Document Number: P0267R8
Date: 2018-06-26
Revises: P0267R7
Reply to: Michael B. McLaughlin
mikebmcl@gmail.com

Herb Sutter
Microsoft Inc.
hsutter@microsoft.com

Jason Zink
jzink_1@yahoo.com

Guy Davidson
guy@creative-assembly.com

Michael Kazakov
mike.kazakov@gmail.com
Audience:

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

List of Tables

List of Figures

0 Revision history [io2d.revisionhistory]

0.1 Revision 8 [io2d.revisionhistory.r8]

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 , 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_props`, `basic_clip_props`, `basic_stroke_props`, and `basic_mask_props` class templates to it.

8 Added 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 have 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.

22 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.

23 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.

24 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.

25 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.

26 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.

27 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.

28 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.).

29 `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.

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

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

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

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

0.2 Revision 7 [io2d.revisionhistory.r7]

1 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.

2 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.

3 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.

4 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

5 The modified classes are as follows

Table 1 — Class identifiers modified since R6

R6 identifier	R7 identifier
`vector_2d`	`basic_point_2d`
`matrix_2d`	`basic_matrix_2d`
`rectangle`	`basic_bounding_box`
`circle`	`basic_circle`
`path_group`	`basic_interpreted_path`
`path_builder`	`basic_path_builder`
`color_stop`	`gradient_stop`

R6 identifier	R7 identifier
`brush`	`basic_brush`
`render_props`	`basic_render_props`
`brush_props`	`basic_brush_props`
`clip_props`	`basic_clip_props`
`stroke_props`	`basic_stroke_props`
`mask_props`	`basic_mask_props`
`image_surface`	`basic_image_surface`
`display_surface`	`basic_output_surface`

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 [io2d.scope]

1 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.

2 Normative references [io2d.refs]

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*

(1.10) — ISO 32000-1:2008, *Document management — Portable document format — Part 1: PDF 1.7*

(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*. 4 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 [io2d.defns]

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:

- IEC Electropedia: available at http://www.electropedia.org/
- 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 **[io2d.defns.point]**
point
coordinate designated by a floating-point x axis value and a floating-point y axis value

3.2 **[io2d.defns.origin]**
origin
point with an x axis value of 0 and a y axis value of 0

3.3 **[io2d.defns.stndcrdspace]**
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 **[io2d.defns.point.integral]**
integral point
point where the x axis value and the y axis value are integers

3.5 **[io2d.defns.normalize]**
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 **[io2d.defns.aspectratio]**
aspect ratio
ratio of the width to the height of a rectangular area

3.7 **[io2d.defns.visdata]**
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 **[io2d.defns.visdataelem]**
visual data element
unit of visual data at a specific point

3.9 **[io2d.defns.channel]**
channel
component of visual data

3.10 **[io2d.defns.colorchannel]**
color channel
channel that only represents the intensity of a specific color

3.11 [io2d.defns.alphachannel]

alpha channel

channel that only represents transparency

3.12 [io2d.defns.visdatafmt]

visual data format

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

3.13 [io2d.defns.premultipliedformat]

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 [io2d.defns.rastergfxdata]

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 [io2d.defns.pixel]

pixel

discrete element of raster graphics data

3.16 [io2d.defns.vectorgfxdata]

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 [io2d.defns.colormodel]

color model

ideal, mathematical representation of color

3.18 [io2d.defns.additivecolor]

additive color

color defined by the emissive intensity of its color channels

3.19 [io2d.defns.rgbcolormodel]

RGB color model

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

3.20 [io2d.defns.rgbacolormodel]

RGBA color model

RGB color model with an alpha channel

3.21 [io2d.defns.colorspace]

color space

systematic mapping of values to colorimetric colors

3.22 [io2d.defns.srgbcolorspace]

sRGB color space

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

3.23 [io2d.defns.startpt]

start point

point that begins a segment

3.24 **[io2d.defns.endpt]**
end point
point that ends a segment

3.25 **[io2d.defns.controlpt]**
control point
point, other than the start point and the end point, that is used in defining a curve

3.26 **[io2d.defns.bezier.quadratic]**
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 **[io2d.defns.bezier.cubic]**
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 **[io2d.defns.seg]**
segment
line, Bézier curve, or arc

3.29 **[io2d.defns.initialseg]**
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 **[io2d.defns.newfigpt]**
new figure point
point that is the start point of the initial segment

3.31 **[io2d.defns.finalseg]**
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 **[io2d.defns.currentpt]**
current point
point used as the start point of a segment

3.33 **[io2d.defns.openfigure]**
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 **[io2d.defns.closedfigure]**
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 **[io2d.defns.degenerateseg]**
degenerate segment
segment that has the same values for its start point, end point, and, if any, control points

3.36 **[io2d.defns.command.closefig]**
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 **[io2d.defns.command.newfig]**
command
⟨new figure command⟩ an instruction that creates a new path

3.38 **[io2d.defns.figitem]**
figure item
segment, new figure command, close figure command, or path command

3.39 **[io2d.defns.figure]**
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 **[io2d.defns.path]**
path
collection of figures

3.41 **[io2d.defns.pathtransform]**
path transformation matrix
affine transformation matrix used to apply affine transformations to the points in a path

3.42 **[io2d.defns.pathcommand]**
path command
instruction that modifies the path transformation matrix

3.43 **[io2d.defns.degenfigure]**
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 **[io2d.defns.graphicsstatedata]**
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 **[io2d.defns.render]**
render
transform a path into visual data

3.46 **[io2d.defns.compositionalgorithm]**
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 **[io2d.defns.compose]**
compose
apply a composition algorithm

3.48 **[io2d.defns.renderingandcomposingop]**
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 **[io2d.defns.filter]**
filter
algorithm that determines a color value from a raster graphics data source for a non-integral point

3.50 [io2d.defns.sample]
sample
apply a filter

3.51 [io2d.defns.alias]
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 [io2d.defns.antialias]
anti-aliasing
application of an algorithm while rendering to reduce aliasing

4 Error reporting [io2d.err.report]

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.

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;
```

```
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 GraphicsMath>
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>::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;
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;
```

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

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

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>;
```

```
    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 [io2d.colors]

6.1 Introduction to color [io2d.colors.intro]

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

2 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.

3 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.

4 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.

5 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.

6 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.

7 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.

8 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 [io2d.colors.reqs]

1 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` [io2d.rgbacolor]

6.3.1 `rgba_color` overview [io2d.rgbacolor.intro]

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

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

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

4 Legal values for each channel are in the range `[0.0f, 1.0f]`.

6.3.2 `rgba_color` synopsis [io2d.rgbacolor.synopsis]

```
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_khaki;
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 lavender_blush;
static const rgba_color lawn_green;
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_grey;
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 [io2d.rgbacolor.cons]

```
constexpr rgba_color() noexcept;
```

1 *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;
```

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

3 *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` .

4 *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;
```

5 *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`.

6 *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`.

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

6.3.4 `rgba_color` modifiers [io2d.rgbacolor.modifiers]

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

1 *Requires:* `val >= 0` and `val <= 255`.

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

3 *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;
```

4 *Requires:* `val >= 0.0f` and `val <= 1.0f`.

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

6 *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;
```

7 *Requires:* `val >= 0` and `val <= 255`.

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

9 *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;
```

10 *Requires:* `val >= 0.0f` and `val <= 1.0f`.

11 *Effects:* The green channel is `val * a()`.

12 *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;
```

13 *Requires:* `val >= 0` and `val <= 255`.

14 *Effects:* The blue channel is `val / 255.0F * a()`.

15 *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;
```

16 *Requires:* `val >= 0.0f` and `val <= 1.0f`.

17 *Effects:* The blue channel is `val * a()`.

18 *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;
```

19 *Requires:* `val >= 0` and `val <= 255`.

20 *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`.

21 *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;
```

22 *Requires:* `val >= 0.0f` and `val <= 1.0f`.

23 *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`.

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

6.3.5 `rgba_color` observers [io2d.rgbacolor.observers]

```
constexpr float r() const noexcept;
```

1 *Returns:* The red channel.

```
constexpr float g() const noexcept;
```

2 *Returns:* The green channel.

```
constexpr float b() const noexcept;
```

3 *Returns:* The blue channel.

`constexpr float a() const noexcept;`

4 *Returns:* The alpha channel.

6.3.6 `rgba_color` static members [io2d.rgbacolor.statics]

1 The `alpha` value of all of the predefined `rgba_color` static member object 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

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

Table 2 — `rgba_color` static members values (continued)

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

Table 2 — `rgba_color` static members values (continued)

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

6.3.7 `rgba_color` operators [io2d.rgbacolor.ops]

```
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))`.

5 `a(min(a() * rhs / 255.0F, 1.0F))`.

Returns: `*this`.

6 *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;
```

7 *Requires:* `rhs >= 0.0F` and `rhs <= 1.0F`.

8 *Effects:* `r(min(r() * rhs, 1.0F))`.

9 `g(min(g() * rhs, 1.0F))`.

10 `b(min(b() * rhs, 1.0F))`.

11 `a(min(a() * rhs, 1.0F))`.

Returns: `*this`.

12 *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;
```

13 *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;
```

14 *Requires:* `rhs >= 0` and `rhs <= 255`.

15 *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))
```

16 *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;
```

17 *Requires:* `rhs >= 0.0F` and `rhs <= 1.0F`.

18 *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))
```

19 *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;
```

20 *Requires:* `lhs >= 0` and `lhs <= 255`.

21 *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))
```

22 *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;
```

23 *Requires:* `lhs >= 0.0F` and `lhs <= 1.0F`.

24 *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))
```

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

7 Graphics math [io2d.graphmath]

7.1 General [io2d.graphmath.general]

1 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*]

2 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

Subclause		Header(s)
7.2	GraphicsMath traits	
Clause 9	Linear algebra classes	`<experimental/io2d>`
Clause 10	Geometry classes	`<experimental/io2d>`

7.2 Requirements [io2d.graphmath.reqs]

1 This subsection defines requirements on *GraphicsMath* types.

2 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.

3 Let `X` be a GraphicsMath type.

4 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

typedef-name	Class data
`bounding_box_data_type`	`basic_bounding_box<X>`
`circle_data_type`	`basic_circle<X>`
`display_point_data_type`	`basic_display_point<X>`
`matrix_2d_data_type`	`basic_matrix_2d<X>`
`point_2d_data_type`	`basic_point_2d<X>`

5 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

Type	Member data	Member data type
`bounding_box_data_type`	`x`	`float`
`bounding_box_data_type`	`y`	`float`
`bounding_box_data_type`	`w`	`float`
`bounding_box_data_type`	`h`	`float`
`circle_data_type`	`x`	`float`
`circle_data_type`	`y`	`float`
`circle_data_type`	`r`	`float`
`display_point_data_type`	`x`	`int`
`display_point_data_type`	`y`	`int`

Type	Member data	Member data type
matrix_2d_data_type	m00	float
matrix_2d_data_type	m01	float
matrix_2d_data_type	m02	float
matrix_2d_data_type	m10	float
matrix_2d_data_type	m11	float
matrix_2d_data_type	m12	float
matrix_2d_data_type	m20	float
matrix_2d_data_type	m21	float
matrix_2d_data_type	m22	float
point_2d_data_type	x	float
point_2d_data_type	y	float

6 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.

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

Table 6 — GraphicsMath requirements

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
X::create_point_2d()	P	Equivalent to: create_point_2d(0.0f, 0.0f);	
X::create_point_2d(float x, float y)	P	Returns an object p.	*Postconditions:* p.x == x and p.y == y.
X::x(P& p, float x)	void		*Postconditions:* p.x == x.
X::y(P& p, float y)	void		*Postconditions:* p.y == y.
X::x(const P& p)	float	Returns p.x.	
X::y(const P& p)	float	Returns p.y.	
X::dot(const P& p1, const P& p2)	float	Returns p1.x * p2.x + p1.y * p2.y.	
X::magnitude(const P& p)	float	Returns sqrt(p.x * p.x + p.y * p.y).	
X::magnitude_squared(const P& p)	float	Returns p.x * p.x + p.y * p.y.	
X::angular_direction(const P& p)	float	Returns atan2(p.y, p.x) < 0.0f ? atan2(p.y, p.x) + two_pi<float> : atan2(p.y, p.x).	*Remarks:* The purpose of adding two_pi<float> if the result is negative is to produce values in the range [0.0f, two_pi<float>)
X::to_unit(const P& p)	P	Returns an object r.	*Postconditions:* r.x == p.x / magnitude(p) and r.y == p.y / magnitude(p).

Table 6 — `GraphicsMath` requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::add(const P& p1, const P& p2)`	`P`	Returns an object `r`.	*Postconditions:* `r.x == p1.x + p2.x` and `r.y == p1.y + p2.y`.
`X::add(const P& p, float f)`	`P`	Returns an object `r`.	*Postconditions:* `r.x == p.x + f` and `r.y == p.y + f`.
`X::add(float f, const P& p)`	`P`	Returns an object `r`.	*Postconditions:* `r.x == f + p.x` and `r.y == f + p.y`.
`X::subtract(const P& p1, const P& p2)`	`P`	Returns an object `r`.	*Postconditions:* `r.x == p1.x - p2.x` and `r.y == p1.y - p2.y`.
`X::subtract(const P& p, float f)`	`P`	Returns an object `r`.	*Postconditions:* `r.x == p.x - f` and `r.y == p.y - f`.
`X::subtract(float f, const P& p)`	`P`	Returns an object `r`.	*Postconditions:* `r.x == f - p.x` and `r.y == f - p.y`.
`X::multiply(const P& p1, const P& p2)`	`P`	Returns an object `r`.	*Postconditions:* `r.x == p1.x * p2.x` and `r.y == p1.y * p2.y`.
`X::multiply(const P& p, float f)`	`P`	Returns an object `r`.	*Postconditions:* `r.x == p.x * f` and `r.y == p.y * f`.
`X::multiply(float f, const P& p)`	`P`	Returns an object `r`.	*Postconditions:* `r.x == f * p.x` and `r.y == f * p.y`.
`X::divide(const P& p1, const P& p2)`	`P`	Returns an object `r`.	*Postconditions:* `r.x == p1.x / p2.x` and `r.y == p1.y / p2.y`.
`X::divide(const P& p, float f)`	`P`	Returns an object `r`.	*Postconditions:* `r.x == p.x / f` and `r.y == p.y / f`.
`X::divide(float f, const P& p)`	`P`	Returns an object `r`.	*Postconditions:* `r.x == f / p.x` and `r.y == f / p.y`.
`X::equal(const P& l, const P& r)`	`bool`	Returns `l.x == r.x && l.y == r.y`	

Table 6 — `GraphicsMath` requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::not_equal(const P& l, const P& r)`	`bool`	Equivalent to `return !equal(l, r);`	
`X::negate(const P& p)`	`P`	Returns `r`, where `r.x == -p.x` and `r.y == -p.y`	
`X::create_matrix_2d()`	`M`	Equivalent to `return create_matrix_2d(1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f);`	
`X::create_matrix_2d(float m00, float m01, float m10, float m11, float m20, float m21)`	`M`	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`
`X::create_translate(const P& p)`	`M`	Equivalent to `return X::create_matrix_2d(1.0f, 0.0f, 0.0f, 1.0f, p.x, p.y);`	
`X::create_scale(const P& p)`	`M`	Equivalent to `return X::create_matrix_2d(p.x, 0.0f, 0.0f, p.y, 0.0f, 0.0f);`	
`X::create_rotate(float r)`	`M`	Equivalent to `return X::create_matrix_2d(cos(r), sin(r), sin(r), -cos(r), 0.0f, 0.0f);`	
`X::create_rotate(float r, const P& p)`	`M`	Equivalent to `return multiply(multiply(create_translate(p), create_rotate(r)), create_translate(negate(p)));`	
`X::create_reflect(float r)`	`M`	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;`	
`X::create_shear_x(float f)`	`M`	Equivalent to `return create_matrix_2d(1.0f, 0.0f, f, 1.0f, 0.0f, 0.0f);`	
`X::create_shear_y(float f)`	`M`	Equivalent to `return create_matrix_2d(1.0f, f, 0.0f, 1.0f, 0.0f, 0.0f);`	

Table 6 — `GraphicsMath` requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::multiply(const M& l, const M& r)`	`M`	Equivalent to `return create_matrix_2d(l.m00 * r.m00 + l.m01 * r.m10, l.m00 * r.m01 + l.m01 * r.m11, l.m10 * r.m00 + l.m11 * r.m10, l.m10 * r.m01 + l.m11 * r.m11, l.m20 * r.m00 + l.m21 * r.m10 + r.m20, l.m20 * r.m01 + l.m21 * r.m11 + r.m21 );`	
`X::translate(M& m, const P& p)`	`void`	Equivalent to `m = multiply(m, create_translate(p));`	
`X::scale(M& m, const P& p)`	`void`	Equivalent to `m = multiply(m, create_scale(p));`	
`X::rotate(M& m, float r)`	`void`	Equivalent to `m = multiply(m, create_rotate(r));`	
`X::rotate(M& m, float r, const P& p)`	`void`	Equivalent to `m = multiply(m, create_rotate(r, p));`	
`X::reflect(M& m, float r)`	`void`	Equivalent to `m = multiply(m, create_reflect(r));`	
`X::shear_x(M& m, float f)`	`void`	Equivalent to `m = multiply(m, create_shear_x(f));`	
`X::shear_y(M& m, float f)`	`void`	Equivalent to `m = multiply(m, create_shear_y(f));`	
`X::is_finite(const M& m)`	`bool`	Equivalent to `return isfinite(m.m00) && isfinite(m.m01) && isfinite(m.m10) && isfinite(m.m10) && isfinite(m.11) && isfinite(m.20) && isfinite(m.21);`	
`X::is_-invertible(const M& m)`	`bool`	Equivalent to `return (m.m00 * m.11 - m.m01 * m.10) != 0.0f;`	
`X::determinant(const M& m)`	`float`	Equivalent to `return m.m00 * m.11 - m.01 * m.10;`	
`X::inverse(const M& m)`	`M`	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.m10 * 1.0f - 0.0f * m.m20) * id, (m.m00 * 1.0f - 0.0f * m.m20) * id, (m.m10 * m.m21 - m.m11 * m.m20) * id, -(m.m00 * m.21 - m.m01 * m.m20) * id)`	

Table 6 — `GraphicsMath` requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::transform_pt(const M& m, const P& p)`	`P`	`return create_point_2d(m.m00 * p.x + m10 * p.y + m.m20, m.01 * p.x + m.m11 * p.y + m.21);`	
`X::equal(const M& l, const M& r)`	`bool`	Returns `l.m00 == r.m00 && l.m01 == r.m01 && l.m11 == r.m11 && l.m20 == r.m20 && l.m21 == r.m21`.	
`X::not_equal(const M& l, const M& r)`	`bool`	Equivalent to `return !equal(l, r);`	
`create_display_point()`	`D`	Equivalent to `return create_display_point(0, 0);`	
`create_display_point(int x, int y)`	`D`	Returns an object `d`.	*Postconditions:* `d.x == x` and `d.y == y`.
`X::x(D& d, int x)`	`void`		*Postconditions:* `d.x == x`.
`X::y(D& d, int y)`	`void`		*Postconditions:* `d.y == y`.
`X::x(const D&d)`	`int`	Returns `d.x`.	
`X::y(const D&d)`	`int`	Returns `d.y`.	
`X::equal(const D& l, const D& r)`	`bool`	Returns `l.x == r.x && l.y == r.y`.	
`X::not_equal(const D& l, const D& r)`	`bool`	Equivalent to `return !equal(l, r);`	
`X::create_bounding_box()`	`B`	Equivalent to `return create_bounding_box(0.0f, 0.0f, 0.0f, 0.0f);`	
`X::create_bounding_box(float x, float y, float w, float h)`	`B`	Returns an object `b`.	*Postconditions:* `b.x == x`, `b.y == y`, `b.w == w`, and `b.h == h`.
`X::create_bounding_box(const P& tl, const P& br)`	`B`	Equivalent to `return create_bounding_box(tl.x, tl.y, max(0.0f, br.x - tl.x), max(0.0f, br.y - tl.y));`	
`X::x(B& b, float x)`	`void`		*Postconditions:* `b.x == x`.
`X::y(B& b, float y)`	`void`		*Postconditions:* `b.y == y`.
`X::width(B& b, float w)`	`void`		*Postconditions:* `b.w == w`.
`X::height(B& b, float h)`	`void`		*Postconditions:* `b.h == h`.
`X::top_left(B& b, const P& p)`	`void`		*Postconditions:* `b.x == p.x` and `b.y == p.y`.

