.has_extension() == true \). If \( p.has_extension() == false \), the implementation does not associate an image file format with \( p.extension() \), or the implementation does not support saving in that image file format, the error specified below occurs.
6. **Throws:** As specified in Error reporting (Clause 4).
7. **Error conditions:** Any error that could result from trying to create \( f \), access \( f \), or write data to \( f \).
8. ```
errc::not_supported if image_file_format::unknown is passed as an argument and the implementation is unable to determine the file format or does not support saving in the image file format it determined.
```

### 15.4.5 basic_image_surface static members

[io2d.imagesurface.staticmembers]

```static basic_display_point<graphics_math_type> max_dimensions() noexcept;```

1. **Returns:** <TODO> The maximum height and width for a basic_image_surface object.

### 15.4.6 basic_image_surface observers

[io2d.imagesurface.observers]

```io2d::format format() const noexcept;```

1. **Returns:** The pixel format.

```basic_display_point<graphics_math_type> dimensions() const noexcept;```

2. **Returns:** <TODO> The height and width.

### 15.4.7 basic_image_surface modifiers

[io2d.imagesurface.modifiers]

```void clear();```

1. **Effects:** <TODO>

```void paint(const basic_brush<GraphicsSurfaces>& b, ```

2. **Effects:** Performs the painting rendering and composing operation as specified by 15.3.4.
3. The meanings of the parameters are specified by 15.3.2.
4. **Throws:** As specified in Error reporting (Clause 4).
Error conditions: The errors, if any, produced by this function are implementation-defined.

```cpp
template <class Allocator>
void stroke(const basic_brush<GraphicsSurfaces>& b,
            const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
            const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
            const optional<basic_stroke_props<GraphicsSurfaces>>& sp = nullopt,
            const optional<basic_dashes<GraphicsSurfaces>>& d = nullopt,
            const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
            const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
void stroke(const basic_brush<GraphicsSurfaces>& b,
            const basic_interpreted_path<GraphicsSurfaces>& ip,
            const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
            const optional<basic_stroke_props<GraphicsSurfaces>>& sp = nullopt,
            const optional<basic_dashes<GraphicsSurfaces>>& d = nullopt,
            const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
            const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
```

Effects: Performs the stroking rendering and composing operation as specified by 15.3.6.

The meanings of the parameters are specified by 15.3.2.

Throws: As specified in Error reporting (Clause 4).

Error conditions: The errors, if any, produced by this function are implementation-defined.

```cpp
template <class Allocator>
void fill(const basic_brush<GraphicsSurfaces>& b,
          const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
          const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
          const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
          const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
void fill(const basic_brush<GraphicsSurfaces>& b,
          const basic_interpreted_path<GraphicsSurfaces>& ip,
          const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
          const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
          const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
```

Effects: Performs the filling rendering and composing operation as specified by 15.3.5.

The meanings of the parameters are specified by 15.3.2.

Throws: As specified in Error reporting (Clause 4).

Error conditions: The errors, if any, produced by this function are implementation-defined.

```cpp
void mask(const basic_brush<GraphicsSurfaces>& b,
          const basic_brush<GraphicsSurfaces>& mb,
          const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
          const optional<basic_mask_props<GraphicsSurfaces>>& mp = nullopt,
          const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
          const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
```

Effects: Performs the masking rendering and composing operation as specified by 15.3.7.

The meanings of the parameters are specified by 15.3.2.

Throws: As specified in Error reporting (Clause 4).

Error conditions: The errors, if any, produced by this function are implementation-defined.
class basic_output_surface {
public:
  using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;

  // 15.5.3. constructors:
  basic_output_surface(int preferredWidth, int preferredHeight,
                        io2d::format preferredFormat,
                        io2d::scaling scl = io2d::scaling::letterbox,
                        io2d::refresh_style rr = io2d::refresh_style::as_fast_as_possible,
                        float fps = 30.0f);
  basic_output_surface(int preferredWidth, int preferredHeight,
                        io2d::format preferredFormat,
                        error_code& ec, io2d::scaling scl = io2d::scaling::letterbox,
                        io2d::refresh_style rr = io2d::refresh_style::as_fast_as_possible,
                        float fps = 30.0f) noexcept;
  basic_output_surface(int preferredWidth, int preferredHeight,
                        io2d::format preferredFormat, int preferredDisplayWidth,
                        int preferredDisplayHeight, io2d::format preferredDisplayFormat,
                        io2d::scaling scl = io2d::scaling::letterbox,
                        io2d::refresh_style rr = io2d::refresh_style::as_fast_as_possible,
                        float fps = 30.0f);
  basic_output_surface(int preferredWidth, int preferredHeight,
                        io2d::format preferredFormat, int preferredDisplayWidth,
                        int preferredDisplayHeight, io2d::format preferredDisplayFormat,
                        error_code& ec, io2d::scaling scl = io2d::scaling::letterbox,
                        io2d::refresh_style rr = io2d::refresh_style::as_fast_as_possible,
                        float fps = 30.0f) noexcept;

  // 15.5.4. modifiers:
  int begin_show();
  void end_show();
  void clear();
  void paint(const basic_brush<GraphicsSurfaces>& b,
             const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
             const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
             const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
  template <class Allocator>
  void stroke(const basic_brush<GraphicsSurfaces>& b,
              const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
              const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
              const optional<basic_stroke_props<GraphicsSurfaces>>& sp = nullopt,
              const optional<basic_dashes<GraphicsSurfaces>>& d = nullopt,
              const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
              const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
  template <class Allocator>
  void fill(const basic_brush<GraphicsSurfaces>& b,
            const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
            const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
            const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
            const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
  void paint(const basic_brush<GraphicsSurfaces>& b,
             const basic_interpreted_path<GraphicsSurfaces>& ip,
             const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
             const optional<basic_stroke_props<GraphicsSurfaces>>& sp = nullopt,
             const optional<basic_dashes<GraphicsSurfaces>>& d = nullopt,
             const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
             const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
  template <class Allocator>
  void stroke(const basic_brush<GraphicsSurfaces>& b,
              const basic_interpreted_path<GraphicsSurfaces>& ip,
              const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
              const optional<basic_stroke_props<GraphicsSurfaces>>& sp = nullopt,
              const optional<basic_dashes<GraphicsSurfaces>>& d = nullopt,
              const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
              const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
  template <class Allocator>
  void fill(const basic_brush<GraphicsSurfaces>& b,
            const basic_interpreted_path<GraphicsSurfaces>& ip,
            const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
            const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
            const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
};
void mask(const basic_brush<GraphicsSurfaces>& b,
    const basic_brush<GraphicsSurfaces>& mb,
    const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
    const optional<basic_mask_props<GraphicsSurfaces>>& mp = nullopt,
    const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
    const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);

void draw_callback(const function<void(basic_output_surface& sfc)>& fn);
void size_change_callback(const function<void(basic_output_surface& sfc)>& fn);
void dimensions(basic_display_point<graphics_math_type> dp);
void dimensions(basic_display_point<graphics_math_type> dp, error_code& ec) noexcept;
void output_dimensions(basic_display_point<graphics_math_type> dp);
void output_dimensions(basic_display_point<graphics_math_type> dp,
    error_code& ec) noexcept;
void scaling(io2d::scaling scl) noexcept;
void user_scaling_callback(const function<basic_bounding_box<graphics_math_type>(const basic_output_surface&, bool&)>& fn);
void letterbox_brush(const optional<basic_brush<GraphicsSurfaces>>& b,
    const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt) noexcept;
void letterbox_brush_props(const optional<basic_brush_props<GraphicsSurfaces>>& bp) noexcept;
void auto_clear(bool val) noexcept;
void redraw_required(bool val = true) noexcept;

// 15.5.5, observers:
io2d::format format() const noexcept;
basic_display_point<graphics_math_type> dimensions() const noexcept;
basic_display_point<graphics_math_type> max_dimensions() const noexcept;
basic_display_point<graphics_math_type> output_dimensions() const noexcept;
basic_display_point<graphics_math_type> max_output_dimensions() const noexcept;
io2d::scaling scaling() const noexcept;
online<basic_brush<GraphicsSurfaces>> letterbox_brush() const noexcept;
online<basic_brush_props<GraphicsSurfaces>> letterbox_brush_props() const noexcept;
bool auto_clear() const noexcept;

15.5.3 basic_output_surface constructors

basic_output_surface(int preferredWidth, int preferredHeight,
io2d::format preferredFormat,
io2d::scaling scl = io2d::scaling::letterbox,
io2d::refresh_style rr = io2d::refresh_style::as_fast_as_possible,
float fps = 30.0f);
basic_output_surface(int preferredWidth, int preferredHeight,
io2d::format preferredFormat,
error_code& ec, io2d::scaling scl = io2d::scaling::letterbox,
io2d::refresh_style rr = io2d::refresh_style::as_fast_as_possible,
float fps = 30.0f) noexcept;

basic_output_surface(int preferredWidth, int preferredHeight,
io2d::format preferredFormat, int preferredDisplayWidth,
io2d::format preferredDisplayFormat,
io2d::scaling scl = io2d::scaling::letterbox,
io2d::refresh_style rr = io2d::refresh_style::as_fast_as_possible,
float fps = 30.0f);
basic_output_surface(int preferredWidth, int preferredHeight,
io2d::format preferredFormat, int preferredDisplayWidth,
preferredDisplayHeight, io2d::format preferredDisplayFormat,
io2d::scaling scl = io2d::scaling::letterbox,
io2d::refresh_style rr = io2d::refresh_style::as_fast_as_possible,
float fps = 30.0f) noexcept;

1 <TODO>

2 <TODO>
§ 15.5.4  basic_output_surface modifiers

void begin_show();

Effects: Performs the following actions in a continuous loop:

1. Handle any implementation and host environment matters. If there are no pending implementation or host environment matters to handle, proceed immediately to the next action.
2. Run the size change callback if doing so is required by its specification and it does not have a value equivalent to its default value.
3. If the refresh style requires that the draw callback be called then:
   a) Evaluate auto clear and perform the actions required by its specification, if any.
   b) Run the draw callback.
   c) Ensure that all operations from the draw callback that can effect the back buffer have completed.
   d) Transfer the contents of the back buffer to the display buffer using sampling with an unspecified filter. If the user scaling callback does not have a value equivalent to its default value, use it to determine the position where the contents of the back buffer shall be transferred to and whether or not the letterbox brush should be used. Otherwise use the value of scaling type to determine the position and whether the letterbox brush should be used.

If basic_output_surface::end_show is called from the draw callback, the implementation shall finish executing the draw callback and shall immediately cease to perform any actions in the continuous loop other than handling any implementation and host environment matters needed to exit the loop properly.