Table 6 — `GraphicsMath` requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::bottom_right(B& b, const P& p)`	`void`		*Postconditions:* `b.w == max(p.x - b.x, 0.0f)` and `b.h == max(p.y - b.y, 0.0f)`.
`X::x(const B& b)`	`float`	Returns `b.x`.	
`X::y(const B& b)`	`float`	Returns `b.y`.	
`X::width(const B& b)`	`float`	Returns `b.w`.	
`X::height(const B& b)`	`float`	Returns `b.h`.	
`X::top_left(const B& b)`	`P`	Returns an object `p`.	*Postconditions:* `p.x == b.x` and `p.y == b.y`.
`X::bottom_right(const B& b)`	`P`	Returns an object `p`.	*Postconditions:* `p.x == b.x + b.w` and `p.y == b.y + b.h`.
`X::equal(const B& l, const B& r)`	`bool`	Returns `l.x == r.x && l.y == r.y && l.w == r.w && l.h == r.h`.	
`X::not_equal(const B& l, const B& r)`	`bool`	Equivalent to `return !equal(l, r);`	
`X::create_circle()`	`C`	Equivalent to `return create_circle(create_point_2d(0.0f, 0.0f), 0.0f);`	
`X::create_circle(const P& p, float r)`	`C`	Returns an object `c`.	*Requires:* `r >= 0.0f`. *Postconditions:* `c.x == p.x`, `c.y == p.y`, and `c.r == r`.
`X::center(C& c, const P&p)`	`void`		*Postconditions:* `c.x == p.x` and `c.y == p.y`.
`X::radius(C& c, float r)`	`void`		*Requires:* `r >= 0.0f`. *Postconditions:* `c.r == r`.
`X::center(const C& c)`	`P`	Returns an object `p`.	*Postconditions:* `p.x == c.x` and `p.y == c.y`.
`X::radius(const C& c)`	`float`	Returns `c.r`.	
`X::equal(const C& l, const C& r)`	`bool`	Returns `l.x == r.x && l.y == r.y && l.r == r.r`.	
`X::not_equal(const C& l, const C& r)`	`bool`	Equivalent to `return !equal(l, r);`	

Table 6 — GraphicsMath requirements (continued)

<table>
<tr><th>Expression</th><th>Return type</th><th>Operational semantics</th><th>Assertion/note pre-/post-condition</th></tr>
<tr><td>X::point_for_-angle(float a, float m)</td><td>P</td><td>Returns transform_-pt(create_rotate(a), create_point_2d(m, 0.0f)).</td><td></td></tr>
<tr><td>X::point_for_-angle(float a, const P& r)</td><td>P</td><td>Returns multiply(transform_-pt(create_rotate(a), create_point_2d(1.0f, 0.0f)), r).</td><td></td></tr>
<tr><td>X::angle_for_-point(const P& c, const P& p)</td><td>float</td><td>Equivalent to: const float co = pi<float> / 180’000.0f;
auto a = atan2(-(p.y - c.y), p.x - c.x);
if (abs(a) < co || abs(a - two_pi<float>) < co) {
  return 0.0f;
}
if (a < 0.0f) {
  return
    a + two_pi<float>;
}
return a;</td><td></td></tr>
<tr><td>X::arc_start(const P& c, float sa, const P& r, const M& m)</td><td>P</td><td>Equivalent to: auto lm = m;
lm.m20 = 0.0f;
lm.m21 = 0.0f;
return add(c, transform_pt(lm, point_for_angle(sa, r)));</td><td></td></tr>
<tr><td>X::arc_center(const P& c, float sa, const P& r, const M& m)</td><td>P</td><td>Equivalent to: auto lm = m;
lm.m20 = 0.0f;
lm.m21 = 0.0f;
auto o = point_for_-angle(two_pi<float> - sa, r); o.y = -o.y;
return subtract(c, transform_pt(lm, o));</td><td></td></tr>
<tr><td>X::arc_end(const P& c, float ea, const P& r, const M& m)</td><td>P</td><td>Equivalent to: 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));</td><td></td></tr>
</table>

8 Graphics surfaces [io2d.graphsurf]

8.1 General [io2d.graphsurf.general]

1 This Clause defines requirements on *GraphicsSurfaces* types.

2 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 [io2d.graphsurf.reqs]

8.2.1 Classes [io2d.graphsurf.reqs.classes]

1 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).

2 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.

3 [*Example:*

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

— *end example*]

4 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 [io2d.graphsurf.reqs.addimgform]

1 Let `X` be a GraphicsSurfaces type.

2 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.*

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

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

1. `read_only`
2. `write_only`

5 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.*

6 The values of read only image format enumerators shall be in the range `[20000, 29999]`.

7 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*.

8 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 7 — `X::brushes` requirements

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::brushes::create_brush(const rgba_color& c)`	`brush_data_type`	Returns an object `b`.	*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*]

Table 7 — `X::brushes` requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`template <class InputIterator> create_brush(const basic_point_2d<M>& be, const basic_point_2d<M>& en, InputIterator first, InputIterator last)`	`brush_data_type`	Returns an object `b`.	*Postconditions:* `b` is a linear gradient brush, its begin point is `be`, its end point is `en`, its gradient stops are formed using the sequential series of `gradient_stop` objects beginning at `first` and ending at `last - 1` (See 13.2.2 and 13.2.4), and `X::brushes::get_brush_type(b) == brush_type::linear`.
`create_brush(const basic_point_2d<M>& be, const basic_point_2d<M>& en, initializer_list<gradient_stop> il)`	`brush_data_type`	Returns an object `b`.	*Postconditions:* `b` is a linear gradient brush, its begin point is `be`, its end point is `en`, its gradient stops are formed using the sequential series of `gradient_stop` objects in `il` (See 13.2.2 and 13.2.4), and `X::brushes::get_brush_type(b) == brush_type::linear`.

Table 7 — `X::brushes` requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`template <class InputIterator> create_brush(const basic_circle<M>& be, const basic_circle<M>& en, InputIterator first, InputIterator last)`	`brush_data_type`	Returns an object `b`.	*Postconditions:* `b` is a radial gradient brush, its start circle is `be`, its end circle is `en`, its gradient stops are formed using the sequential series of `gradient_stop` objects beginning at `first` and ending at `last - 1` (See 13.2.3 and 13.2.4), and `X::brushes::get_brush_type(b) == brush_type::radial`.
`create_brush(const basic_circle<M>& be, const basic_circle<M>& en, initializer_list<gradient_stop> il)`	`brush_data_type`	Returns an object `b`.	*Postconditions:* `b` is a radial gradient brush, its start circle is `be`, its end circle is `en`, its gradient stops are formed using the sequential series of `gradient_stop` objects in `il` (See 13.2.3 and 13.2.4), and `X::brushes::get_brush_type(b) == brush_type::radial`.
`create_brush(basic_image_surface<graphics_surfaces_type>&& i)`	`brush_data_type`	Returns an object `b`.	*Postconditions:* `b` is a surface brush, its visual data is the raster graphics data from `i`, and `X::brushes::get_brush_type(b) == brush_type::surface`

Table 7 — `X::brushes` requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`get_brush_type(const brush_data_type& data)`	`brush_type`	Returns the brush type of `data`.	

8.2.5 paths requirements [io2d.graphsurf.reqs.paths]

1 Paths are described in Clause 12.

2 Let `X` be a GraphicsSurfaces type.

3 Let `G` be `X::graphics_math_type`.

4 Table 10 describes the observable effects of the member functions of `X::paths`.

5 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

typedef-name	**Class data**
`abs_cubic_curve_data_type`	`basic_figure_items<X>::abs_cubic_curve`
`abs_line_data_type`	`basic_figure_items<X>::abs_line`
`abs_matrix_data_type`	`basic_figure_items<X>::abs_matrix`
`abs_new_figure_data_type`	`basic_figure_items<X>::abs_new_figure`
`abs_quadratic_curve_data_type`	`basic_figure_items<X>::abs_quadratic_curve`
`arc_data_type`	`basic_figure_items<X>::arc`
`close_figure_data_type`	`basic_figure_items<X>::close_data`
`interpreted_path_data_type`	`basic_interpreted_path<X>`
`rel_cubic_curve_data_type`	`basic_figure_items<X>::rel_cubic_curve`
`rel_line_data_type`	`basic_figure_items<X>::rel_line`
`rel_matrix_data_type`	`basic_figure_items<X>::rel_matrix`
`rel_new_figure_data_type`	`basic_figure_items<X>::rel_new_figure`
`rel_quadratic_curve_data_type`	`basic_figure_items<X>::rel_quadratic_curve`
`revert_matrix_data_type`	`basic_figure_items<X>::revert_matrix`

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`, `AM` 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`, `M` 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

Type	Member data	Member data type
`abs_cubic_curve_data_type`	`cpt1`	`basic_point_2d<G>`
`abs_cubic_curve_data_type`	`cpt2`	`basic_point_2d<G>`
`abs_cubic_curve_data_type`	`ept`	`basic_point_2d<G>`
`abs_line_data_type`	`pt`	`basic_point_2d<G>`
`abs_matrix_data_type`	`m`	`basic_matrix_2d<G>`
`abs_new_figure_data_type`	`pt`	`basic_point_2d<G>`
`abs_quadratic_curve_data_type`	`cpt`	`basic_point_2d<G>`
`abs_quadratic_curve_data_type`	`ept`	`basic_point_2d<G>`
`arc_data_type`	`rad`	`basic_point_2d<G>`
`arc_data_type`	`rot`	`float`
`arc_data_type`	`sa`	`float`
`rel_cubic_curve_data_type`	`cpt1`	`basic_point_2d<G>`
`rel_cubic_curve_data_type`	`cpt2`	`basic_point_2d<G>`
`rel_cubic_curve_data_type`	`ept`	`basic_point_2d<G>`
`rel_line_data_type`	`pt`	`basic_point_2d<G>`
`rel_matrix_data_type`	`m`	`basic_matrix_2d<G>`
`rel_new_figure_data_type`	`pt`	`basic_point_2d<G>`
`rel_quadratic_curve_data_type`	`cpt`	`basic_point_2d<G>`
`rel_quadratic_curve_data_type`	`ept`	`basic_point_2d<G>`

Table 10 — `X::paths` requirements

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

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

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::paths::create_abs_line(const P& p)`	`AL`	*Returns:* An object `al`.	*Postconditions:* `al.pt == p`.
`X::paths::to(AL& al, const P& p)`	`void`		*Postconditions:* `al.pt == p`.
`X::paths::to(const AL& al)`	`P`	*Returns:* `al.pt`.	
`X::paths::create_abs_matrix()`	`AM`	Equivalent to `return create_abs_matrix(M());`	
`X::paths::create_abs_matrix(const M& m)`	`AM`	*Returns:* An object `am`.	*Postconditions:* `am.m == m`.
`X::paths::matrix(AM& am, const M& m)`	`void`		*Postconditions:* `am.m == m`.
`X::paths::matrix(const AM& am)`	`M`	*Returns:* `am.m`.	
`X::paths::create_abs_new_figure()`	`AN`	*Effects:* Equivalent to `return create_abs_new_figure(P());`	
`X::paths::create_abs_new_figure(const P& p)`	`AN`	*Returns:* An object `an`.	*Postconditions:* `an.pt == p`.
`X::paths::at(AN& an, const P& p)`	`void`		*Postconditions:* `an.pt == p`.
`X::paths::at(const AN& an)`	`P`	*Returns:* `an.pt`.	
`X::paths::create_abs_quadratic_curve()`	`AQ`	Equivalent to `return create_abs_quadratic_curve(P(), P());`	
`X::paths::create_abs_quadratic_curve(const P& cpt, const P& ept)`	`AQ`	*Returns:* An object `aq`.	*Postconditions:* `aq.cpt == cpt` and `aq.ept == ept`.
`X::paths::control_pt(AQ& aq, const P& p)`	`void`		*Postconditions:* `aq.cpt == p`.
`X::paths::end_pt(AQ& aq, const P& p)`	`void`		*Postconditions:* `aq.ept == p`.
`X::paths::control_pt(const AQ& aq)`	`P`	*Returns:* `aq.cpt`.	
`X::paths::end_pt(const AQ& aq)`	`P`	*Returns:* `aq.ept`.	
`X::paths::create_arc()`	`ARC`	*Effects:* Equivalent to `return create_arc(P(), 0.0f, 0.0f);`	
`X::paths::create_arc(const P& rad, float rot, float sa)`	`ARC`	*Returns:* An object `arc`.	*Postconditions:* `arc.rad == rad`, `arc.rot == rot`, and `arc.sa == sa`.

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

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::paths::radius(ARC& arc, const P& rad)`	`void`		*Postconditions:* `arc.rad == rad`.
`X::paths::rotation(ARC& arc, float rot)`	`void`		*Postconditions:* `arc.rot == rot`.
`X::paths::start_angle(ARC& arc, float sa)`	`void`		*Postconditions:* `arc.sa == sa`.
`X::paths::radius(const ARC& arc)`	`P`	*Returns:* `arc.rad`.	
`X::paths::rotation(const ARC& arc)`	`float`	*Returns:* `arc.rot`.	
`X::paths::start_angle(const ARC& arc)`	`float`	*Returns:* `arc.sa`.	
`X::paths::center(const ARC& arc, const P& spt, const M& m)`	`P`	*Returns:* As-if: `auto lmtx = m; lmtx.m20 = 0.0f; lmtx.m21 = 0.0f; auto ctrOffset = point_for_angle<G>( two_pi<float> - arc.sa, arc.rad); ctrOffset.y(-ctrOffset.y); return spt - ctrOffset * lmtx;`	[*Note:* `spt` is the starting point of the arc. `m` is the transformation matrix being used. — *end note*]
`X::paths::end_pt(const ARC& arc, const P& spt, const M& m)`	`P`	*Returns:* As-if: `auto lmtx = m; lmtx.m20 = 0.0f; lmtx.m21 = 0.0f; auto tfrm = M::create_rotate(arc.sa + arc.rot); auto pt = arc.rad * tfrm; pt.y(-pt.y()); return spt + pt * lmtx;`	[*Note:* `spt` is the starting point of the arc. `m` is the transformation matrix being used. — *end note*]
`X::paths::create_interpreted_path()`	`IP`	*Returns:* An object `ip`.	*Postconditions:* `ip` has zero figures (See: 12.3.16)

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

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`template <class ForwardIterator> X::paths::create_interpreted_path<ForwardIterator first, ForwardIterator last`	`IP`	*Returns:* An object `ip`.	*Postconditions:* `ip` contains a zero or more figure items as determined by evaluating the sequence of `figure_items::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`.
`X::paths::create_rel_cubic_curve()`	`RC`	*Effects:* Equivalent to `return create_rel_cubic_curve(P(), P(), P());`	
`X::paths::create_rel_cubic_curve(const P& cpt1, const P& cpt2, const P& ept)`	`RC`	*Returns:* An object `rc`.	*Postconditions:* `rc.cpt1 == cpt1`, `rc.cpt2 == cpt2`, and `rc.ept == ept`.
`X::paths::control_pt1(RC& rc, const P& pt)`	`void`		*Postconditions:* `rc.cpt1 == pt`.
`X::paths::control_pt2(RC& rc, const P& pt)`	`void`		*Postconditions:* `rc.cpt2 == pt`.
`X::paths::end_pt(RC& rc, const P& pt)`	`void`		*Postconditions:* `rc.ept == pt`.
`X::paths::control_pt1(const RC& a)`	`P`	*Returns:* `rc.cpt1`.	
`X::paths::control_pt2(const RC& rc)`	`P`	*Returns:* `rc.cpt2`.	

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

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::paths::end_-pt(const RC& rc)`	`P`	*Returns:* `rc.ept`.	
`X::paths::create_-rel_line()`	`RL`	*Effects:* Equivalent to `return create_rel_line(P);`	
`X::paths::create_-rel_line(const P& p)`	`RL`	*Returns:* An object `rl`.	*Postconditions:* `rl.pt == p`.
`X::paths::to(RL& al, const P& p)`	`void`		*Postconditions:* `rl.pt == p`.
`X::paths::to(const RL& rl)`	`P`	*Returns:* `rl.pt`.	
`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:* `rm.m == m`.
`X::paths::matrix(RM& am, const M& m)`	`void`		*Postconditions:* `rm.m == m`.
`X::paths::matrix(const RM& rm)`	`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:* `rn.pt == p`.
`X::paths::at(RN& rn, const P& p)`	`void`		*Postconditions:* `rn.pt == p`.
`X::paths::at(const RN& rn)`	`P`	*Returns:* `rn.pt`.	
`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, const P& ept)`	`RQ`	*Returns:* An object `rq`.	*Postconditions:* `rq.cpt == cpt` and `rq.ept == ept`.
`X::paths::control_-pt(RQ& rq, const P& p)`	`void`		*Postconditions:* `rq.cpt == p`.
`X::paths::end_-pt(RQ& rq, const P& p)`	`void`		*Postconditions:* `rq.ept == p`.
`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 [io2d.graphsurf.reqs.surfstate]

1 Surface state data are described in Clause 14.

2 Let X be a GraphicsSurfaces type.

3 Let G be X::graphics_math_type.

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

5 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

typedef-name	Class data
render_props_data_type	basic_render_props
brush_props_data_type	basic_brush_props
clip_props_data_type	basic_clip_props
stroke_props_data_type	basic_stroke_props
mask_props_data_type	basic_mask_props
dashes_data_type	basic_dashes

6 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>.

7 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

Type	Member data	Member data type
render_props_data_type	fi	filter
render_props_data_type	m	M
render_props_data_type	c	compositing_op
brush_props_data_type	w	wrap_mode
brush_props_data_type	fi	filter
brush_props_data_type	m	M
clip_props_data_type	optional<IP>	c
clip_props_data_type	fr	fill_rule
stroke_props_data_type	lw	float
stroke_props_data_type	ml	float
stroke_props_data_type	lc	line_cap
stroke_props_data_type	lj	line_join
stroke_props_data_type	aa	antialias
fill_props_data_type	fr	fill_rule
fill_props_data_type	aa	antialias
mask_props_data_type	wm	wrap_mode
mask_props_data_type	fi	filter
mask_props_data_type	m	M
dashes_props_data_type	o	float
dashes_props_data_type	p	vector<float>

Table 13 — `X::surface_state_props` requirements

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::surface_state_props::create_render_props(filter f = filter::good, M m = M{}, compositing op c = compositing_op::over)`	`RE`	*Returns:* An object `r`.	*Postconditions:* `r.fi == f`, `r.m == m`, and `r.c == c`.
`X::surface_state_props::filtering(RA& r, filter fi)`	`void`		*Postconditions:* `r.fi == fi`.
`X::surface_state_props::surface_matrix(RA& r, const M& m)`	`void`		*Postconditions:* `ra.m == m`.
`X::surface_state_props::compositing(RA& r, compositing_op c)`	`void`		*Postconditions:* `ra.c == c`.
`X::surface_state_props::filtering(const RA& r)`	`filter`	*Returns:* `r.fi`.	
`X::surface_state_props::surface_matrix(const RA& r)`	`M`	*Returns:* `r.m`.	
`X::surface_state_props::compositing(const RA& r)`	`compositing_op`	*Returns:* `r.c`.	
`X::surface_state_props::create_brush_props(wrap_mode w = wrap_mode::none, filter fi = filter::good, const M& m = M{})`	`BR`	*Returns:* An object `b`.	*Postconditions:* `b.w == w`, `b.fi == fi`, and `b.m == m`.
`X::surface_state_props::brush_wrap_mode(BR& b, wrap_mode w)`	`void`		*Postconditions:* `b.w == w`.
`X::surface_state_props::brush_filter(BR& b, filter fi)`	`void`		*Postconditions:* `b.fi == fi`.
`X::surface_state_props::brush_matrix(BR& b, const M& m)`	`void`		*Postconditions:* `b.m == m`.
`X::surface_state_props::brush_wrap_mode(const BR& b)`	`wrap_mode`	*Returns:* `b.w`.	

Table 13 — `X::surface_state_props` requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::surface_state_props::brush_filter(const BR& b)`	`filter`	*Returns:* `b.fi`.	
`X::surface_state_props::brush_matrix(const BR& b)`	`M`	*Returns:* `b.m`.	
`X::surface_state_props::create_clip_props()`	`CL`	*Returns:* An object `c`.	*Postconditions:* `c.c == nullopt` and `c.fr == fill_rule::winding`.
`X::surface_state_props::create_clip_props(const BB& b, fill_rule fr)`	`CL`	*Returns:* An object `c`.	*Postconditions:* `c.c == clip(c, b)` and `c.fr == fr`.
`template <class Allocator> X::surface_state_props::create_clip_props(const basic_path_builder<X, Allocator>& pb, fill_rule fr)`	`CL`	*Returns:* An object `c`.	*Postconditions:* `c.c == IP(pb)` and `c.fr == fr`.
`template <class InputIterator> X::surface_state_props::create_clip_props(InputIterator first, InputIterator last, fill_rule fr)`	`CL`	*Returns:* An object `c`.	*Postconditions:* `c.c == IP(first, last)` and `c.fr == fr`.
`X::surface_state_props::create_clip_props(initializer_list<FI> il, fill_rule fr)`	`CL`	*Returns:* An object `c`.	*Postconditions:* `c.c == IP(il)` and `c.fr == fr`.
`X::surface_state_props::create_clip_props(const IP& ip, fill_rule fr)`	`CL`	*Returns:* An object `c`.	*Postconditions:* `c.c == ip` and `c.fr == fr`.
`X::surface_state_props::clip(CL& c, nullopt_t)`	`void`		*Postconditions:* `c.c == nullopt`.
`X::surface_state_props::clip(CL& c, const BB& b)`	`void`		*Postconditions:* `c.c == IP(b)`.

Table 13 — X::surface_state_props requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
template <class Allocator> X::surface_state_props::clip(CL& c, const basic_path_builder<X, Allocator>& pb)	void		*Postconditions:* c.c == IP(pb).
template <class InputIterator> X::surface_state_props::clip(CL& c, InputIterator first, InputIterator last)	void		*Postconditions:* c.c == IP(first, last).
X::surface_state_props::clip(CL& c, initializer_list<FI> il)	void		*Postconditions:* c.c == IP(il).
X::surface_state_props::clip(CL& c, const IP& ip)	void		*Postconditions:* c.c == ip.
X::surface_state_props::clip_fill_rule(CL&c, fill_rule fr)	void		*Postconditions:* c.fr == fr.
X::surface_state_props::clip(const CL& c)	IP	*Returns:* c.c.	
X::surface_state_props::clip_fill_rule(const CL& c)	fill_rule	*Returns:* c.fr.	
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)	ST	*Returns:* An object s.	*Requires:* lw >= 0.0f, ml >= 1.0f, and ml <= max_miter_limit(). *Postconditions:* s.lw == lw, s.lc == lc, s.lj == lj, s.ml == ml, and s.aa == aa.
X::surface_state_props::line_width(ST& s, float lw)	void		*Postconditions:* s.lw == lw.
X::surface_state_props::stroke_line_cap(ST& s, line_cap lc)	void		*Postconditions:* s.lc == lc.

Table 13 — `X::surface_state_props` requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::surface_state_props::stroke_line_join(ST& s, line_join lj)`	`void`		*Postconditions:* `s.lj == lj`.
`X::surface_state_props::miter_limit(ST& s, float ml)`	`void`		*Postconditions:* `s.ml == ml`.
`X::surface_state_props::anti_aliasing(ST& s, antialias aa)`	`void`	*Postconditions:* `s.aa == aa`.	
`X::surface_state_props::line_width(const ST& s)`	`float`	*Returns:* `s.lw`.	
`X::surface_state_props::stroke_line_cap(const ST& s)`	`line_cap`	*Returns:* `s.lc`.	
`X::surface_state_props::stroke_line_join(const ST& s)`	`line_join`	*Returns:* `s.lj`.	
`X::surface_state_props::miter_limit(const ST& s)`	`float`	*Returns:* `s.ml`.	
`X::surfaces_state_props::anti_aliasing(const ST& s)`	`antialias`	*Returns:* `s.aa`.	
`X::surface_state_props::max_miter_limit()`	`float`	*Returns:* An implementation-defined maximum value for `ST::ml`.	
`X::surface_state_props::create_fill_props(fill_rule fr = fill_rule::winding, antialias aa = antialias::good)`	`FP`	*Returns:* An object `f`.	*Postconditions:* `f.fr == fr` and `f.aa == aa`.
`X::surface_state_props::fill_fill_rule(FP& f, fill_rule fr)`	`void`		*Postconditions:* `f.fr == fr`.
`X::surface_state_props::antialiasing(FP& f, antialias aa)`	`void`		*Postconditions:* `f.aa == aa`.
`X::surface_state_props::fill_fill_rule(const FP& f)`	`fill_rule`	*Returns:* `f.fr`.	

Table 13 — `X::surface_state_props` requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::surface_state_props::antialiasing(const FP& f)`	`antialias`	*Returns:* `f.aa`.	
`X::surface_state_props::create_mask_props(wrap_mode wm = wrap_mode::none, filter fi = filter::good, const M& m = M())`	`MA`	*Returns:* An object `ma`.	*Postconditions:* `ma.wm == wm`, `ma.fi == fi`, and `ma.m == m`.
`X::surface_state_props::mask_wrap_mode(MA& ma, wrap_mode wm)`	`void`		*Postconditions:* `ma.wm == wm`.
`X::surface_state_props::mask_filter(MA& ma, filter fi)`	`void`		*Postconditions:* `ma.fi == fi`.
`X::surface_state_props::mask_matrix(MA& ma, const M& m)`	`void`		*Postconditions:* `ma.m == m`.
`X::surface_state_props::mask_wrap_mode(const MA& ma)`	`wrap_mode`	*Returns:* `ma.wm`.	
`X::surface_state_props::mask_filter(const MA& ma)`	`filter`	*Returns:* `ma.fi`.	
`X::surface_state_props::mask_matrix(const MA& ma)`	`M`	*Returns:* `ma.m`.	
`X::surface_state_props::create_dashes()`	`DA`	*Returns:* An object `d`.	*Postconditions:* `d.o == 0.0f` and `d.p == vector<float>{}`.
`template <class InputIterator> X::surface_state_props::create_dashes(float o, InputIterator first, InputIterator last)`	`DA`	*Returns:* An object `d`.	*Postconditions:* `d.o == o` and `d.p == vector<float>(first, last)`.
`X::surface_state_props::create_dashes(float o, initializer_list<float> il)`	`DA`	*Returns:* An object `d`.	*Postconditions:* `d.o == o` and `d.p == vector<float>(il)`.

Table 13 — `X::surface_state_props` requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::surface_state_props::equal(const DA& lhs, const DA& rhs)`	`bool`	*Returns:* `lhs.o == rhs.o && lhs.p == rhs.p`.	
`X::surface_state_props::not_equal(const DA& lhs, const DA& rhs)`	`bool`	*Returns:* `lhs.o != rhs.o || lhs.p != rhs.p`.	

8.2.7 `surfaces` requirements [io2d.graphsurf.reqs.surfaces]

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-name*s 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

typedef-name	Class data
`image_surface_data_type`	`basic_image_surface`
`output_surface_data_type`	`basic_output_surface`
`unmanaged_output_surface_data_type`	`basic_unmanaged_output_surface`

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

Type	Member data	Member data type
`I`	`im`	`IM`

Type	Member data	Member data type
I	fmt	format
I	dm	DP
O	ou	OU
O	fmt	format
O	dm	DP
O	bb	I
O	lb	optional
O	lbp	BP
O	sc	scaling
O	ac	bool
O	rr	bool
O	rs	refresh_style
O	dfr	float
O	dc	function<void(OUS&)>
O	scc	function<void(OUS&)>
U	un	UN
U	dm	DP
U	bb	I
U	lb	optional
U	lbp	BP
U	sc	scaling
U	ac	bool
U	dc	function<void(OUS&)>
U	scc	function<void(OUS&)>

12 [*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*]

13 [*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

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::surfaces::max_-image_dimensions()`	`DP`	*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`.	
`X::surfaces::max_-output_dimensions()`	`DP`	*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`.	

Table 16 — Graphics surfaces requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::surfaces::max_unmanaged_output_dimensions()`	`DP`	*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`.	
`X::surfaces::create_image_surface(format fmt, int w, int h)`	`I`	*Returns:* An object `i`.	*Requires:* `fmt != format::invalid`, `w > 0`, `w <= max_image_dimensions().x()`, `h > 0`, and `h <= max_image_dimensions().y()`. *Postconditions:* The bounds of `i.im` are `[0, w)` along the x axisand `[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*.

Table 16 — Graphics surfaces requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::surfaces::create_image_surface(filesystem::path f, image_file_format iff, format fmt)` `X::surfaces::create_image_surface(filesystem::path f, image_file_format iff, format fmt, error_code& ec) noexcept`	I	*Returns:* An object `i`.	*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`. *Postconditions:* The visual data format of `i.im` is `fmt`. The bounds and visual data of `i.im` are the result of processing the contents of `f` and transforming the visual data it contains into the visual data format `fmt`. `i.fmt == fmt`. `i.dm` is equal to the result of creating an object of type `DP` using the bounds obtained from processing the contents of `f`.

Table 16 — Graphics surfaces requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
(Continued...)			*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.
`X::surfaces::save(I& i, filesystem::path f, image_file_format iff)` `X::surfaces::save(I& i, filesystem::path f, image_file_format iff, error_code& ec) noexcept`	`void`	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`.	*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.
`X::surfaces::format(const I& i) noexcept`	`io2d::format`	*Returns:* `i.fmt`.	
`X::surfaces::dimensions(const I& i) noexcept`	`BP`	*Returns:* `i.dm`.	

Table 16 — Graphics surfaces requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::surfaces::clear(I& i)`	`void`	*Effects:* Equivalent to: `X::surfaces::paint(i, BR(rgba_color::transparent_black), nullopt, make_optional<RP>(antialias::none, M(), compositing_op::clear);`	
`X::surfaces::paint(I& i, const BB& b, const BP& bp, const RP& rp, const CP& cl);`	`void`	*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.	
`X::surfaces::stroke(I& i, const BB& b, const IP& ip, const BP& bp, const SP& sp, const D& d, const RP& rp, const CP& cl);`	`void`	*Effects:* Perform the stroking operation on `i.im` as specified in 15.3.6.	
`X::surfaces::fill(I& i, const BB& b, const IP& ip, const BP& bp, const RP& rp, const CP& cl);`	`void`	*Effects:* Perform the filling operation on `i.im` as specified in 15.3.5.	
`X::surfaces::mask(I& i, const BB& b, const BB& m, const BP& bp, const MP& mp, const RP& rp, const CP& cl);`	`void`	*Effects:* Perform the masking operation on `i.im` as specified in 15.3.7.	

Table 16 — Graphics surfaces requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::surfaces::create_output_surface(int pw, int ph, io2d::format pfmt, io2d::scaling scl, io2d::refresh_style rs, float fps)` `X::surfaces::create_output_surface(int pw, int ph, io2d::format pfmt, error_code& ec, io2d::scaling scl, io2d::refresh_style rs, float fps) noexcept`	O	*Returns:* An object `o`.	*Requires:* `pw > 0`, `pw <= min(max_image_dimensions().x(), max_output_dimensions().x())`, `ph > 0`, `ph <= min(max_image_dimensions().y(), max_output_dimensions().y())`, `pfmt != format::invalid`, and `fps < 0.0f`. *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*.

Table 16 — Graphics surfaces requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
(Continued...)			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`. *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)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
(Continued...)			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`. *Throws:* As specified in Error reporting (Clause 4). *Error conditions:* `errc::not_supported` 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).

Table 16 — Graphics surfaces requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`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)` `X::surfaces::create_output_surface(int pw, int ph, io2d::format pfmt, int pdw, int pdh, io2d::format pdfmt, error_code& ec, io2d::scaling scl, io2d::refresh_style rs, float fps) noexcept`	`O`	*Returns:* An object `o`.	*Requires:* `pw > 0`, `pw <= max_image_dimensions().x())`, `ph > 0`, `ph <= max_image_dimensions().y()`, `pfmt != format::invalid`, `pdw > 0`, `pdh <= max_output_dimensions().x()`, `pdh > 0`, `pdh <= max_output_dimensions().y()`, `pdfmt != format::invalid`, and `fps < 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`*.

Table 16 — Graphics surfaces requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
(Continued...)			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`. *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)

Expression	**Return type**	**Operational semantics**	**Assertion/note pre-/post-condition**
(Continued...)			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`. *Throws:* As specified in Error reporting (Clause 4). *Error conditions:* `errc::not_supported` 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).

Table 16 — Graphics surfaces requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::surfaces::begin_show(O& o, OUS& sfc)` `X::surfaces::begin_show(O& o, OUS& src, error_code& ec) noexcept`	`void`	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`.	*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 and `basic_unmanaged_display_surface` objects supported by the implementation (See: 15.3.8).

Table 16 — Graphics surfaces requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
(Continued...)			*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_show(O& o)`	`void`	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)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`dimensions(O& o, const DP& dm)`	`void`	Attempts to change the bounds of `o.bb.im` to `dm`. If successful, `o.bb.dm == dm`, otherwise there are no effects.	*Requires:* `dm.x() > 0`, `dm.x() <= max_image_dimensions().x())`, `dm.y() > 0`, `dm.y() <= max_image_dimensions().y()`
`output_dimensions(O& o, const DP& dm)`	`void`	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.	*Requires:* `dm.x() > 0`, `dm.x() <= max_output_dimensions().x())`, `dm.y() > 0`, `dm.y() <= max_output_dimensions().y()`
`scaling(O& o, io2d::scaling sc)`	`void`		*Postconditions:* `o.sc == sc`.
`refresh_style(O& o, io2d::refresh_style rs)`	`void`		*Postconditions:* `o.rs == rs`.
`desired_frame_rate(O& o, float dfr)`	`void`		*Requires:* `dfr > 0.0f`. *Postconditions:* `o.dfr == dfr`.
`letterbox_brush(O& o, const optional<BB>& lb, const optional<BP>& lbp)`	`void`		*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.lpb == lbp.value()`, otherwise `o.lbp == BP()`.
`letterbox_brush_properties(O& o, const optional<BP>& lbp)`	`void`		*Postconditions:* If `lbp.has_value() == true` then `o.lpb == lbp.value()`, otherwise `o.lbp == BP()`.
`auto_clear(O& o, bool ac)`	`void`		*Postconditions:* `o.ac == ac`.
`redraw_required(O& o, bool rr)`	`void`		*Postconditions:* `o.rr == rr`.