No later than when this function returns, the output device shall cease to display the contents of the display buffer.

What the output device shall display when it is not displaying the contents of the display buffer is unspecified.

Returns: The possible values and meanings of the possible values returned are implementation-defined.

Throws: As specified in Error reporting (Clause 4).

Remarks: Since this function calls the draw callback and can call the size change callback and the user scaling callback, in addition to the errors documented below, any errors that the callback functions produce can also occur.

Error conditions: errc::operation_would_block if the value of draw callback is equivalent to its default value or if it becomes equivalent to its default value before this function returns.

Other errors, if any, produced by this function are implementation-defined.

void end_show();

Effects: If this function is called outside of the draw callback while it is being executed in the basic_output_surface::begin_show function’s continuous loop, it does nothing.

Otherwise, the implementation initiates the process of exiting the basic_output_surface::begin_show function’s continuous loop.

If possible, any procedures that the host environment requires in order to cause the basic_output_surface::show function’s continuous loop to stop executing without error should be followed.

The basic_output_surface::begin_show function’s loop continues execution until it returns.

void clear();

Effects: <TODO>

void paint(const basic_brush<GraphicsSurfaces>& b,
        const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
        const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
        const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);

Effects: Performs the painting rendering and composing operation as specified by 15.3.4.
The meanings of the parameters are specified by 15.3.2.

**Throws:** As specified in Error reporting (Clause 4).

**Error conditions:** The errors, if any, produced by this function are implementation-defined.

template <class Allocator>
void stroke(const basic_brush<GraphicsSurfaces>& b,
            const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
            const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
            const optional<basic_stroke_props<GraphicsSurfaces>>& sp = nullopt,
            const optional<basic_dashes<GraphicsSurfaces>>& d = nullopt,
            const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
            const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);

void stroke(const basic_brush<GraphicsSurfaces>& b,
            const basic_interpreted_path<GraphicsSurfaces>& ip,
            const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
            const optional<basic_stroke_props<GraphicsSurfaces>>& sp = nullopt,
            const optional<basic_dashes<GraphicsSurfaces>>& d = nullopt,
            const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
            const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);

**Effects:** Performs the stroking rendering and composing operation as specified by 15.3.6.

The meanings of the parameters are specified by 15.3.2.

**Throws:** As specified in Error reporting (Clause 4).

**Error conditions:** The errors, if any, produced by this function are implementation-defined.

template <class Allocator>
void fill(const basic_brush<GraphicsSurfaces>& b,
          const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
          const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
          const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
          const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);

void fill(const basic_brush<GraphicsSurfaces>& b,
          const basic_interpreted_path<GraphicsSurfaces>& ip,
          const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
          const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
          const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);

**Effects:** Performs the filling rendering and composing operation as specified by 15.3.5.

The meanings of the parameters are specified by 15.3.2.

**Throws:** As specified in Error reporting (Clause 4).

**Error conditions:** The errors, if any, produced by this function are implementation-defined.

void mask(const basic_brush<GraphicsSurfaces>& b,
          const basic_brush<GraphicsSurfaces>& mb,
          const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
          const optional<basic_mask_props<GraphicsSurfaces>>& mp = nullopt,
          const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
          const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);

**Effects:** Performs the masking rendering and composing operation as specified by 15.3.7.

The meanings of the parameters are specified by 15.3.2.

**Throws:** As specified in Error reporting (Clause 4).

**Error conditions:**

The errors, if any, produced by this function are implementation-defined.

void draw_callback(const function<void(basic_output_surface& sfc)>& fn);

**Effects:** <TODO>

void size_change_callback(const function<void(basic_output_surface& sfc)>& fn);

**Effects:** <TODO>
void dimensions(basic_display_point<graphics_math_type> dp);
void dimensions(basic_display_point<graphics_math_type> dp, error_code& ec) noexcept;

Effects: <TODO>

void display_dimensions(basic_display_point<graphics_math_type> dp);
void display_dimensions(basic_display_point<graphics_math_type> dp, error_code& ec) noexcept;

Effects: <TODO>

void scaling(io2d::scaling scl) noexcept;

Effects: <TODO>

void user_scaling_callback(const function<basic_bounding_box<graphics_math_type>(const basic_output_surface&, bool&)>& fn);

Effects: <TODO>

void letterbox_brush(const optional<basic_brush<GraphicsSurfaces>>& b,
const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt) noexcept;
void letterbox_brush_props(const optional<basic_brush_props<GraphicsSurfaces>>& bp) noexcept;

Effects: <TODO>

void auto_clear(bool val) noexcept;
Effects: <TODO>

void redraw_required(bool val = true) noexcept;
Effects: <TODO>

\section{basic_output_surface observers \[io2d.outputsurface.observers\]}

io2d::format format() const noexcept;
Returns: The pixel format.

basic_display_point<graphics_math_type> dimensions() const noexcept;
Return: <TODO>The height and width.

basic_display_point<graphics_math_type> max_dimensions() const noexcept;
Returns: <TODO>The maximum available height and width of a basic_output_surface for the device.

basic_display_point<graphics_math_type> display_dimensions() const noexcept;
Returns: <TODO>

basic_display_point<graphics_math_type> max_output_dimensions() const noexcept;
Returns: <TODO>

io2d::scaling scaling() const noexcept;
Returns: The scaling type.