Table 16 — Graphics surfaces requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
draw_callback(O& o, function<void(OUS&)> dc)	void		*Postconditions:* o.dc == dc.
size_change_callback(O& o, function<void(OUS&)> scc)	void		*Postconditions:* o.scc == scc.
X::surfaces::clear(O& o)	void	*Effects:* Equivalent to: paint(o.bb).	
X::surfaces::paint(O& o, const BB& b, const BP& bp, const RP& rp, const CP& cl);	void	*Effects:* Equivalent to: paint(o.bb, <TODO>).	
X::surfaces::stroke(I& i, const BB& b, const IP& ip, const BP& bp, const SP& sp, const D& d, const RP& rp, const CP& cl);	void	Perform the stroking operation on i.im as specified in 15.3.6.	
X::surfaces::fill(I& i, const BB& b, const IP& ip, const BP& bp, const RP& rp, const CP& cl);	void	Perform the filling operation on i.im as specified in 15.3.5.	
X::surfaces::mask(I& i, const BB& b, const BB& m, const BP& bp, const MP& mp, const RP& rp, const CP& cl);	void	Perform the masking operation on i.im as specified in 15.3.5.	
X::surfaces::dimensions(const O& o) noexcept	DP	*Returns:* o.bb.dm.	
X::surfaces::output_dimensions(const O& o) noexcept	DP	*Returns:* o.dm.	
X::surfaces::refresh_style(const O& o) noexcept	io2d::refresh_style	*Returns:* o.rs	
X::surfaces::desired_frame_rate(const O& o) noexcept	float	*Returns:* o.dfr.	
X::surfaces::scaling(const O& o) noexcept	io2d::scaling	*Returns:* o.sc.	
X::surfaces::letterbox_brush(const O& o) noexcept	optional<BB>	*Returns:* o.lb.	
X::surfaces::letterbox_brush_props(const O& o) noexcept	BP	*Returns:* o.lbp.	
X::surfaces::auto_clear(const O& o) noexcept	bool	*Returns:* o.ac.	

Table 16 — Graphics surfaces requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`X::surfaces::redraw_-required(const O& o) noexcept`	`bool`	*Returns:* `o.rr`.	
`X::surfaces::copy_-surface(const I& i)`	`IMS`	*Returns:* An object `c`.	*Postconditions:* `c.data().im` is a copy of the data in `i.im`. `c.data().fmt == i.fmt`. `c.data().dm == i.dm`.
`X::surfaces::copy_-surface(const O& o)`	`IMS`	*Returns:* An object `c`.	*Postconditions:* `c.data().im` is a copy of the data in `o.bb.im`. `c.data().fmt == o.bb.fmt`. `c.data().dm == o.bb.dm`.

Table 16 — Graphics surfaces requirements (continued)

Expression	**Return type**	**Operational semantics**	**Assertion/note pre-/post-condition**
`X::surfaces::create_unmanaged_output_surface(/*` *`implementation-defined`*`*/)`	`UN`	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.	[*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*]
`template <class F> X::surfaces::draw_callback(UN& un, F&& f)`	`void`	<TODO>	*Requires:* `f` shall be `CopyConstructible`.

Table 16 — Graphics surfaces requirements (continued)

Expression	**Return type**	**Operational semantics**	**Assertion/note pre-/post-condition**

9 Linear algebra [io2d.linearalgebra]

9.1 Class basic_point_2d [io2d.point2d]

9.1.1 basic_point_2d description [io2d.point2d.intro]

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 [io2d.point2d.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;
  };

  // 9.1.8, non-member operators:
  template <class GraphicsMath>
  bool operator==(const basic_point_2d<GraphicsMath>& lhs,
    const basic_point_2d<GraphicsMath>& rhs) noexcept;
  template <class GraphicsMath>
  bool 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>& 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;
    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;
```

1 *Effects:* Constructs an object of type `basic_point_2d`.

2 *Postconditions:* `data() == GraphicsMath::create_point_2d()`.

```
basic_point_2d(float x, float y) noexcept;
```

3 *Effects:* Constructs an object of type `basic_point_2d`.

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

```
basic_point_2d(const data_type& d) noexcept;
```

5 *Effects:* Constructs an object of type `basic_point_2d`.

6 *Postconditions:* `data() == d`.

9.1.4 `basic_point_2d` accessors [io2d.point2d.accessors]

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

1 *Returns:* A reference to the `basic_point_2d` object's data object (See: 9.1.1).

9.1.5 basic_point_2d modifiers [io2d.point2d.modifiers]

`void x(float val) noexcept;`

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

`void y(float val) noexcept;`

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

9.1.6 basic_point_2d observers [io2d.point2d.observers]

`float x() const noexcept;`

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

`float y() const noexcept;`

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

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

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

`float magnitude() const noexcept;`

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

`float magnitude_squared() const noexcept;`

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

`float angular_direction() const noexcept;`

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

`basic_point_2d to_unit() const noexcept;`

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

9.1.7 basic_point_2d member operators [io2d.point2d.member.ops]

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

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

2 *Returns:* `*this`.

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

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

4 *Returns:* `*this`.

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

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

6 *Returns:* `*this`.

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

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

8 *Returns:* `*this`.

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

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

10 *Returns:* `*this`.

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

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

12 *Returns:* `*this`.

9.1.8 basic_point_2d non-member operators [io2d.point2d.ops]

```
bool operator==(const basic_point_2d& lhs, const basic_point_2d& rhs) noexcept;
```

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

```
bool operator!=(const basic_point_2d& lhs, const basic_point_2d& rhs) noexcept;
```

2 *Returns:* `GraphicsMath::not_equal(lhs.data(), rhs.data())`.

```
basic_point_2d operator+(const basic_point_2d& val) noexcept;
```

3 *Returns:* `val`.

```
basic_point_2d operator+(const basic_point_2d& lhs, const basic_point_2d& rhs) noexcept;
```

4 *Returns:* `basic_point_2d(GraphicsMath::add(lhs.data(), rhs.data()))`.

```
basic_point_2d operator-(const basic_point_2d& val) noexcept;
```

5 *Returns:* `basic_point_2d(GraphicsMath::negate(val.data()))`.

```
basic_point_2d operator-(const basic_point_2d& lhs, const basic_point_2d& rhs) noexcept;
```

6 *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;
```

7 *Returns:* `basic_point_2d(GraphicsMath::multiply(lhs.data(), rhs.data()))`.

```
basic_point_2d operator*(const basic_point_2d& lhs, float rhs) noexcept;
```

8 *Returns:* `basic_point_2d(GraphicsMath::multiply(lhs.data(), rhs))`.

```
basic_point_2d operator*(float lhs, const basic_point_2d& rhs) noexcept;
```

9 *Returns:* `basic_point_2d(GraphicsMath::multiply(lhs, rhs.data()))`.

```
basic_point_2d operator/(const basic_point_2d& lhs, const basic_point_2d& rhs) noexcept;
```

10 *Requires:* `rhs.x() != 0.0f` and `rhs.y() != 0.0f`.

11 *Returns:* `basic_point_2d(GraphicsMath::divide(lhs.data(), rhs.data()))`.

```
basic_point_2d operator/(const basic_point_2d& lhs, float rhs) noexcept;
```

12 *Requires:* `rhs != 0.0f`.

13 *Returns:* `basic_point_2d(GraphicsMath::divide(lhs.data(), rhs))`.

```
basic_point_2d operator/(float lhs, const basic_point_2d& rhs) noexcept;
```

14 *Requires:* `rhs.x() != 0.0f` and `rhs.y() != 0.0f`.

15 *Returns:* `basic_point_2d(GraphicsMath::divide(lhs, rhs.data()))`.

9.2 Class basic_matrix_2d [io2d.matrix2d]

9.2.1 basic_matrix_2d description [io2d.matrix2d.intro]

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

2 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.

3 The specification of the `basic_matrix_2d` class, as described in this subclause, uses the following ordering:
[[$m00$ $m01$ $m02$]]
[[$m10$ $m11$ $m12$]]
[[$m20$ $m21$ $m22$]]

4 [*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*]

5 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.

6 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 [io2d.matrix2d.synopsis]

```
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;
  };
```

```
  // 9.2.9, member operators:
  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;
}
```

9.2.3 basic_matrix_2d constructors [io2d.matrix2d.cons]

```
basic_matrix_2d() noexcept;
```

1 *Effects:* Constructs an object of type `basic_matrix_2d`.

2 [*Note:* The resulting matrix is the identity matrix. — *end note*]

3 *Postconditions:* `data() == GraphicsMath::create_matrix_2d()`.

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

4 *Effects:* Constructs an object of type `basic_matrix_2d`.

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

```
basic_matrix_2d(const data_type& v) noexcept;
```

6 *Effects:* Constructs an object of type `basic_matrix_2d`.

7 *Postconditions:* `data() == v`.

9.2.4 basic_matrix_2d accessors [io2d.matrix2d.accessors]

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

1 *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 [io2d.matrix2d.staticfactories]

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

1 *Returns:* `basic_matrix_2d(GraphicsMath::create_translate(v.data()))`.

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

2 *Returns:* `basic_matrix_2d(GraphicsMath::create_scale(v.data())`.

```
static basic_matrix_2d init_rotate(float radians) noexcept;
```

3 *Returns:* `basic_matrix_2d(GraphicsMath::create_rotate(radians))`.

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

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

```
static basic_matrix_2d init_reflect(float radians) noexcept;
```

5 *Returns:* `basic_matrix_2d(GraphicsMath::create_reflect(radians))`

```
static basic_matrix_2d init_shear_x(float factor) noexcept;
```

6 *Returns:* `basic_matrix_2d(GraphicsMath::create_shear_x(factor))`.

```
static basic_matrix_2d init_shear_y(float factor) noexcept;
```

7 *Returns:* `basic_matrix_2d(GraphicsMath::create_shear_y(factor))`.

9.2.6 `basic_matrix_2d` modifiers [io2d.matrix2d.modifiers]

```
void m00(float v) noexcept;
```

1 *Effects:* Equivalent to `GraphicsMath::m00(data(), v);`

```
void m01(float v) noexcept;
```

2 *Effects:* Equivalent to `GraphicsMath::m01(data(), v);`

```
void m10(float v) noexcept;
```

3 *Effects:* Equivalent to `GraphicsMath::m10(data(), v);`

```
void m11(float v) noexcept;
```

4 *Effects:* Equivalent to `GraphicsMath::m11(data(), v);`

```
void m20(float v) noexcept;
```

5 *Effects:* Equivalent to `GraphicsMath::m20(data(), v);`

```
void m21(float v) noexcept;
```

6 *Effects:* Equivalent to `GraphicsMath::m21(data(), v);`

```
basic_matrix_2d& translate(basic_point_2d<GraphicsMath> v) noexcept;
```

7 *Effects:* Equivalent to `data() = GraphicsMath::translate(data(), v.data());`

8 *Returns:* `*this`.

```
basic_matrix_2d& scale(basic_point_2d<GraphicsMath> v) noexcept;
```

9 *Effects:* Equivalent to `data() = GraphicsMath::scale(data(), v.data());`

10 *Returns:* `*this`.

```
basic_matrix_2d& rotate(float radians) noexcept;
```

11 *Effects:* Equivalent to `data() = GraphicsMath::rotate(data(), radians);`

12 *Returns:* `*this`.

```
basic_matrix_2d& rotate(float radians, basic_point_2d<GraphicsMath> origin) noexcept;
```

13 *Effects:* Equivalent to `data() = GraphicsMath::rotate(data(), radians, origin.data());`

14 *Returns:* `*this`.

```
basic_matrix_2d& reflect(float radians) noexcept;
```

15 *Effects:* Equivalent to `data() = GraphicsMath::reflect(data(), radians);`

16 *Returns:* `*this`.

```
basic_matrix_2d& shear_x(float factor) noexcept;
```

17 *Effects:* Equivalent to `data() = GraphicsMath::shear_x(data(), factor);`

18 *Returns:* `*this`.

```
basic_matrix_2d& shear_y(float factor) noexcept;
```

19 *Effects:* Equivalent to `data() = GraphicsMath::shear_y(factor);`

20 *Returns:* `*this`.

9.2.7 `basic_matrix_2d` observers [io2d.matrix2d.observers]

```
float m00() const noexcept;
```

1 *Returns:* `GraphicsMath::m00(data())`.

```
float m01() const noexcept;
```

2 *Returns:* GraphicsMath::m01(data()).

```
float m10() const noexcept;
```

3 *Returns:* GraphicsMath::m10(data()).

```
float m11() const noexcept;
```

4 *Returns:* GraphicsMath::m11(data()).

```
float m20() const noexcept;
```

5 *Returns:* GraphicsMath::m20(data()).

```
float m21() const noexcept;
```

6 *Returns:* GraphicsMath::m21(data()).

```
bool is_finite() const noexcept;
```

7 *Returns:* GraphicsMath::is_finite(data()).

```
bool is_invertible() const noexcept;
```

8 *Requires:* is_finite() == true.

9 *Returns:* GraphicsMath::is_invertible(data()).

```
basic_matrix_2d inverse() const noexcept;
```

10 *Requires:* is_invertible() == true.

11 *Returns:* basic_matrix_2d(GraphicsMath::inverse(data())).

```
float determinant() const noexcept;
```

12 *Requires:* is_finite() == true.

13 *Returns:* GraphicsMath::determinant(data()).

```
basic_point_2d<GraphicsMath> transform_pt(basic_point_2d<GraphicsMath> pt) const noexcept;
```

14 *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;
```

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

2 *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;
```

1 *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;
```

2 *Returns:* Equivalent to m.transform_pt(v).

```
bool operator==(const basic_matrix_2d& lhs, const basic_matrix_2d& rhs) noexcept;
```

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

```
bool operator!=(const basic_matrix_2d& lhs, const basic_matrix_2d& rhs) noexcept;
```

4 *Returns:* Equivalent to GraphicsMath::not_equal(lhs.data(), rhs.data()).

10 Geometry [io2d.geometry]

10.1 Class template `basic_display_point` [io2d.displaypt]

10.1.1 `basic_display_point` description [io2d.displaypt.intro]

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 [io2d.displaypt.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 [io2d.displaypt.cons]

`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;
```

7 *Effects:* Constructs an object of type `basic_display_point`.

8 *Postconditions:* `data() == val`.

9 *Remarks:* The x coordinate is `GraphicsMath::x(val)` and the y coordinate is `GraphicsMath::y(val)`.

10.1.4 basic_display_point accessors [io2d.displaypt.accessors]

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

1 *Returns:* A reference to the `basic_display_point` object's data object (See: 10.1.1).

10.1.5 basic_display_point modifiers [io2d.displaypt.modifiers]

```
void x(int v) noexcept;
```

1 *Effects:* Equivalent to `GraphicsMath::x(data(), v);`

```
void y(int v) noexcept;
```

2 *Effects:* Equivalent to `GraphicsMath::y(data(), v);`

10.1.6 basic_display_point observers [io2d.displaypt.observers]

```
int x() const noexcept;
```

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

```
int y() const noexcept;
```

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

10.1.7 basic_display_point operators [io2d.displaypt.ops]

```
bool operator==(const basic_display_point<GraphicsMath>& lhs,
  const basic_display_point<GraphicsMath>& rhs) noexcept;
```

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

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

2 *Returns:* `GraphicsMath::not_equal(lhs.data(), rhs.data())`.

10.2 Class basic_bounding_box [io2d.bounding_box]

10.2.1 basic_bounding_box description [io2d.bounding_box.intro]

1 The class template `basic_bounding_box` describes a bounding_box.

2 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`.

3 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 [io2d.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;
```

```
    // 10.2.4, 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;
```

2 *Effects:* Equivalent to GraphicsMath::y(data(), v);

```
void width(float v) noexcept;
```

3 *Effects:* Equivalent to GraphicsMath::width(data(), v);

```
void height(float val) noexcept;
```

4 *Effects:* Equivalent to GraphicsMath::height(data(), v);

```
void top_left(const basic_point_2d<GraphicsMath>& v) noexcept;
```

5 *Effects:* Equivalent to GraphicsMath::top_left(data(), v.data());

```
void bottom_right(const basic_point_2d<GraphicsMath>& v) noexcept;
```

6 *Effects:* Equivalent to GraphicsMath::bottom_right(data(), v.data());

10.2.6 basic_bounding_box observers [io2d.bounding_box.observers]

```
float x() const noexcept;
```

1 *Returns:* GraphicsMath::x(data()).

```
float y() const noexcept;
```

2 *Returns:* GraphicsMath::y(data()).

```
float width() const noexcept;
```

3 *Returns:* GraphicsMath::width(data()).

```
float height() const noexcept;
```

4 *Returns:* GraphicsMath::height(data()).

```
basic_point_2d<GraphicsMath> top_left() const noexcept;
```

5 *Returns:* basic_point_2d<GraphicsMath>(GraphicsMath::top_left(data())).

```
basic_point_2d<GraphicsMath> bottom_right() const noexcept;
```

6 *Returns:* basic_point_2d<GraphicsMath>(GraphicsMath::bottom_right(data())).

10.2.7 basic_bounding_box operators [io2d.bounding_box.ops]

```
bool operator==(const basic_bounding_box<GraphicsMath>& lhs,
  const basic_bounding_box<GraphicsMath>& rhs) noexcept;
```

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

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

2 *Returns:* GraphicsMath::not_equal(lhs.data(), rhs.data()).

10.3 Class basic_circle [io2d.circle]

10.3.1 basic_circle description [io2d.circle.intro]

1 The class template basic_circle describes a circle.

2 It has a *center* of type basic_point_2d<GraphicsMath> and a *radius* of type float.

3 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 [io2d.circle.synopsis]

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

```
    // 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 [io2d.circle.cons]

`basic_circle() noexcept;`

1 *Effects:* Constructs an object of type `basic_circle`.

2 *Postconditions:* `data() == GraphicsMath::create_circle()`.

`basic_circle(const basic_point_2d<GraphicsMath>& ctr, float r) noexcept;`

Requires: `r >= 0.0f`.

3 *Effects:* Constructs an object of type `basic_circle`.

4 *Postconditions:* `data() == GraphicsMath::create_circle(ctr, r)`.

10.3.4 `basic_circle` accessors [io2d.circle.accessors]

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

1 *Returns:* A reference to the `basic_circle` object's data object (See: 10.3.1).

10.3.5 `basic_circle` modifiers [io2d.circle.modifiers]

`void center(const basic_point_2d<GraphicsMath>& ctr) noexcept;`

1 *Effects:* Equivalent to `GraphicsMath::center(data(), ctr.data());`

`void radius(float r) noexcept;`

Requires: `r >= 0.0f`.

2 *Effects:* Equivalent to `GraphicsMath::radius(data(), r);`

10.3.6 `basic_circle` observers [io2d.circle.observers]

`basic_point_2d<GraphicsMath> center() const noexcept;`

1 *Returns:* `(basic_point_2d<GraphicsMath>(GraphicsMath::center(data()))`.

`float radius() const noexcept;`

2 *Returns:* `GraphicsMath::radius(data())`.

10.3.7 basic_circle operators [io2d.circle.ops]

```
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 [io2d.paths]

12.1 Overview of paths [io2d.paths.overview]

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

2 A path contains zero or more figures.

3 A figure is composed of at least one segment.

4 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*]

5 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.

6 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) [io2d.paths.example]

12.2.1 Overview [io2d.paths.example.intro]

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

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

```
#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.

  // Example code ends.

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

12.2.2 Example 1 [io2d.paths.examples.one]

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

```
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
```

```
    // [220 + 60, 160 + 20]. The "to" point is relative to the starting point.
pb.rel_line({ -260.0f, 0.0f }); // Line from [280, 180] to
    // [280 - 260, 180 + 0].
pb.close_figure(); // Creates a line from [20, 180] to [80, 20]
    // (the new figure point), which makes this a closed figure.
imgSfc.stroke(foreBrush, pb, nullopt, nullopt, nullopt, aliased);
```

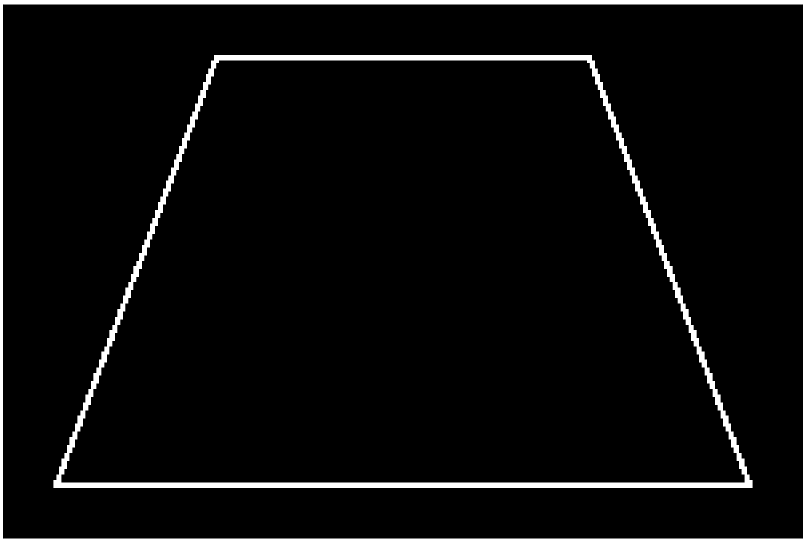

Figure 1 — Example 1 result

12.2.3 Example 2 [io2d.paths.examples.two]

1 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):

```
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, alised);
```

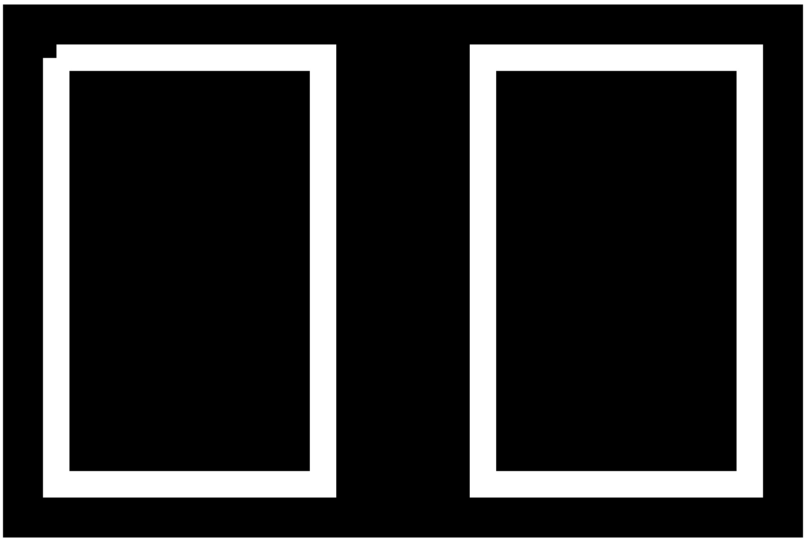

Figure 2 — Example 2 result

2 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`.

3 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.

4 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 }`.

5 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 [io2d.paths.examples.three]

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

```
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);
```

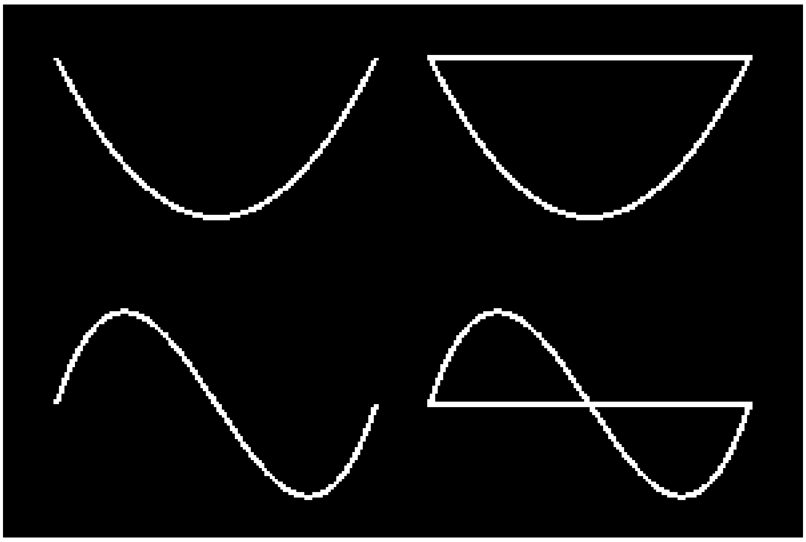

Figure 3 — Path example 3

2 [*Note:* `pb.quadratic_curve({ 80.0f, 140.0f }, { 140.0f, 20.0f });` would be the absolute equivalent of the first curve in example 3. — *end note*]

12.2.5 Example 4 [io2d.paths.examples.four]

1 Example 4 shows how to draw "C++" using figures.

2 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.

3 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.

4 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.

```
// 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({ 0.0f, 25.0f });
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);
```

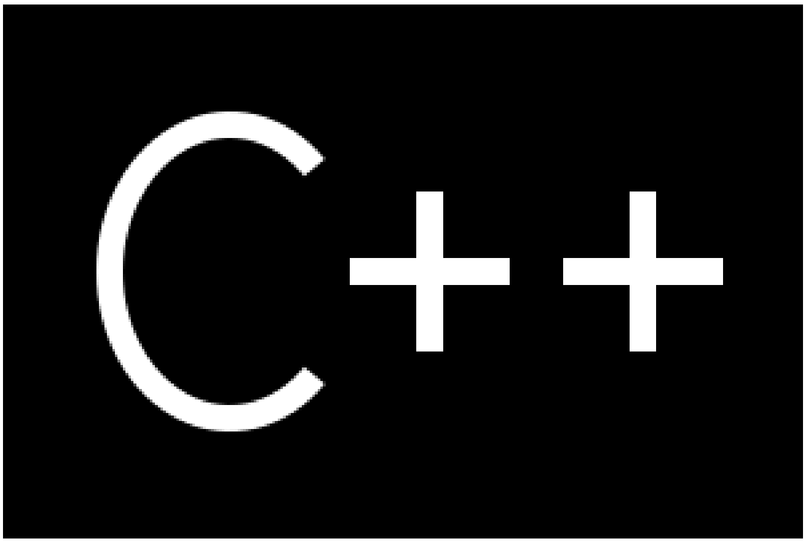

Figure 4 — Path example 4

12.3 Class template `basic_figure_items` [io2d.paths.figureitems]

12.3.1 Introduction [io2d.paths.figureitems.intro]

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 [io2d.paths.figureitems.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::experimemtal::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();
```

1 *Effects:* Constructs an object of type `abs_new_figure`.

2 *Postconditions:* `data() == GraphicsSurfaces::paths::create_abs_new_figure()`.

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

```
explicit abs_new_figure(const basic_point_2d<graphics_math_type>& pt);
```

4 *Effects:* Constructs an object of type `abs_new_figure`.

5 *Postconditions:* `data() == GraphicsSurfaces::paths::create_abs_new_figure(pt)`.

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

12.3.3.4 Accessors [io2d.absnewfigure.acc]

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

1 *Returns:* A reference to the `abs_new_figure` object's data object (See: 12.3.3.1).

12.3.3.5 Modifiers [io2d.absnewfigure.mod]

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

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

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

12.3.3.6 Observers [io2d.absnewfigure.obs]

```
basic_point_2d<graphics_math_type> at() const noexcept;
```

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

2 *Remarks:* The returned value is the at point.

12.3.3.7 Equality operators [io2d.absnewfigure.eq]

```
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;
```

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

```
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;
```

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

12.3.4 Class template `basic_figure_items<GraphicsSurfaces>::rel_new_figure` [io2d.relnewfigure]

12.3.4.1 Overview [io2d.relnewfigure.intro]

1 The class template `basic_figure_items<GraphicsSurfaces>::rel_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::rel_new_figure_data_-type`. It is accessible using the `data` member functions.

12.3.4.2 Synopsis [io2d.relnewfigure.synopsis]

```
namespace std::experimemtal::io2d::v1 {
  template <class GraphicsSurfaces>
  class basic_figure_items<GraphicsSurfaces>::rel_new_figure {
  public:
    using graphics_math_type = typename GraphicsSurfaces::graphics_math_type;
    using data_type =
      typename GraphicsSurfaces::paths::rel_new_figure_data_type;
```

```
    // 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 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;
}
```

12.3.4.3 Constructors [io2d.relnewfigure.ctor]

```
rel_new_figure() noexcept;
```

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

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

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

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

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

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

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

12.3.4.4 Accessors [io2d.relnewfigure.acc]

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

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

12.3.4.5 Modifiers [io2d.relnewfigure.mod]

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

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

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

12.3.4.6 Observers [io2d.relnewfigure.obs]

```
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 [io2d.relnewfigure.eq]

```
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` [io2d.closefigure]

12.3.5.1 Overview [io2d.closefigure.intro]

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

12.3.5.2 Synopsis [io2d.closefigure.synopsis]

```
namespace std::experimemtal::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 [io2d.closefigure.eq]

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

```
  noexcept;
```

1 *Returns:* `true`.

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

2 *Returns:* `false`.

12.3.6 Class template `basic_figure_items<GraphicsSurfaces>::abs_matrix` [io2d.absmatrix]

12.3.6.1 Overview [io2d.absmatrix.intro]

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

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

3 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]

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

    // 12.3.6.3, construct:
    abs_matrix();
    explicit abs_matrix(const basic_matrix_2d<graphics_math_type>& m) noexcept;
    abs_matrix(const abs_matrix& other) = default;
    abs_matrix(abs_matrix&& other) noexcept = default;

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

    // 12.3.6.4, accessors:
    const data_type& data() const noexcept;
    data_type& data() noexcept;

    // 12.3.6.5, modifiers:
    void matrix(const basic_matrix_2d<graphics_math_type>& m) noexcept;

    // 12.3.6.6, observers:
    basic_matrix_2d<graphics_math_type> matrix() const noexcept;
  };

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

12.3.6.3 Constructors [io2d.absmatrix.ctor]

```
abs_matrix() noexcept;
```

1 *Effects:* Constructs an abs_matrix object.

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

3 *Remarks:* The transform matrix is basic_matrix_2d<graphics_math_type().

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

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

5 *Effects:* Constructs an abs_matrix object.

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

7 *Remarks:* The transform matrix is m.

12.3.6.4 Accessors [io2d.absmatrix.acc]

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

1 *Returns:* A reference to the abs_matrix object's data object (See: 12.3.6.1).

12.3.6.5 Modifiers [io2d.absmatrix.mod]

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

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

2 *Effects:* The transform matrix is m.

12.3.6.6 Observers [io2d.absmatrix.obs]

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

1 *Returns:* The transform matrix.

12.3.6.7 Equality operators [io2d.absmatrix.eq]

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

1 *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;
```

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

12.3.7 Class template basic_figure_items<GraphicsSurfaces>::rel_matrix [io2d.relmatrix]

12.3.7.1 Overview [io2d.relmatrix.intro]

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

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

3 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 [io2d.relmatrix.synopsis]

```
namespace std::experimemtal::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 [io2d.relmatrix.ctor]

`rel_matrix() noexcept;`

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

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

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

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

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

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

12.3.7.4 Accessors [io2d.relmatrix.acc]

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

1 *Returns:* A reference to the `rel_matrix` object's data object (See: 12.3.7.1).

12.3.7.5 Modifiers [io2d.relmatrix.mod]

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

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

2 *Effects:* The transform matrix is `m`.

12.3.7.6 Observers [io2d.relmatrix.obs]

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

1 *Returns:* The transform matrix.

12.3.7.7 Equality operators [io2d.relmatrix.eq]

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

1 *Returns:* `lhs.matrix() == rhs.matrix()`.

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

2 *Returns:* `lhs.matrix() != rhs.matrix()`.

12.3.8 Class template `basic_figure_items<GraphicsSurfaces>::revert_matrix` [io2d.revertmatrix]

12.3.8.1 Overview [io2d.revertmatrix.intro]

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

12.3.8.2 Synopsis [io2d.revertmatrix.synopsis]

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

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

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

12.3.8.3 Equality operators [io2d.revertmatrix.eq]

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

1 *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;
```

2 *Returns:* `false`.

12.3.9 Class template `basic_figure_items<GraphicsSurfaces>::abs_cubic_curve` [io2d.abscubiccurve]

12.3.9.1 Overview [io2d.abscubiccurve.intro]

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

2 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>`.

3 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 [io2d.abscubiccurve.synopsis]

```
namespace std::experimemtal::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:
    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;
  }
```

12.3.9.3 Constructors [io2d.abscubiccurve.ctor]

```
abs_cubic_curve() noexcept;
```

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

```
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;
```

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

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

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

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

12.3.9.4 Accessors [io2d.abscubiccurve.acc]

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

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

12.3.9.5 Modifiers [io2d.abscubiccurve.mod]

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

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

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

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

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

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

12.3.9.6 Observers [io2d.abscubiccurve.obs]

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

1 *Returns:* The first control point.

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

2 *Returns:* The second control point.

```
basic_point_2d<graphics_math_type> end_pt() const noexcept;
```

3 *Returns:* The end point.

12.3.9.7 Equality operators [io2d.abscubiccurve.eq]

```
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;
```

1 *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>::abs_cubic_curve& lhs,
  const typename basic_figure_items<GraphicsSurfaces>::abs_cubic_curve& rhs)
  noexcept;
```

2 *Returns:* `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` [io2d.relcubiccurve]

12.3.10.1 Overview [io2d.relcubiccurve.intro]

1 The class `basic_figure_items<GraphicsSurfaces>::rel_cubic_curve` describes a figure item that is a segment.

2 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>`.

3 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 [io2d.relcubiccurve.synopsis]

```
namespace std::experimemtal::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:
    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:
  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;
}
```

12.3.10.3 Constructors [io2d.relcubiccurve.ctor]

```
rel_cubic_curve() noexcept;
```

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

```
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;
```

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

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

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

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

12.3.10.4 Accessors [io2d.relcubiccurve.acc]

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

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

12.3.10.5 Modifiers [io2d.relcubiccurve.mod]

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

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

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

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

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

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

12.3.10.6 Observers [io2d.relcubiccurve.obs]

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

1 *Returns:* The first control point.

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

2 *Returns:* The second control point.

```
basic_point_2d<graphics_math_type> end_pt() const noexcept;
```

3 *Returns:* The end point.

12.3.10.7 Equality operators [io2d.relcubiccurve.eq]

```
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;
```

1 *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;
```

2 *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` [io2d.absline]

12.3.11.1 Overview [io2d.absline.intro]

1 The class `basic_figure_items<GraphicsSurfaces>::abs_line` describes a figure item that is a segment.

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

3 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 [io2d.absline.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:
    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;
  }
```

12.3.11.3 Constructors [io2d.absline.ctor]

```
abs_line() noexcept;
```

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

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

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

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

12.3.11.4 Accessors [io2d.absline.acc]

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

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

12.3.11.5 Modifiers [io2d.absline.mod]

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

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

12.3.11.6 Observers [io2d.absline.obs]

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

1 *Returns:* The end point.