optional<basic_brush<GraphicsSurfaces>> letterbox_brush() const noexcept;
Returns: An optional<basic_brush<GraphicsSurfaces>> object constructed using the user-provided letterbox brush or, if the letterbox brush is set to its default value, an empty optional<basic_brush<GraphicsSurfaces>> object.

optional<basic_brush_props<GraphicsSurfaces>> letterbox_brush_props() const noexcept;
Returns: An optional<basic_brush_props<GraphicsSurfaces>> object constructed using the user-provided letterbox brush props or, if the letterbox brush props is set to its default value, an empty optional<basic_brush_props<GraphicsSurfaces>> object.

bool auto_block() const noexcept;
Returns: The value of auto clear.
15.6 Class `basic_unmanaged_output_surface`  

15.6.1 `basic_unmanaged_output_surface` summary

```
namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    class basic_unmanaged_output_surface {
        public:
            using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
            using data_type = typename GraphicsSurfaces::surfaces::unmanaged_output_surface_data_type;

            // 15.6.3, constructor:
            basic_unmanaged_output_surface(data_type&& data) noexcept;

            // 15.6.4, observers:
            bool has_draw_callback() const noexcept;
            bool has_size_change_callback() const noexcept;
            bool has_user_scaling_callback() const noexcept;
            io2d::format format() const noexcept;
            basic_display_point<graphics_math_type> dimensions() const noexcept;
            basic_display_point<graphics_math_type> max_dimensions() const noexcept;
            basic_display_point<graphics_math_type> display_dimensions() const noexcept;
            basic_display_point<graphics_math_type> max_display_dimensions() const noexcept;
            io2d::scaling scaling() const noexcept;
            optional<basic_brush<GraphicsSurfaces>> letterbox_brush() const noexcept;
            optional<basic_brush_props<GraphicsSurfaces>> letterbox_brush_props() const noexcept;
            bool auto_clear() const noexcept;

            // 15.6.5, modifiers:
            void invoke_draw_callback();
            void invoke_size_change_callback();
            void draw_to_output();
            void clear();
            void paint(const basic_brush<GraphicsSurfaces>& b,
                        const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
                        const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
                        const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
            template <class Allocator>
            void stroke(const basic_brush<GraphicsSurfaces>& b,
                        const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
                        const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
                        const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
                        const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
            void stroke(const basic_brush<GraphicsSurfaces>& b,
                        const basic_interpreted_path<GraphicsSurfaces>& ip,
                        const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
                        const optional<basic.stroke_props<GraphicsSurfaces>>& sp = nullopt,
                        const optional<basic.dashes<GraphicsSurfaces>>& d = nullopt,
                        const optional<basic.render_props<GraphicsSurfaces>>& rp = nullopt,
                        const optional<basic.clip_props<GraphicsSurfaces>>& cl = nullopt);
            template <class Allocator>
            void fill(const basic_brush<GraphicsSurfaces>& b,
                      const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
                      const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
```
const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
void fill(const basic_brush<GraphicsSurfaces>& b,
const basic_interpreted_path<GraphicsSurfaces>& ip,
const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
void mask(const basic_brush<GraphicsSurfaces>& b,
const basic_brush<GraphicsSurfaces>& mb,
const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
const optional<basic_mask_props<GraphicsSurfaces>>& mp = nullopt,
const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
void draw_callback(
    const function<void(basic_unmanaged_output_surface& sfc)>& fn);
void size_change_callback(
    const function<void(basic_unmanaged_output_surface& sfc)>& fn);
void dimensions(basic_display_point<graphics_math_type> dp);
void dimensions(basic_display_point<graphics_math_type> dp, error_code& ec) noexcept;
void display_dimensions(basic_display_point<graphics_math_type> dp);
void display_dimensions(basic_display_point<graphics_math_type> dp,
    error_code& ec) noexcept;
void scaling(io2d::scaling scl) noexcept;
void letterbox_brush(const optional<basic_brush<GraphicsSurfaces>>& b,
    const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt) noexcept;
void letterbox_brush_props(const optional<basic_brush_props<GraphicsSurfaces>>& bp) noexcept;
void auto_clear(bool val) noexcept;
void redraw_required(bool val = true) noexcept;
}

15.6.3 basic_unmanaged_output_surface constructor
[io2d.unmanagedoutputsurface.cons]
basic_unmanaged_output_surface(data_type&& data) noexcept;
1  <TODO>

15.6.4 basic_unmanaged_output_surface observers
[io2d.unmanagedoutputsurface.observers]
bool has_draw_callback() const noexcept;
1  Returns: <TODO>

bool has_size_change_callback() const noexcept;
2  Returns: <TODO>
io2d::format format() const noexcept;
3  Returns: <TODO>
basic_display_point<graphics_math_type> dimensions() const noexcept;
4  Returns: <TODO>
basic_display_point<graphics_math_type> max_dimensions() const noexcept;
5  Returns: <TODO>
basic_display_point<graphics_math_type> display_dimensions() const noexcept;
6  Returns: <TODO>

§ 15.6.4 175
basic_display_point<graphics_math_type> max_display_dimensions() const noexcept;

Returns: <TODO>

io2d::scaling scaling() const noexcept;

Returns: <TODO>

optional<basic_brush<GraphicsSurfaces>> letterbox_brush() const noexcept;

Returns: <TODO>

optional<basic_brush_props<GraphicsSurfaces>> letterbox_brush_props() const noexcept;

Returns: <TODO>

bool auto_clear() const noexcept;

Returns: <TODO>

15.6.5 basic_unmanaged_output_surface modifiers

[io2d.unmanagedoutputsurface.modifiers]

void clear();

Effects: <TODO>

void paint(const basic_brush<GraphicsSurfaces>& b,
    const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
    const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
    const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);

Effects: Performs the painting rendering and composing operation as specified by 15.3.4.

The meanings of the parameters are specified by 15.3.2.

Throws: As specified in Error reporting (Clause 4).

Error conditions: The errors, if any, produced by this function are implementation-defined.

template <class Allocator>
void stroke(const basic_brush<GraphicsSurfaces>& b,
    const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
    const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
    const optional<basic_stroke_props<GraphicsSurfaces>>& sp = nullopt,
    const optional<basic_dashes<GraphicsSurfaces>>& d = nullopt,
    const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
    const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);

Effects: Performs the stroking rendering and composing operation as specified by 15.3.6.

The meanings of the parameters are specified by 15.3.2.

Throws: As specified in Error reporting (Clause 4).

Error conditions: The errors, if any, produced by this function are implementation-defined.

template <class Allocator>
void fill(const basic_brush<GraphicsSurfaces>& b,
    const basic_path_builder<GraphicsSurfaces, Allocator>& pb,
    const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
    const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
    const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);
void fill(const basic_brush<GraphicsSurfaces>& b,
const basic_interpreted_path<GraphicsSurfaces>& ip,
const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);

Effects: Performs the filling rendering and composing operation as specified by 15.3.5.
The meanings of the parameters are specified by 15.3.2.

Throws: As specified in Error reporting (Clause 4).

Error conditions: The errors, if any, produced by this function are implementation-defined.

void mask(const basic_brush<GraphicsSurfaces>& b,
const basic_brush<GraphicsSurfaces>& mb,
const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt,
const optional<basic_mask_props<GraphicsSurfaces>>& mp = nullopt,
const optional<basic_render_props<GraphicsSurfaces>>& rp = nullopt,
const optional<basic_clip_props<GraphicsSurfaces>>& cl = nullopt);

Effects: Performs the masking rendering and composing operation as specified by 15.3.7.
The meanings of the parameters are specified by 15.3.2.

Throws: As specified in Error reporting (Clause 4).

Error conditions: The errors, if any, produced by this function are implementation-defined.

void draw_callback(const function<void(basic_unmanaged_output_surface& sfc)>& fn);

Effects: <TODO>

void size_change_callback(const function<void(basic_unmanaged_output_surface& sfc)>& fn);

Effects: <TODO>

void dimensions(basic_display_point<graphics_math_type> dp);
void dimensions(basic_display_point<graphics_math_type> dp, error_code& ec) noexcept;

Effects: <TODO>

void display_dimensions(basic_display_point<graphics_math_type> dp);
void display_dimensions(basic_display_point<graphics_math_type> dp, error_code& ec) noexcept;

Effects: <TODO>

void scaling(io2d::scaling scl) noexcept;

Effects: <TODO>

void letterbox_brush(const optional<basic_brush<GraphicsSurfaces>>& b,
const optional<basic_brush_props<GraphicsSurfaces>>& bp = nullopt) noexcept;
void letterbox_brush_props(const optional<basic_brush_props<GraphicsSurfaces>>& bp) noexcept;

Effects: <TODO>

void auto_clear(bool val) noexcept;

Effects: <TODO>
16 Input

[io2d.input]

1 [Note: Input, such as keyboard, mouse, and touch, to user-visible surfaces will be added at a later date. This section is a placeholder. It is expected that input will be added via deriving from a user-visible surface. One possibility is that an io_surface class deriving from display_surface. This would allow developers to choose not to incur any additional costs of input support where the surface does not require user input. —end note]
17 Standalone functions

17.1 Standalone functions synopsis

```cpp
namespace std::experimental::io2d::v1 {
    template <class GraphicsSurfaces>
    basic_image_surface<GraphicsSurfaces> copy_surface(
        basic_image_surface<GraphicsSurfaces>& sfc) noexcept;
    template <class GraphicsSurfaces>
    basic_output_surface<GraphicsSurfaces> copy_surface(
        basic_output_surface<GraphicsSurfaces>& sfc) noexcept;
    template <class T>
    constexpr T degrees_to_radians(T d) noexcept;
    template <class T>
    constexpr T radians_to_degrees(T r) noexcept;
    float angle_for_point(point_2d ctr, point_2d pt) noexcept;
    point_2d point_for_angle(float ang, float rad = 1.0f) noexcept;
    point_2d point_for_angle(float ang, point_2d rad) noexcept;
    point_2d arc_start(point_2d ctr, float sang, point_2d rad,
        const matrix_2d& m = matrix_2d{}) noexcept;
    point_2d arc_center(point_2d cpt, float sang, point_2d rad,
        const matrix_2d& m = matrix_2d{}) noexcept;
    point_2d arc_end(point_2d cpt, float eang, point_2d rad,
        const matrix_2d& m = matrix_2d{}) noexcept;
}
```

17.2 copy_surface

```cpp
template <class GraphicsSurfaces>
basic_image_surface<GraphicsSurfaces> copy_surface(
    basic_image_surface<GraphicsSurfaces>& sfc) noexcept;
```

Returns: `GraphicsSurfaces::surfaces::copy_surface(sfc)`.