12.3.11.7 Equality operators [io2d.absline.eq]

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

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

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

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

12.3.12 Class `rel_line` [io2d.relline]

12.3.12.1 Overview [io2d.relline.intro]

1 The class `basic_figure_items<GraphicsSurfaces>::rel_line` describes a figure item that is a segment.

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

3 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 [io2d.relline.synopsis]

```
namespace std::experimemtal::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;
  };

  // 12.3.12.7, 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;
  template <class GraphicsSurfaces>
  bool operator!=(
    const typename basic_figure_items<GraphicsSurfaces>::rel_line& lhs,
    const typename basic_figure_items<GraphicsSurfaces>::rel_line& rhs)
    noexcept;
}
```

12.3.12.3 Constructors [io2d.relline.ctor]

```
rel_line() noexcept;
```

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

```
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 [io2d.relline.acc]

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

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

12.3.12.5 Modifiers [io2d.relline.mod]

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

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

12.3.12.6 Observers [io2d.relline.obs]

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

1 *Returns:* The end point.

12.3.12.7 Equality operators [io2d.relline.eq]

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

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

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

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

12.3.13 Class template `basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve` [io2d.absquadraticcurve]

12.3.13.1 Overview [io2d.absquadraticcurve.intro]

1 The class `basic_figure_items<GraphicsSurfaces>::abs_quadratic_curve` describes a figure item that is a segment.

2 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>`.

3 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 [io2d.absquadraticcurve.synopsis]

```
namespace std::experimemtal::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 [io2d.absquadraticcurve.ctor]

```
abs_quadratic_curve() noexcept;
```

1 *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;
```

2 *Effects:* Constructs an object of type `abs_quadratic_curve`.

3 *Remarks:* The control point is `cpt`.

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

12.3.13.4 Accessors [io2d.absquadraticcurve.acc]

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

1 *Returns:* A reference to the `abs_quadratic_curve` object's data object (See: 12.3.13.1).

12.3.13.5 Modifiers [io2d.absquadraticcurve.mod]

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

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

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

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

12.3.13.6 Observers [io2d.absquadraticcurve.obs]

```
basic_point_2d<graphics_math_type> control_pt() const noexcept;
```

1 *Returns:* The control point.

```
basic_point_2d<graphics_math_type> end_pt() const noexcept;
```

2 *Returns:* The end point.

12.3.13.7 Equality operators [io2d.absquadraticcurve.eq]

```
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;
```

1 *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;
```

2 *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]

1 The class `basic_figure_items<GraphicsSurfaces>::rel_quadratic_curve` describes a figure item that is a segment.

2 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>`.

3 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]

```
namespace std::experimemtal::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:
  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;
```

```
}
```

12.3.14.3 Constructors [io2d.relquadraticcurve.ctor]

```
rel_quadratic_curve() noexcept;
```

1 *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;
```

2 *Effects:* Constructs an object of type `rel_quadratic_curve`.

3 *Remarks:* The control point is `cpt`.

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

12.3.14.4 Accessors [io2d.relquadraticcurve.acc]

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

1 *Returns:* A reference to the `rel_quadratic_curve` object's data object (See: 12.3.14.1).

12.3.14.5 Modifiers [io2d.relquadraticcurve.mod]

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

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

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

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

12.3.14.6 Observers [io2d.relquadraticcurve.obs]

```
basic_point_2d<graphics_math_type> control_pt() const noexcept;
```

1 *Returns:* The control point.

```
basic_point_2d<graphics_math_type> end_pt() const noexcept;
```

2 *Returns:* The end point.

12.3.14.7 Equality operators [io2d.relquadraticcurve.eq]

```
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;
```

1 *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;
```

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

12.3.15 Class template `basic_figure_items<GraphicsSurfaces>::arc` [io2d.arc]

12.3.15.1 Overview [io2d.arc.intro]

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

2 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`.

3 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.

4 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 [io2d.arc.synopsis]

```
namespace std::experimemtal::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, accessors:
    const data_type& data() const noexcept;
    data_type& data() noexcept;

    // 12.3.15.5, modifiers:
    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, observers:
    basic_point_2d<typename GraphicsSurfaces::graphics_math_type> radius() const noexcept;
    float rotation() const noexcept;
    float start_angle() const noexcept;
    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;
    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;
  };

  // 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 [io2d.arc.ctor]

`arc() noexcept;`

1 *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;
```

2 *Effects:* Constructs an object of type `arc`.

3 The radius is `rad`.

4 The rotation is `rot`.

5 The start angle is `sang`.

12.3.15.4 Accessors [io2d.arc.acc]

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

1 *Returns:* A reference to the `arc` object's data object (See: 12.3.15.1).

12.3.15.5 Modifiers [io2d.arc.mod]

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

1 *Effects:* The radius is `rad`.

```
constexpr void rotation(float rot) noexcept;
```

2 *Effects:* The rotation is `rot`.

```
void start_angle(float sang) noexcept;
```

3 *Effects:* The start angle is `sang`.

12.3.15.6 Observers [io2d.arc.obs]

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

1 *Returns:* The radius.

```
float rotation() const noexcept;
```

2 *Returns:* The rotation.

```
float start_angle() const noexcept;
```

3 *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;
```

4 *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;
```

5 *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 [io2d.arc.eq]

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

1 *Returns:*

```
lhs.radius() == rhs.radius() && lhs.rotation() == rhs.rotation() &&
lhs.start_angle() && rhs.start_angle()
```

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

2 *Returns:*

```
lhs.radius() != rhs.radius() || lhs.rotation() != rhs.rotation() ||
lhs.start_angle() != rhs.start_angle()
```

12.3.16 Path interpretation [io2d.paths.interpretation]

1 This subclause describes how to interpret a path for use in a rendering and composing operation.

2 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.

3 The interpretation of a path requires the state data specified in Table 17.

Table 17 — Path interpretation state data

Name	Description	Type	Initial value
`mtx`	Path transformation matrix	`matrix_2d`	`matrix_2d{ }`
`currPt`	Current point	`point_2d`	*`unspecified`*
`lnfPt`	Last new figure point	`point_2d`	*`unspecified`*
`mtxStk`	Matrix stack	`stack<matrix_2d>`	`stack<matrix_2d>{ }`

4 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:

(4.1) — 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,

(4.2) — any path command figure items are evaluated with full effect.

.

5 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.

6 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 18 — Figure item interpretation effects

Figure item	Effects
`figure_items::abs_new_-figure` `p`	Creates a new figure. Sets `currPt` to `p.at() * mtx`. Sets `lnfPt` to `currPt`.
`figure_items::rel_new_-figure` `p`	Let `mm` equal `mtx`. Let `mm.m20` equal 0.0f. Let `mm.m21` equal `0.0f`. Creates a new figure. Sets `currPt` to `currPt + p.at() * mm`. Sets `lnfPt` to `currPt`.

Table 18 — Figure item interpretation effects (continued)

Figure item	Effects
`figure_items::close_figure p`	Creates a line from `currPt` to `lnfPt`. Makes the current figure a closed figure. Creates a new figure. Sets `currPt` to `lnfPt`.
`figure_items::abs_matrix p`	Calls `mtxStk.push(mtx)`. Sets `mtx` to `p.matrix()`.
`figure_items::rel_matrix p`	Calls `mtxStk.push(mtx)`. Sets `mtx` to `p.matrix() * mtx`.
`figure_items::revert_-matrix p`	If `mtxStk.empty()` is `false`, sets `mtx` to `mtxStk.top()` then calls `mtxStk.pop()`. Otherwise sets `mtx` to its initial value as specified in Table 17.
`figure_items::abs_line p`	Let `pt` equal `p.to() * mtx`. Creates a line from `currPt` to `pt`. Sets `currPt` to `pt`.
`figure_items::rel_line p`	Let `mm` equal `mtx`. Let `mm.m20` equal 0.0f. Let `mm.m21` equal `0.0f`. Let `pt` equal `currPt + p.to() * mm`. Creates a line from `currPt` to `pt`. Sets `currPt` to `pt`.
`figure_items::abs_-quadratic_curve p`	Let `cpt` equal `p.control_pt() * mtx`. Let `ept` equal `p.end_pt() * mtx`. Creates a quadratic Bézier curve from `currPt` to `ept` using `cpt` as the curve's control point. Sets `currPt` to `ept`.
`figure_items::rel_-quadratic_curve p`	Let `mm` equal `mtx`. Let `mm.m20` equal 0.0f. Let `mm.m21` equal `0.0f`. Let `cpt` equal `currPt + p.control_pt() * mm`. Let `ept` equal `currPt + p.control_pt() * mm + p.end_pt() * mm`. Creates a quadratic Bézier curve from `currPt` to `ept` using `cpt` as the curve's control point. Sets `currPt` to `ept`.
`figure_items::abs_cubic_-curve p`	Let `cpt1` equal `p.control_pt1() * mtx`. Let `cpt2` equal `p.control_pt2() * mtx`. Let `ept` equal `p.end_pt() * mtx`. Creates a cubic Bézier curve from `currPt` to `ept` using `cpt1` as the curve's first control point and `cpt2` as the curve's second control point. Sets `currPt` to `ept`.
`figure_items::rel_cubic_-curve p`	Let `mm` equal `mtx`. Let `mm.m20` equal 0.0f. Let `mm.m21` equal `0.0f`. Let `cpt1` equal `currPt + p.control_pt1() * mm`. Let `cpt2` equal `currPt + p.control_pt1() * mm + p.control_pt2() * mm`. Let `ept` equal `currPt + p.control_pt1() * mm + p.control_pt2() * mm + p.end_pt() * mm`. Creates a cubic Bézier curve from `currPt` to `ept` using `cpt1` as the curve's first control point and `cpt2` as the curve's second control point. Sets `currPt` to `ept`.
`figure_items::arc p`	Let `mm` equal `mtx`. Let `mm.m20` equal 0.0f. Let `mm.m21` equal `0.0f`. Creates an arc. It begins at `currPt`, which is at `p.start_angle()` radians on the arc and rotates `p.rotation()` radians. If `p.rotation()` is positive, rotation is counterclockwise, otherwise it is clockwise. The center of the arc is located at `p.center(currPt, mm)`. The arc ends at `p.end_pt(currPt, mm)`. Sets `currPt` to `p.end_pt(currPt, mm)`. [*Note:* `p.radius()`, which specifies the radius of the arc, is implicitly included in the above statement of effects by the specifications of the center of the arc and the end of the arcs. The use of the current point as the origin for the application of the path transformation matrix is also implicitly included by the same specifications. — *end note*]

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]

```
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 [io2d.pathgroup.cons]

```
basic_interpreted_path() noexcept;
```

1 *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);
```

2 *Effects:* Equivalent to: `basic_interpreted_path{ begin(pb), end(pb) };`.

```
template <class InputIterator>
basic_interpreted_path(InputIterator first, InputIterator last);
```

3 *Effects:* Constructs an object of type `interpreted_path`.

4 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);
```

5 *Effects:* <TODO>

12.5 Class `basic_path_builder` [io2d.pathbuilder]

1 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.

2 A `basic_path_builder` is a contiguous container. (See [container.requirements.general] in C++ 2017.)

3 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 [io2d.pathbuilder.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 size_type              = implementation-defined . // See [container.requirements] in C++ 2017.
    using difference_type        = implementation-defined . // See [container.requirements] in C++ 2017.
    using iterator               = implementation-defined . // See [container.requirements] in C++ 2017.
    using const_iterator         = implementation-defined . // See [container.requirements] in C++ 2017.
    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>);
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
void assign(size_type n, const value_type& u);
void assign(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// 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 insert(const_iterator position, InputIterator first, InputIterator last);
    iterator insert(const_iterator position,
    initializer_list<value_type> il);
    iterator erase(const_iterator position);
    iterator erase(const_iterator first, const_iterator last);
    void swap(basic_path_builder&) noexcept(
      allocator_traits<Allocator>::propagate_on_container_swap::value ||
      allocator_traits<Allocator>::is_always_equal::value);
    void clear() 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;
  };

  template <class GraphicsSurfaces, class Allocator>
  bool operator==(const basic_path_builder<GraphicsSurfaces, Allocator>& lhs,
    const basic_path_builder<GraphicsSurfaces, Allocator>& rhs) noexcept;
  template <class GraphicsSurfaces, class Allocator>
  bool operator!=(const basic_path_builder<GraphicsSurfaces, Allocator>& lhs,
    const basic_path_builder<GraphicsSurfaces, Allocator>& rhs) noexcept;
  template <class GraphicsSurfaces, class Allocator>
  void swap(basic_path_builder<GraphicsSurfaces, Allocator>& lhs,
    basic_path_builder<GraphicsSurfaces, Allocator>& rhs) noexcept(noexcept(lhs.swap(rhs)));
}
```

12.5.2 basic_path_builder container requirements [io2d.pathbuilder.containerrequirements]

1 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 [io2d.pathbuilder.cons]

```
basic_path_builder() noexcept(noexcept(Allocator()));
```

1 *Effects:* Constructs an empty `basic_path_builder`.

```
explicit basic_path_builder(const Allocator&);
```

2 *Effects:* Constructs an empty `basic_path_builder`, using the specified allocator.

3 *Complexity:* Constant.

```
explicit basic_path_builder(size_type n, const Allocator& = Allocator());
```

4 *Effects:* Constructs a `basic_path_builder` with `n` default-inserted elements using the specified allocator.

5 *Complexity:* Linear in `n`.

```
basic_path_builder(size_type n, const value_type& value,
  const Allocator& = Allocator());
```

6 *Requires:* `value_type` shall be `CopyInsertable` into `*this`.

7 *Effects:* Constructs a `basic_path_builder` with n copies of `value`, using the specified allocator.

8 *Complexity:* Linear in `n`.

```
template <class InputIterator>
basic_path_builder(InputIterator first, InputIterator last,
  const Allocator& = Allocator());
```

9 *Effects:* Constructs a `basic_path_builder` equal to the range `[first, last)`, using the specified allocator.

10 *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 [io2d.pathbuilder.capacity]

```
size_type capacity() const noexcept;
```

1 *Returns:* The total number of elements that the path builder can hold without requiring reallocation.

```
void reserve(size_type n);
```

2 *Requires:* `value_type` shall be `MoveInsertable` into `*this`.

3 *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.

4 *Complexity:* It does not change the size of the sequence and takes at most linear time in the size of the sequence.

5 *Throws:* `length_error` if `n > max_size()`.[1]

6 *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();
```

7 *Requires:* `value_type` shall be `MoveInsertable` into `*this`.

8 *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

1) `reserve()` uses `Allocator::allocate()` which may throw an appropriate exception.

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.

9 *Complexity:* Linear in the size of the sequence.

10 *Remarks:* Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. If no reallocation happens, they remain valid.

```
void swap(basic_path_builder&)
  noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
  allocator_traits<Allocator>::is_always_equal::value);
```

11 *Effects:* Exchanges the contents and `capacity()` of `*this` with that of `x`.

12 *Complexity:* Constant time.

resize

```
void resize(size_type sz);
```

13 *Effects:* If `sz < size()`, erases the last `size() - sz` elements from the sequence. Otherwise, appends `sz - size()` default-inserted elements to the sequence.

14 *Requires:* `value_type` shall be `MoveInsertable` and `DefaultInsertable` into `*this`.

15 *Remarks:* If an exception is thrown other than by the move constructor of a non-`CopyInsertable` `value_type` there are no effects.

resize

```
void resize(size_type sz, const value_type& c);
```

16 *Effects:* If `sz < size()`, erases the last `size() - sz` elements from the sequence. Otherwise, appends `sz - size()` copies of `c` to the sequence.

17 *Requires:* `value_type` shall be `CopyInsertable` into `*this`.

18 *Remarks:* If an exception is thrown there are no effects.

12.5.5 `basic_path_builder` modifiers [io2d.pathbuilder.modifiers]

```
void new_figure(point_2d pt) noexcept;
```

1 *Effects:* Adds a `figure_items::figure_item` object constructed from `figure_items::abs_new_figure(pt)` to the end of the path.

```
void rel_new_figure(point_2d pt) noexcept;
```

2 *Effects:* Adds a `figure_items::figure_item` object constructed from `figure_items::rel_new_figure(pt)` to the end of the path.

```
void close_figure() noexcept;
```

3 *Requires:* The current point contains a value.

4 *Effects:* Adds a `figure_items::figure_item` object constructed from `figure_items::close_figure()` to the end of the path.

```
void matrix(const matrix_2d& m) noexcept;
```

5 *Requires:* The matrix `m` shall be invertible.

6 *Effects:* Adds a `figure_items::figure_item` object constructed from `(figure_items::abs_matrix(m)` to the end of the path.

```
void rel_matrix(const matrix_2d& m) noexcept;
```

7 *Requires:* The matrix `m` shall be invertible.

8 *Effects:* Adds a `figure_items::figure_item` object constructed from `(figure_items::rel_matrix(m)` to the end of the path.

```
void revert_matrix() noexcept;
```

9 *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;
```

10 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;
```

11 *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;
```

12 *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;
```

13 *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;
```

14 15*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;
```

16 *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;
```

17 *Effects:* Adds a `figure_items::figure_item` object constructed from `figure_items::arc(rad, rot, sang)` to the end of the path.

```
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 insert(const_iterator position, InputIterator first,
  InputIterator last);
iterator insert(const_iterator position, initializer_list<value_type>);
template <class... Args>
reference emplace_back(Args&&... args);
template <class... Args>
iterator emplace(const_iterator position, Args&&... args);
void push_back(const value_type& x);
void push_back(value_type&& x);
```

18 *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.

19 *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();
```

20 *Effects:* Invalidates iterators and references at or after the point of the erase.

21 *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.

22 *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 [io2d.pathbuilder.iterators]

```
iterator begin() noexcept;
const_iterator begin() const noexcept;
const_iterator cbegin() const noexcept;
```

1 *Returns:* An iterator referring to the first `figure_items::figure_item` item in the path.

2 *Remarks:* Changing a `figure_items::figure_item` object or otherwise modifying the path in a way that violates the preconditions of that `figure_items::figure_item` object or of any subsequent `figure_items::figure_item` object in the path produces undefined behavior when the path is interpreted as described in 12.3.16 unless all of the violations are fixed prior to such interpretation.

```
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cend() const noexcept;
```

3 *Returns:* An iterator which is the past-the-end value.

4 *Remarks:* Changing a `figure_items::figure_item` object or otherwise modifying the path in a way that violates the preconditions of that `figure_items::figure_item` object or of any subsequent `figure_items::figure_item` object in the path produces undefined behavior when the path is interpreted as described in 12.3.16 unless all of the violations are fixed prior to such interpretation.

```
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
const_reverse_iterator crbegin() const noexcept;
```

5 *Returns:* An iterator which is semantically equivalent to `reverse_iterator(end)`.

6 *Remarks:* Changing a `figure_items::figure_item` object or otherwise modifying the path in a way that violates the preconditions of that `figure_items::figure_item` object or of any subsequent `figure_items::figure_item` object in the path produces undefined behavior when the path is interpreted as described in 12.3.16 all of the violations are fixed prior to such interpretation.