17.3 degrees_to_radians

```cpp
constexpr T degrees_to_radians(T d) noexcept;
```

Returns: If `d` is positive and is less than one thousandth of a degree, then `static_cast<T>(0)`. If `d` is negative and is less than one thousandth of a degree, then `-static_cast<T>(0)`. Otherwise, the value obtained from converting the degrees value `d` to radians.

Remarks: This function shall not participate in overload resolution unless `T` is a floating-point type.

17.4 radians_to_degrees

```cpp
constexpr T radians_to_degrees(T r) noexcept;
```

Returns: If `r` is positive and is less than one thousandth of a degree in radians, then `static_cast<T>(0)`. If `r` is negative and is less than one thousandth of a degree in radians, then `-static_cast<T>(0)`. Otherwise, the value obtained from converting the radians value `r` to degrees.

Remarks: This function shall not participate in overload resolution unless `T` is a floating-point type.

17.5 angle_for_point

```cpp
float angle_for_point(point_2d ctr, point_2d pt) noexcept;
```

Returns: The angle, in radians, of `pt` as a point on a circle with a center at `ctr`. If the angle is less that `pi<float> / 180000.0f`, returns `0.0f`.

§ 17.5
17.6 \textit{point\_for\_angle} \hfill [io2d.standalone.pointforangle]

\begin{verbatim}
point_2d point_for_angle(float ang, float rad = 1.0f) noexcept;
point_2d point_for_angle(float ang, point_2d rad) noexcept;
\end{verbatim}

\textbf{Requires:} If it is a \textit{float}, \textit{rad} is greater than 0.0f. If it is a \textit{point\_2d}, \textit{rad}.\textit{x} or \textit{rad}.\textit{y} is greater than 0.0f and neither is less than 0.0f.

\textbf{Returns:} The result of rotating the point \textit{point\_2d\{ 1.0f, 0.0f \}}, around an origin of \textit{point\_2d\{ 0.0f, 0.0f \}} by \textit{ang} radians, with a positive value of \textit{ang} meaning counterclockwise rotation and a negative value meaning clockwise rotation, with the result being multiplied by \textit{rad}.

17.7 \textit{arc\_start} \hfill [io2d.standalone.arcstart]

\begin{verbatim}
point_2d arc_start(point_2d ctr, float sang, point_2d rad,
                   const matrix_2d& m = matrix_2d{}) noexcept;
\end{verbatim}

\textbf{Requires:} \textit{rad}.\textit{x} and \textit{rad}.\textit{y} are both greater than 0.0f.

\textbf{Returns:} As-if:
\begin{verbatim}
auto lmtx = m;
lmtx.m20 = 0.0f; lmtx.m21 = 0.0f;
auto pt = point_for_angle(sang, rad);
return ctr + pt * lmtx;
\end{verbatim}

\textbf{Note:} Among other things, this function is useful for determining the point at which a new figure should begin if the first item in the figure is an arc and the user wishes to clearly define its center.

--- end note ---

17.8 \textit{arc\_center} \hfill [io2d.standalone.arccenter]

\begin{verbatim}
point_2d arc_center(point_2d cpt, float sang, point_2d rad,
                    const matrix_2d& m = matrix_2d{}) noexcept;
\end{verbatim}

\textbf{Requires:} \textit{rad}.\textit{x} and \textit{rad}.\textit{y} are both greater than 0.0f.

\textbf{Returns:} As-if:
\begin{verbatim}
auto lmtx = m;
lmtx.m20 = 0.0f; lmtx.m21 = 0.0f;
auto centerOffset = point_for_angle(two_pi<float> - sang, rad);
centerOffset.y = -centerOffset.y;
return cpt - centerOffset * lmtx;
\end{verbatim}

17.9 \textit{arc\_end} \hfill [io2d.standalone.arcend]

\begin{verbatim}
point_2d arc_end(point_2d cpt, float eang, point_2d rad,
                 const matrix_2d& m = matrix_2d{}) noexcept;
\end{verbatim}

\textbf{Requires:} \textit{rad}.\textit{x} and \textit{rad}.\textit{y} are both greater than 0.0f.

\textbf{Returns:} As-if:
\begin{verbatim}
auto lmtx = m;
auto tfrm = matrix_2d::init_rotate(eang);
lmtx.m20 = 0.0f; lmtx.m21 = 0.0f;
auto pt = (rad * tfrm);
pt.y = -pt.y;
return cpt + pt * lmtx;
\end{verbatim}
Annex A  (informative)
Bibliography

1 The following is a list of informative resources intended to assist in the understanding or use of this Technical Specification.