```
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
const_reverse_iterator crend() const noexcept;
```

7 *Returns:* An iterator which is semantically equivalent to `reverse_iterator(begin)`.

8 *Remarks:* Changing a `figure_items::figure_item` object or otherwise modifying the path in a way that violates the preconditions of that `figure_items::figure_item` object or of any subsequent `figure_items::figure_item` object in the path produces undefined behavior when the path is interpreted as described in 12.3.16 unless all of the violations are fixed prior to such interpretation.

12.5.7 basic_path_builder specialized algorithms [io2d.pathbuilder.special]

```
template <class Allocator>
void swap(basic_path_builder<Allocator>& lhs, basic_path_builder<Allocator>& rhs)
  noexcept(noexcept(lhs.swap(rhs)));
```

1 *Effects:* As if by `lhs.swap(rhs)`.

13 Brushes [io2d.brushes]

13.1 Overview of brushes [io2d.brushes.general]

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 [io2d.gradients]

13.2.1 Common properties of gradients [io2d.gradients.common]

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 [io2d.gradients.linear]

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.

7 [*Example:* Given a linear gradient with a begin point of <TODO>`basic_point_2d(0.0f, 0.0f)` and an end point of <TODO>`basic_point_2d(10.0f, 5.0f)`, a gradient stop offset value of 0.6f would correspond to the point <TODO>`basic_point_2d(6.0f, 3.0f)`. — *end example*]

8 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 [io2d.gradients.radial]

1 A radial gradient is a type of gradient.

2 A radial gradient has a *start circle* and an *end circle*, each of which is defined by a `basic_circle` object.

3 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 <TODO>`basic_point_2d::zero()`; or,

(3.4) — its start circle has a radius of `0.0f` and its end circle has a radius of `0.0f`.

4 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.

5 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.

6 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.

7 A radial gradient shall be rendered as a continuous series of interpolated circles defined by the following equations:

a) $x(o) = x_{start} + o \times (x_{end} - x_{start})$

b) $y(o) = y_{start} + o \times (y_{end} - y_{start})$

c) $radius(o) = radius_{start} + o \times (radius_{end} - radius_{start})$

where o is a gradient stop offset value.

8 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 [`numeric_limits<float>::lowest(),numeric_limits<float>::max()`].

9 The interpolated circles shall be rendered starting from the smallest potential value of o.

10 An interpolated circle shall not be rendered if its value for o results in $radius(o)$ evaluating to a negative value.

13.2.4 Sampling from gradients [io2d.gradients.sampling]

1 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 `rgba_color::transparent_black` and the remainder of these rules are inapplicable.

b) If exactly one gradient stop has an offset value equal to o, o'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 o, o'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 o 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 `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 `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 *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 *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:

(1.1) — If wrap mode is `wrap_mode::none`, the color value of o shall be `rgba_color::transparent_black`.

(1.2) — 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`.

(1.3) — 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*]

(1.4) — 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 $[0, 2]$, 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*]

2 Interpolation between the color values of two adjacent gradient stops is performed linearly on each color channel.

13.3 Enum class `wrap_mode` [io2d.wrapmode]

13.3.1 `wrap_mode` summary [io2d.wrapmode.summary]

1 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.

2 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`.

3 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 axisvalue and a y axisvalue that is nearest to `0.0f`.

4 See Table 19 for the meaning of each `wrap_mode` enumerator.

13.3.2 `wrap_mode` synopsis [io2d.wrapmode.synopsis]

```
namespace std::experimental::io2d::v1 {
  enum class wrap_mode {
    none,
    repeat,
    reflect,
```

```
    pad
  };
}
```

13.3.3 `wrap_mode` enumerators [io2d.wrapmode.enumerators]

Table 19 — `wrap_mode` enumerator meanings

Enumerator	Meaning
`none`	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 `rgba_color::transparent_black`.
`repeat`	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.
`reflect`	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.
`pad`	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.

13.4 Enum class `filter` [io2d.filter]

13.4.1 `filter` summary [io2d.filter.summary]

1 The `filter` enum class specifies the type of filter to use when sampling from raster graphics data.

2 Three of the `filter` enumerators, `filter::fast`, `filter::good`, and `filter::best`, specify desired characteristics of the filter, leaving the choice of a specific filter to the implementation.

The other two, `filter::nearest` and `filter::bilinear`, each specify a particular filter that shall be used.

3 [*Note:* The only type of brush that has raster graphics data as its visual data is a brush with a brush type of `brush_type::surface`. — *end note*]

4 See Table 20 for the meaning of each `filter` enumerator.

13.4.2 `filter` synopsis [io2d.filter.synopsis]

```
namespace std::experimental::io2d::v1 {
  enum class filter {
    fast,
    good,
    best,
    nearest,
    bilinear
  };
}
```

13.4.3 `filter` enumerators [io2d.filter.enumerators]

Table 20 — `filter` enumerator meanings

Enumerator	Meaning
`fast`	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*]
`good`	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*]
`best`	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*]
`nearest`	Nearest-neighbor interpolation filtering
`bilinear`	Bilinear interpolation filtering

13.5 Enum class `brush_type` [io2d.brushtype]

13.5.1 `brush_type` summary [io2d.brushtype.summary]

1 The `brush_type` enum class denotes the type of a `brush` object.

2 See Table 21 for the meaning of each `brush_type` enumerator.

13.5.2 `brush_type` synopsis [io2d.brushtype.synopsis]

```
namespace std::experimental::io2d::v1 {
  enum class brush_type {
    solid_color,
    surface,
    linear,
    radial
  };
}
```

13.5.3 `brush_type` enumerators [io2d.brushtype.enumerators]

Table 21 — `brush_type` enumerator meanings

Enumerator	Meaning
`solid_color`	The `brush` object is a solid color brush.
`surface`	The `brush` object is a surface brush.
`linear`	The `brush` object is a linear gradient brush.
`radial`	The `brush` object is a radial gradient brush.

13.6 Class `gradient_stop` [io2d.gradientstop]

13.6.1 Overview [io2d.gradientstop.intro]

1 The class `gradient_stop` describes a gradient stop that is used by gradient brushes.

2 It has an *offset* of type `float` and an *offset color* of type `rgba_color`.

13.6.2 gradient_stop synopsis [io2d.gradientstop.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;<TODO><Implementation mismatch>

    // 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 [io2d.gradientstop.cons]

```
constexpr gradient_stop() noexcept;
```

1 *Effects:* Equivalent to: `gradient_stop(0.0f, rgba_color::transparent_black)`.

```
constexpr gradient_stop(float o, rgba_color c) noexcept;
```

2 *Requires:* `o >= 0.0f` and `o <= 1.0f`.

3 *Effects:* Constructs a `gradient_stop` object.

4 The offset is `o` rounded to the nearest multiple of `0.00001f`. The offset color is `c`.

13.6.4 gradient_stop modifiers [io2d.gradientstop.modifiers]

```
constexpr void offset(float o) noexcept;
```

1 *Requires:* `o >= 0.0f` and `o <= 1.0f`.

2 *Effects:* The offset is `o` rounded to the nearest multiple of `0.00001f`.

```
constexpr void color(rgba_color c) noexcept;
```

3 *Effects:* The offset color is `c`.

13.6.5 gradient_stop observers [io2d.gradientstop.observers]

```
constexpr float offset() const noexcept;
```

1 *Returns:* The offset.

```
constexpr rgba_color color() const noexcept;
```

2 *Returns:* The offset color.

13.6.6 gradient_stop operators [io2d.gradientstop.ops]

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

1 *Returns:* `lhs.offset() == rhs.offset() && lhs.color() == rhs.color();`

13.7 Class template basic_brush [io2d.brush]

13.7.1 Summary [io2d.brush.intro]

1 The class template `basic_brush` describes an opaque wrapper for visual data. It takes one type parameter, which is a GraphicsSurfaces.

2 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.

3 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).

4 A `basic_brush` object has a brush type of `brush_type`, which indicates which type of brush it is (Table 21).

5 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.

6 The data are stored in an object of type `typename GraphicsMath::brushes::brush_data_type`.

13.7.2 Synopsis [io2d.brush.synopsis]

```
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 [io2d.brush.sampling]

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 [io2d.brush.sampling.color]

1 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 [io2d.brush.sampling.linear]

1 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 [io2d.brush.sampling.radial]

1 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 [io2d.brush.sampling.surface]

1 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 [io2d.brush.ctor]

```
explicit basic_brush(const rgba_color& c);<TODO><Implementation mismatch> MM: I'm not seeing a mismatch. Am I mis
```

1 *Effects:* Constructs an object of type `basic_brush`.

2 *Postconditions:* `data() == GraphicsSurfaces::brushes::create_brush(c)`.

3 *Remarks:* The visual data format of the visual data are as-if it is that specified by `format::argb32`.

4 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);
```

5 *Effects:* Constructs an object of type `basic_brush`.

6 *Postconditions:* `data() == GraphicsSurfaces::brushes::create_brush(begin, end, first, last)`.

7 *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);
```

8 *Effects:* Constructs an object of type `basic_brush`.

9 *Postconditions:* `data() == GraphicsSurfaces::brushes::create_brush(begin, end, il)`.

10 *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);
```

11 *Effects:* Constructs an object of type `basic_brush`.

12 *Postconditions:* `data() == GraphicsSurfaces::brushes::create_brush(start, end, first, last)`.

13 *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);
```

14 *Effects:* Constructs an object of type `basic_brush`.

15 *Postconditions:* `data() == GraphicsSurfaces::brushes::create_brush(start, end, il)`.

16 *Remarks:* Sampling from this brush produces the results specified in 13.7.3.3.

```
basic_brush(basic_image_surface<GraphicsSurfaces>&& img);
```

17 *Effects:* Constructs an object of type `basic_brush`.

18 *Postconditions:* `data() == GraphicsSurfaces::brushes::create_brush(move(img))`.

19 Sampling from this brush produces the results specified in 13.7.3.4.

13.7.5 Accessors [io2d.brush.acc]

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

1 *Returns:* A reference to the `basic_brush` object's data object (See 13.7.1).

```
brush_type type() const noexcept;
```

2 *Returns:* `GraphicsSurfaces::brushes::get_brush_type(data())`.

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

Enumerator	Meaning
`none`	No anti-aliasing is performed when performing a rendering operation.
`fast`	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.
`good`	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.
`best`	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*]

14.3 Enum class fill_rule [io2d.fillrule]

14.3.1 fill_rule summary [io2d.fillrule.summary]

1 The fill_rule enum class determines how the filling operation (15.3.5) is performed on a path.

2 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.

3 See Table 23 for the meaning of each fill_rule enumerator.

14.3.2 fill_rule synopsis [io2d.fillrule.synopsis]

```
namespace std::experimental::io2d::v1 {
  enum class fill_rule {
    winding,
    even_odd
  };
}
```

14.3.3 fill_rule enumerators [io2d.fillrule.enumerators]

Table 23 — fill_rule enumerator meanings

Enumerator	Meaning
winding	If the *fill rule* (14.10.1) is fill_rule::winding, 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.
even_odd	If the fill rule is fill_rule::even_odd, 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*]

14.4 Enum class line_cap [io2d.linecap]

14.4.1 line_cap summary [io2d.linecap.summary]

1 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 [io2d.linecap.synopsis]

```
namespace std::experimental::io2d::v1 {
  enum class line_cap {
    none,
    round,
    square
  };
}
```

14.4.3 line_cap enumerators [io2d.linecap.enumerators]

Table 24 — `line_cap` enumerator meanings

Enumerator	Meaning
`none`	The line has no cap. It terminates exactly at the end point.
`round`	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.
`square`	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.

14.5 Enum class `line_join` [io2d.linejoin]

14.5.1 `line_join` summary [io2d.linejoin.summary]

1 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 [io2d.linejoin.synopsis]

```
namespace std::experimental::io2d::v1 {
  enum class line_join {
    miter,
    round,
    bevel
  };
}
```

14.5.3 `line_join` enumerators [io2d.linejoin.enumerators]

Table 25 — `line_join` enumerator meanings

Enumerator	Meaning
`miter`	Joins will be mitered or beveled, depending on the miter limit (see: 14.12.1).
`round`	Joins will be rounded, with the center of the circle being the join point.
`bevel`	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.

14.6 Enum class `compositing_op` [io2d.compositingop]

14.6.1 `compositing_op` Summary [io2d.compositingop.summary]

1 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 [io2d.compositingop.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 [io2d.compositingop.enumerators]

1 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 $[0.0, 1.0]$.

2 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.

3 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.

4 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

5 The color symbols R, S, and D may appear with or without any specifiers.

6 If a color symbol appears alone, it designates the entire color as a tuple in the unsigned normalized form (red, green, blue, alpha).

7 The specifiers c and a may appear alone or together after any of the three color symbols.

8 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).

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.

13 The basic enumerators specify a value for *bound*. This value may be 'Yes', 'No', or 'N/A'.

14 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.

15 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.

16 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.

17 If a clip is set and the bound value is 'Yes' or 'N/A', then only those areas of the surface that the are within the clip will be affected by the compositing operation.

18 If a clip is set and the bound value is 'No', then only those areas of the surface that the 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

Enumerator	Bound	Color	Alpha
`clear`	Yes	$Rc = 0$	$Ra = 0$
`source`	Yes	$Rc = Sc$	$Ra = Sa$
`over`	N/A	$Rc = \frac{(Sca + Dca \times (1 - Sa))}{Ra}$	$Ra = Sa + Da \times (1 - Sa)$
`in`	No	$Rc = Sc$	$Ra = Sa \times Da$
`out`	No	$Rc = Sc$	$Ra = Sa \times (1 - Da)$
`atop`	N/A	$Rc = Sca + Dc \times (1 - Sa)$	$Ra = Da$
`dest_over`	N/A	$Rc = \frac{(Sca \times (1 - Da) + Dca)}{Ra}$	$Ra = (1 - Da) \times Sa + Da$
`dest_in`	No	$Rc = Dc$	$Ra = Sa \times Da$
`dest_out`	N/A	$Rc = Dc$	$Ra = (1 - Sa) \times Da$
`dest_atop`	No	$Rc = Sc \times (1 - Da) + Dca$	$Ra = Sa$
`xor_op`	N/A	$Rc = \frac{(Sca \times (1 - Da) + Dca \times (1 - Sa))}{Ra}$	$Ra = Sa + Da - 2 \times Sa \times Da$
`add`	N/A	$Rc = \frac{(Sca + Dca)}{Ra}$	$Ra = min(1, Sa + Da)$
`saturate`	N/A	$Rc = \frac{(min(Sa, 1 - Da) \times Sc + Dca)}{Ra}$	$Ra = min(1, Sa + Da)$

19 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: $Rc = \frac{1}{Ra} \times ((1 - Da) \times Sca + (1 - Sa) \times Dca + Sa \times Da \times f(Sc, Dc))$. The function $f(Sc, Dc)$ is the component of the formula that is enumerator dependent.

20 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.

21 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*]

22 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*]

23 All of the blend enumerators and hsl enumerators have a bound value of 'N/A'.

Table 27 — `compositing_op` blend enumerator meanings

Enumerator	Color
`multiply`	$f(Sc, Dc) = Sc \times Dc$
`screen`	$f(Sc, Dc) = Sc + Dc - Sc \times Dc$

`overlay`	$if(Dc \leq 0.5f)$ {
	$f(Sc, Dc) = 2 \times Sc \times Dc$
	}
	$else$ {
	$f(Sc, Dc) =$
	$1 - 2 \times (1 - Sc) \times$
	$(1 - Dc)$
	}
	[*Note:* The difference between this enumerator and `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)

Enumerator	Color
`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`	$f(Sc, Dc) = abs(Dc - Sc)$
`exclusion`	$f(Sc, Dc) = Sc + Dc - 2 \times Sc \times Dc$

24 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).

25 The following additional functions are used to define the hsl enumerator formulas:

26 $min(x,\ y,\ z)\ =\ min(x,\ min(y,\ z))$

27 $max(x,\ y,\ z)\ =\ max(x,\ max(y,\ z))$

28 $sat(C) = max(Cr,\ Cg,\ Cb) - min(Cr,\ Cg,\ Cb)$

29 $lum(C) = Cr \times 0.3 + Cg \times 0.59 + Cb \times 0.11$

30 $clip\_color(C) =\ \{$

$L = lum(C)$

$N = min(Cr, Cg, Cb)$

$X = max(Cr, Cg, Cb)$

$if\ (N < 0.0)\ \{$

$$Cr = L + \frac{((Cr - L) \times L)}{(L - N)}$$

$$Cg = L + \frac{((Cg - L) \times L)}{(L - N)}$$

$$Cb = L + \frac{((Cb - L) \times L)}{(L - N)}$$

$\}$

$if\ (X > 1.0)\ \{$

$$Cr = L + \frac{((Cr - L) \times (1 - L))}{(X - L)}$$

$$Cg = L + \frac{((Cg - L) \times (1 - L))}{(X - L)}$$

$$Cb = L + \frac{((Cb - L) \times (1 - L))}{(X - L)}$$

$\}$

$return\ C$
}

31 $set\_lum(C, L) = \{$
$D = L - lum(C)$
$Cr = Cr + D$
$Cg = Cg + D$
$Cb = Cb + D$
$return\ clip\_color(C)$
}

32 $set\_sat(C, S) = \{$
$R = C$
$auto\&\ max = (Rr > Rg)\ ?\ ((Rr > Rb)\ ?\ Rr : Rb) : ((Rg > Rb)\ ?\ Rg : Rb)$
$auto\&\ mid = (Rr > Rg)\ ?\ ((Rr > Rb)\ ?\ ((Rg > Rb)\ ?\ Rg : Rb) : Rr) : ((Rg > Rb)\ ?\ ((Rr > Rb)\ ?\ Rr : Rb) : Rg)$
$auto\&\ min = (Rr > Rg)\ ?\ ((Rg > Rb)\ ?\ Rb : Rg) : ((Rr > Rb)\ ?\ Rb : Rr)$
$if\ (max > min)\ \{$
$$mid = \frac{((mid - min) \times S)}{max - min}$$
$max = S$
}
$else\ \{$
$mid = 0.0$
$max = 0.0$
}
$min = 0.0$
$return\ R$
} [*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 28 — `compositing_op` hsl enumerator meanings

Enumerator	Color & Alpha
`hsl_hue`	$f(Sc, Dc) = set\_lum(set\_sat(Sc,\ sat(Dc)),\ lum(Dc))$
`hsl_saturation`	$(Sc, Dc) = set\_lum(set\_sat(Dc, sat(Sc)),\ lum(Dc))$
`hsl_color`	$f(Sc, Dc) = set\_lum(Sc,\ lum(Dc))$
`hsl_luminosity`	$f(Sc, Dc) = set\_lum(Dc,\ lum(Sc))$

14.7 Enum class `format` [io2d.format]

14.7.1 Summary [io2d.format.summary]

1 The `format` enum class indicates a visual data format. See Table 29 for the meaning of each `format` enumerator.

14.7.2 Synopsis [io2d.format.synopsis]

```
namespace std::experimental::io2d::v1 {
  enum class format {
    invalid,
    argb32,
    xrgb32,
    xrgb16,
    a8
  };
}
```

14.7.3 Enumerators [io2d.format.enumerators]

Table 29 — `format` enumerator meanings

Enumerator	Meaning
`invalid`	A previously requested `format` is unsupported by the implementation.
`argb32`	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.
`xrgb32`	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.
`xrgb16`	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.
`a8`	An 8-bit transparency data pixel format. All 8 bits are an alpha channel.