Index

2D graphics
  synopsis, 12–18

additive color, 7
aliasing, 10
alpha channel, 6
anti-aliasing, 10
aspect ratio, 6

Bézier curve, 8
  cubic, 8
  quadratic, 8

C
  Unicode TR, 5
channel, 6
closed figure, 8
color
  transparent black, 137
color channel, 6
color model, 7
  RGB, 7
  RGBA, 7
color space, 7
  sRGB, 7
command, 8, 9
compose, 9
composition algorithm, 9
control point, 8
CSS Colors Specification, 5
current point, 8
definitions, 6–10
degenerate figure, 9
degenerate segment, 8
degenerate segment, 8
  end point, 7
figure, 9
figure item, 9
filter, 9
final segment, 8
format
  JPEG, 5
  PNG, 5
  TIFF, 5
graphics state data, 9
initial segment, 8
integral point, 6
new figure point, 8
normalize, 6
open figure, 8
origin, 6
output device, 160
path, 9
path command, 9
path transformation matrix, 9
pixel, 7
point, 6
premultiplied format, 7
raster graphics data, 7
references
  normative, 5
render, 9
rendering and composing operation, 9
sample, 9
scope, 4
segment, 8
standard coordinate space, 6
start point, 7
surface, 157
surface class templates, 157
vector graphics data, 7
visual data, 6
visual data element, 6
visual data format, 7

Bibliography
Index of library names

a
  rgba_color, 24, 25
  abs_cubic_curve, 100
    constructor, 101
    control_pt1, 101
    control_pt2, 101
    data, 101
    end_pt, 101
    operator==, 102
  abs_line, 104
    constructor, 105
    data, 105
    operator==, 105
    to, 105
  abs_matrix, 96
    constructor, 97
    data, 97
    matrix, 97
    operator==, 97
  abs_new_figure, 92
    at, 93
    constructor, 92, 93
    data, 93
    operator==, 93
  abs_quadratic_curve, 107
    constructor, 108
    control_pt, 108
    data, 108
    end_pt, 108
    operator==, 108
  angle_for_point, 179
  angular_direction
    basic_point_2d, 74
  antialias, 132
  antialiasing
    basic_fill_props, 151
  arc, 110
    basic_path_builder, 120
    center, 112
    constructor, 111, 112
    data, 112
    operator==, 113
    radius, 112
    rotation, 112
    start_angle, 112
  arc_center, 180
  arc_end, 180
  arc_start, 180
  at
    abs_new_figure, 93
    rel_new_figure, 94, 95
  auto_clear
    basic_output_surface, 173

b
  rgba_color, 24
  basic_bounding_box, 81
    bottom_right, 83
    constructor, 82
    data, 82
    height, 83
    operator==, 83
    top_left, 83
    width, 83
    x, 82, 83
    y, 83
  basic_brush, 128
    constructor, 130, 131
    data, 131, 153
    type, 131
  basic_brush_props, 145
    brush_matrix, 145, 146
    constructor, 145
    fill_rule, 145, 146
    filter, 145, 146
    wrap_mode, 145, 146
  basic_circle, 83
    center, 84
    constructor, 84
    data, 84
    operator==, 85
    radius, 84
  basic_clip_props
    clip, 148
    constructor, 147
    data, 148
    fill_rule, 148
  basic_dashes, 152
    constructor, 153
  basic_display_point
    basic_point_2d, 80
    constructor, 80, 81
    data, 81
    operator==, 81
    x, 81
    y, 81
  basic_fill_props
    antialiasing, 151
    constructor, 151
    data, 151
    fill_rule, 151
    filtering, 151
  basic_image_surface, 165
    clear, 167
    constructor, 166
    dimensions, 167
    fill, 168
data, 144
filtering, 144
surface_matrix, 144
surface_member, 144
basic_stroke_props
  constructor, 149
  line_cap, 149, 150
  line_join, 149, 150
  line_width, 149, 150
  max_miter_limit, 150
  miter_limit, 150
basic_unmanaged_output_surface, 174
  auto_clear, 176, 177
  clear, 176
  constructor, 175
  dimensions, 175, 177
  display_dimensions, 175, 177
  draw_callback, 177
  fill, 176
  format, 175
  has_draw_callback, 175
  has_size_change_callback, 175
  letterbox_brush, 176, 177
  letterbox_brush_props, 176
  mask, 177
  max_dimensions, 175
  max_display_dimensions, 176
  paint, 176
  scaling, 176, 177
  size_change_callback, 177
  stroke, 176
begin
  basic_path_builder, 121
begin_show
  basic_output_surface, 171
bottom_right
  basic_bounding_box, 83
brush_matrix
  basic_brush_props, 145, 146
capacity
  basic_path_builder, 118
cbegin
  basic_path_builder, 121
cend
  basic_path_builder, 121
center
  arc, 112
  basic_circle, 84
clear
  basic_image_surface, 167
  basic_output_surface, 171
  basic_unmanaged_output_surface, 176
close_figure, 95
  basic_path_builder, 119
  operator==, 95
color
  gradient_stop, 128
compositing
  basic_render_props, 144
control_pt
  abs_quadratic_curve, 108
  rel_quadratic_curve, 110
control_pt1
  abs_cubic_curve, 101
  rel_cubic_curve, 103
control_pt2
  abs_cubic_curve, 101
  rel_cubic_curve, 103
copy_surface, 179
crbegin
  basic_path_builder, 121
crend
  basic_path_builder, 121
cubic_curve
  basic_path_builder, 120
data
  abs_cubic_curve, 101
  abs_line, 105
  abs_matrix, 97
  abs_new_figure, 93
  abs_quadratic_curve, 108
  arc, 112
  basic_bounding_box, 82
  basic_brush, 131, 153
  basic_circle, 84
  basic_clip_props, 148
  basic_display_point, 81
  basic_fill_props, 151
  basic_matrix_2d, 77
  basic_point_2d, 74
  basic_render_props, 144
  basic_unmanaged_output_surface, 177
determinant
  basic_matrix_2d, 79
dimensions
  basic_image_surface, 167
  basic_output_surface, 173
  basic_unmanaged_output_surface, 173
  display_dimensions
  basic_output_surface, 173
  basic_unmanaged_output_surface, 173
dot
  basic_point_2d, 74
draw_callback
  basic_output_surface, 172
  basic_unmanaged_output_surface, 177
emplace_back
  basic_path_builder, 120
end
basic_path_builder, 121
eend_pt
abs_cubic_curve, 101
abs_quadratic_curve, 108
rel_cubic_curve, 103, 104
rel_quadratic_curve, 110
end_show
basic_output_surface, 171
erase
basic_path_builder, 120
<experimental/io2d>, 12
fill
basic_image_surface, 168
basic_output_surface, 172
basic_unmanaged_output_surface, 176
fill_rule
basic_brush_props, 145, 146
basic_clip_props, 148
basic_fill_props, 151
filter
basic_brush_props, 145, 146
basic_mask_props, 152
filtering
basic_fill_props, 151
basic_render_props, 144
format
basic_image_surface, 167
basic_output_surface, 173
basic_unmanaged_output_surface, 175
g
rgba_color, 23, 24
gradient_stop, 127
color, 128
constructor, 128
offset, 128
operator==, 128
has_draw_callback
basic_unmanaged_output_surface, 175
has_size_change_callback
basic_unmanaged_output_surface, 175
height
basic_bounding_box, 83
image_file_format, 156
init_reflect
basic_matrix_2d, 77
init_rotate
basic_matrix_2d, 77
init_scale
basic_matrix_2d, 77
init_shear_y
basic_matrix_2d, 77, 78
init_translate
basic_matrix_2d, 77
insert
basic_path_builder, 120
interpreted_path
constructor, 115
inverse
basic_matrix_2d, 79
is_finite
basic_matrix_2d, 79
is_invertible
basic_matrix_2d, 79
letterbox_brush
basic_output_surface, 173
basic_unmanaged_output_surface, 176, 177
letterbox_brush_props
basic_output_surface, 173
basic_unmanaged_output_surface, 176
line
basic_path_builder, 120
line_cap
basic_stroke_props, 149, 150
line_join
basic_stroke_props, 149, 150
line_width
basic_stroke_props, 149, 150
m00
basic_matrix_2d, 78
m01
basic_matrix_2d, 78, 79
m10
basic_matrix_2d, 78, 79
m11
basic_matrix_2d, 78, 79
m20
basic_matrix_2d, 78, 79
m21
basic_matrix_2d, 78, 79
magnitude
basic_point_2d, 74
magnitude_squared
basic_point_2d, 74
mask
basic_image_surface, 168
basic_output_surface, 172
basic_unmanaged_output_surface, 177
mask_matrix
basic_mask_props, 152
matrix
abs_matrix, 97
rel_matrix, 99
max_dimensions
basic_image_surface, 167
basic_output_surface, 173
basic_unmanaged_output_surface, 175
max_display_dimensions
basic_unmanaged_output_surface, 176
max_miter_limit
basic_stroke_props, 150
max_output_dimensions
basic_output_surface, 173
constructor, 23
    g, 23, 24
operator*, 28
operator**, 27, 28
operator==, 28
r, 23, 24
rotate
    basic_matrix_2d, 78
rotation
    arc, 112
save
    basic_image_surface, 167
scale
    basic_matrix_2d, 78
scaling
    basic_output_surface, 173
    basic_unmanaged_output_surface, 176, 177
set_matrix
    basic_path_builder, 119
shear_x
    basic_matrix_2d, 78
shear_y
    basic_matrix_2d, 78
shrink_to_fit
    basic_path_builder, 118
size_change_callback
    basic_output_surface, 172
    basic_unmanaged_output_surface, 177
start_angle
    arc, 112
stroke
    basic_image_surface, 168
    basic_output_surface, 172
    basic_unmanaged_output_surface, 176
surface, 157
surface_matrix
    basic_render_props, 144
surface_member
    basic_render_props, 144
swap
    basic_path_builder, 119, 121
to
    abs_line, 105
    rel_line, 106, 107
to_unit
    basic_point_2d, 74
top_left
    basic_bounding_box, 83
transform_pt
    basic_matrix_2d, 79
translate
    basic_matrix_2d, 78
user_scaling_callback
    basic_output_surface, 173
width

Bibliography 188
Index of implementation-defined behavior

The entries in this section are rough descriptions; exact specifications are at the indicated page in the general text.

additional data formats, 37
additional read only data formats, 37
additional visual data formats, 38
additional write only data formats, 38
antialiasing
    antialiasing
        best, 132
        fast, 132
        good, 132
basic_image_surface
    fill, 168
    mask, 168
    paint, 168
    save, 167
    stroke, 168
basic_output_surface
    begin_show, 171
    begin_show return value, 171
    fill, 172
    paint, 172
    stroke, 172
basic_unmanaged_output_surface
    fill, 177
    mask, 177
    paint, 176
    stroke, 176
create_unmanaged_output_surface, 70
dash pattern
    offset value, 160
display_surface
    unsupported draw format, 162
filter
    best, 127
    fast, 127
    good, 127
format::argb32, 140
format::xrgb16, 140
format::xrgb32, 140
implementation system_error-derived type, 11
max miter limit, 51
output device, 160
output device visual format support, 59, 62

Bibliography