1 Implementations may support additional visual data formats (See: 8.2.3).

14.8 Enum class `scaling` [io2d.scaling]

14.8.1 `scaling` summary [io2d.scaling.summary]

1 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 [io2d.scaling.synopsis]

```
namespace std::experimental::io2d::v1 {
  enum class scaling {
    letterbox,
    uniform,
    fill_uniform,
    fill_exact,
    none
  };
}
```

14.8.3 `scaling` enumerators [io2d.scaling.enumerators]

1 [*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 $1.\overline{3}$.

The display buffer (**??**) of `ds` is 1280x720, giving it an aspect ratio of $1.\overline{7}$.

When a rectangle is defined in an example, the coordinate $(x1, y1)$ denotes the top left corner of the rectangle, inclusive, and the coordinate $(x2, y2)$ denotes the bottom right corner of the rectangle, exclusive. As such, a rectangle with $(x1, y1) = (10, 10)$, $(x2, y2) = (20, 20)$ 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

Enumerator	Meaning
`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 transfered to the display buffer using sampling such that it is in the rectangle $(x1, y1) = (\frac{1280}{2} - \frac{960}{2}, 0) = (160, 0)$, $(x2, y2) = (960 + (\frac{1280}{2} - \frac{960}{2}), 720) = (1120, 720)$. 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). *— end example*]

Table 30 — `scaling` enumerator meanings (continued)

Enumerator	Meaning
`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 transfered to the display buffer using sampling such that it is in the rectangle $(x1, y1) = (\frac{1280}{2} - \frac{960}{2}, 0) = (160, 0)$, $(x2, y2) = (960 + (\frac{1280}{2} - \frac{960}{2}), 720) = (1120, 720)$. 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). — *end example*] [*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 note*]
`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 $(x1, y1) = (0, -120)$, $(x2, y2) = (1280, 840)$. 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)

Enumerator	Meaning
`fill_exact`	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 scaled back buffer to the display buffer using sampling such that its origin is at the origin of the display buffer. [*Example:* The back buffer will be drawn in the rectangle $(x1, y1) = (0, 0)$, $(x2, y2) = (1280, 720)$. This fulfills all of the conditions. Each 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 and a height of 720 pixels) and the origin of the scaled back buffer is at the origin of the display buffer. — *end example*]
`none`	Do not perform any scaling. Transfer the back buffer to the display buffer using sampling such that its origin is at the origin of the display buffer. [*Example:* The back buffer of `ds` will be drawn in the rectangle $(x1, y1) = (0, 0)$, $(x2, y2) = (640, 480)$ such that no scaling occurs and the origin of the back buffer is at the origin of the display buffer. — *end example*]

14.9 Class template `basic_render_props` [io2d.renderprops]

14.9.1 Overview [io2d.renderprops.intro]

1 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 [io2d.renderprops.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;
```

```
    basic_matrix_2d<graphics_math_type> surface_matrix() const noexcept;
  };
}
```

14.9.3 Constructors [io2d.renderprops.ctor]

```
basic_render_props() noexcept;
```

1 *Effects:* Constructs an object of type `basic_render_props`.

2 *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;
```

3 *Requires:* `m.is_invertible() == true`.

4 *Effects:* Constructs an object of type `basic_render_props`.

5 *Postconditions:* `data() == GraphicsSurfaces::surface_state_props::create_render_props(f, m, co)`.

14.9.4 Accessors [io2d.renderprops.acc]

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

1 *Returns:* A reference to the `basic_render_props` object's data object (See: 14.9.1).

14.9.5 Modifiers [io2d.renderprops.mod]

```
void filtering(filter f) noexcept;
```

1 *Effects:* Calls `GraphicsSurfaces::surface_state_props::filtering(data(), f))`.

Remarks: The filtering algorithm is `f`.

```
void compositing(compositing_op co) noexcept;
```

2 *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;
```

3 *Requires:* `m.is_invertible() == true`.

4 *Effects:* Calls `GraphicsSurfaces::surface_state_props::surface_matrix(data(), m)`.

5 *Remarks:* The surface matrix is `m`.

14.9.6 Observers [io2d.renderprops.obs]

```
filter filtering() const noexcept;
```

1 *Returns:* `GraphicsSurfaces::surface_state_props::filtering(data())`.

2 *Remarks:* The returned value is the filter algorithm.

```
compositing_op compositing() const noexcept;
```

3 *Returns:* `GraphicsSurfaces::surface_state_props::compositing(data())`.

4 *Remarks:* The returned value is the compositing operator.

```
basic_matrix_2d<graphics_math_type> surface_matrix() const noexcept;
```

5 *Returns:* `GraphicsSurfaces::surface_state_props::surface_matrix(data())`.

6 *Remarks:* The returned value is the surface matrix.

14.10 Class basic_brush_props [io2d.brushprops]

14.10.1 basic_brush_props summary [io2d.brushprops.summary]

1 The `basic_brush_props` class provides general state information that is applicable to all rendering and composing operations (15.3.2).

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 [io2d.brushprops.synopsis]

```
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 [io2d.brushprops.cons]

```
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 [io2d.brushprops.modifiers]

```
void wrap_mode(io2d::wrap_mode w) noexcept;
```

1 *Effects:* The wrap mode is `w`.

```
void filter(io2d::filter fi) noexcept;
```

2 *Effects:* The filter is `fi`.

```
void fill_rule(io2d::fill_rule fr) noexcept;
```

3 *Effects:* The fill rule is `fr`.

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

4 *Requires:* `m.is_invertible() == true`.

5 *Effects:* The brush matrix is `m`.

14.10.5 basic_brush_props observers [io2d.brushprops.observers]

```
io2d::wrap_mode wrap_mode() const noexcept;
```

1 *Returns:* The wrap mode.

```
io2d::filter filter() const noexcept;
```

2 *Returns:* The filter.

```
io2d::fill_rule fill_rule() const noexcept;
```

3 *Returns:* The fill rule.

```
basic_matrix_2d<graphics_math_type> brush_matrix() const noexcept;
```

4 *Returns:* The brush matrix.

14.11 Class template basic_clip_props [io2d.clipprops]

14.11.1 Overview [io2d.clipprops.intro]

1 The `basic_clip_props` class template provides general state information that is applicable to all rendering and composing operations (15.3.2).

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.

3 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 [io2d.clipprops.synopsis]

```
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 [io2d.clipprops.ctor]

```
basic_clip_props() noexcept;
```

1 *Effects:* Constructs an object of type basic_clip_props.

2 *Postconditions:* data() == GraphicsSurfaces::surface_state_props::create_clip_props().

3 *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);
```

4 *Effects:* Constructs an object of type basic_clip_props.

5 *Postconditions:* data() == GraphicsSurfaces::surface_state_props::create_clip_props(pb, fr).

6 *Remarks:* The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(basic_interpreted_path<GraphicsSurfaces>(pb)).

7 The fill rule is fr.

```
template <class InputIterator>
basic_clip_props(InputIterator first, InputIterator last,
  io2d::fill_rule fr = io2d::fill_rule::winding);
```

8 *Effects:* Constructs an object of type basic_clip_props.

9 *Postconditions:* data() == GraphicsSurfaces::surface_state_props::create_clip_props(first, last, fr).

10 *Remarks:* The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(basic_interpreted_path<GraphicsSurfaces>(first, last)).

11 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);
```

12 *Effects:* Constructs an object of type basic_clip_props.

13 *Postconditions:* data() == GraphicsSurfaces::surface_state_props::create_clip_props(il, fr).

14 *Remarks:* The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(basic_interpreted_path<GraphicsSurfaces>(il)).

15 The fill rule is fr.

```
explicit basic_clip_props(const basic_interpreted_path<GraphicsSurfaces>& ip,
  io2d::fill_rule fr = io2d::fill_rule::winding) noexcept;
```

16 *Effects:* Constructs an object of type basic_clip_props.

17 *Postconditions:* data() == GraphicsSurfaces::surface_state_props::create_clip_props(ip, fr).

18 *Remarks:* The clip area is optional<basic_interpreted_path<GraphicsSurfaces>(ip.

19 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)
```

20 *Effects:* Constructs an object of type basic_clip_props.

21 *Postconditions:* data() == GraphicsSurfaces::surface_state_props::create_clip_props(r, fr).

22 *Remarks:* The clip area is optional<basic_interpreted_path<GraphicsSurfaces>>(basic_interpreted_path<GraphicsSurfaces>(r));

23 The fill rule is fr.

14.11.4 Accessors [io2d.clipprops.acc]

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

1 *Returns:* A reference to the `basic_clip_props` object's data object (See: 14.11.1).

14.11.5 `basic_clip_props` modifiers [io2d.clipprops.mod]

```
template <class Allocator>
void clip();
```

1 *Effects:* The clip area is `optional<basic_interpreted_path<GraphicsSurfaces>>()`.

```
void clip(const basic_bounding_box<GraphicsSurfaces>& bb);
```

2 *Effects:* Calls `GraphicsSurfaces::surface_state_props::clip(data(), bb);`

3 *Remarks:* The clip area is `optional<basic_interpreted_path<GraphicsSurfaces>>(basic_interpreted_path<GraphicsSurfaces>(bb))`.

```
template <class Allocator>
void clip(const basic_path_builder<GraphicsSurfaces, Allocator>& pb);
```

4 *Effects:* Calls `GraphicsSurfaces::surface_state_props::clip(data(), pb);`

5 *Remarks:* The clip area is `optional<basic_interpreted_path<GraphicsSurfaces>>(basic_interpreted_path<GraphicsSurfaces>(pb))`.

```
template <class InputIterator>
void clip(InputIterator first, InputIterator last);
```

6 *Effects:* Calls `GraphicsSurfaces::surface_state_props::clip(data(), first, last);`

7 *Remarks:* The clip area is `optional<basic_interpreted_path<GraphicsSurfaces>>(basic_interpreted_path<GraphicsSurfaces>(first, last))`.

```
void clip(
  const initializer_list<typename
  basic_figure_items<GraphicsSurfaces>::figure_item> il);
```

8 *Effects:* Calls `GraphicsSurfaces::surface_state_props::clip(data(), il);`

9 *Remarks:* The clip area is `optional<basic_interpreted_path<GraphicsSurfaces>>(basic_interpreted_path<GraphicsSurfaces>(il))`.

```
void clip(const basic_bounding_box<GraphicsSurfaces>& bb);
```

10 *Effects:* Calls `GraphicsSurfaces::surface_state_props::clip(data(), ip);`

11 *Remarks:* The clip area is `optional<basic_interpreted_path<GraphicsSurfaces>>(ip)`.

```
void fill_rule(experimental::io2d::fill_rule fr) noexcept;
```

12 *Effects:* Calls `GraphicsSurfaces::surface_state_props::clip_fill_rule(fr)`.

13 *Remarks:* The fill rule is `fr`.

14.11.6 `basic_clip_props` observers [io2d.clipprops.obs]

```
optional<basic_interpreted_path<GraphicsSurfaces>> clip() const noexcept;
```

1 *Returns:* `GraphicsSurfaces::surface_state_props::clip(data())`.

2 *Remarks:* The return value is the clip area.

```
io2d::fill_rule fill_rule() const noexcept;
```

3 *Returns:* `GraphicsSurfaces::surface_state_props::clip_fill_rule(data())`.

4 *Remarks:* The return value is the fill rule.

14.12 Class `basic_stroke_props` [io2d.strokeprops]

14.12.1 `basic_stroke_props` summary [io2d.strokeprops.summary]

1 The `basic_stroke_props` class provides state information that is applicable to the stroking operation (see: 15.3.2 and 15.3.6).

2 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 [io2d.strokeprops.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;
    float max_miter_limit() const noexcept;
  };
}
```

14.12.3 `basic_stroke_props` constructors [io2d.strokeprops.cons]

```
basic_stroke_props() noexcept;
```

1 *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;
```

2 *Requires:* `w > 0.0f`. `ml >= 10.0f`. `ml <= max_miter_limit()`.

3 *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 [io2d.strokeprops.modifiers]

```
void line_width(float w) noexcept;
```

1 *Requires:* `w >= 0.0f`.

2 *Effects:* The line width is `w`.

```
void line_cap(io2d::line_cap lc) noexcept;
```

3 *Effects:* The line cap is `lc`.

```
void line_join(io2d::line_join lj) noexcept;
```

4 *Effects:* The line join is `lj`.

```
void miter_limit(float ml) noexcept;
```

5 *Requires:* `ml >= 1.0f` and `ml <= max_miter_limit`.

6 *Effects:* The miter limit is `ml`.

14.12.5 `basic_stroke_props` observers [io2d.strokeprops.observers]

```
float line_width() const noexcept;
```

1 *Returns:* The line width.

```
io2d::line_cap line_cap() const noexcept;
```

2 *Returns:* The line cap.

```
io2d::line_join line_join() const noexcept;
```

3 *Returns:* The line join.

```
float miter_limit() const noexcept;
```

4 *Returns:* The miter limit.

```
float max_miter_limit() const noexcept;
```

5 *Requires:* This value shall be finite and greater than `10.0f`.

6 *Returns:* The implementation-defined maximum value of miter limit.

14.13 Class template `basic_fill_props` [io2d.fillprops]

14.13.1 Overview [io2d.fillprops.intro]

1 The `basic_fill_props` class template provides state information that is applicable to the filling rendering and composing operation (15.3.2).

2 It has a *fill rule* of type `fill_rule` and an *antialiasing algorithm* of type `antialias`.

3 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 [io2d.fillprops.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 [io2d.fillprops.ctor]

```
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 [io2d.fillprops.acc]

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

1 *Returns:* A reference to the `basic_fill_props` object's data object (See: 14.13.1).

14.13.5 Modifiers [io2d.fillprops.mod]

```
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.fillprops.obs]

```
io2d::fill_rule fill_rule() const noexcept;
```

1 *Returns:* `GraphicsSurfaces::surface_state_props::fill_fill_rule(data())`.

2 *Remarks:* The returned value is the fill rule.

```
antialias antialiasing() const noexcept;
```

3 *Returns:* `GraphicsSurfaces::surface_state_props::antialiasing(data())`.

4 *Remarks:* The returned value is the antialiasing algorithm.

14.14 Class `basic_mask_props` [io2d.maskprops]

14.14.1 `basic_mask_props` summary [io2d.maskprops.summary]

1 The `basic_mask_props` class provides state information that is applicable to the mask rendering and composing operation (15.3.2).

2 It has a *wrap mode* of type `wrap_mode`, a *filter* of type `filter`, and a *mask matrix* of type `matrix_2d`.

14.14.2 `basic_mask_props` synopsis [io2d.maskprops.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;
```

```
    // 14.14.4, 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 [io2d.maskprops.cons]

```
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`.

1 *Effects:* The wrap mode is `w`. The filter is `fi`. The mask matrix is `m`.

14.14.4 basic_mask_props modifiers [io2d.maskprops.modifiers]

```
void wrap_mode(io2d::wrap_mode w) noexcept;
```

1 *Effects:* The wrap mode is `w`.

```
void filter(io2d::filter fi) noexcept;
```

2 *Effects:* The filter is `fi`.

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

3 *Requires:* `m.is_invertible() == true`.

4 *Effects:* The mask matrix is `m`.

14.14.5 basic_mask_props observers [io2d.maskprops.observers]

```
io2d::wrap_mode wrap_mode() const noexcept;
```

1 *Returns:* The wrap mode.

```
io2d::filter filter() const noexcept;
```

2 *Returns:* The filter.

```
basic_matrix_2d<graphics_math_type> mask_matrix() const noexcept;
```

3 *Returns:* The mask matrix.

14.15 Class template basic_dashes [io2d.dashes]

14.15.1 basic_dashes class template [io2d.dashes.intro]

1 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.

2 It has an *offset* of type `float` and a *pattern* of an ***`unspecified`*** type capable of sequentially storing floating-point values.

3 The data are stored in an object of type `typename GraphicsSurfaces::surface_props_data::dashes_props_data_type`. It is accessible using the `data` member function.

14.15.2 Synopsis [io2d.dashes.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 [io2d.dashes.cons]

```
basic_dashes() noexcept;
```

1 *Effects:* Constructs an object of type `basic_dashes`.

2 *Postconditions:* `data() == GraphicsSurfaces::surface_state_props::create_dashes()`.

3 *Remarks:* The offset is `0.0f` and the pattern contains no values.

```
template <class InputIterator>
basic_dashes(float o, InputIterator first, InputIterator last);
```

4 *Requires:* The value type of `InputIterator` is `float`.

5 Each value from `first` through `last - 1` is greater than or equal to `0.0f`.

6 *Effects:* Constructs an object of type `basic_dashes`.

7 *Postconditions:* `data() == GraphicsSurfaces::surface_state_props::create_dashes(o, first, last)`.

8 *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);
```

9 *Requires:* Each value in `il` is greater than or equal to `0.0f`.

10 *Effects:* Constructs an object of type `basic_dashes`.

11 *Postconditions:* `data() == GraphicsSurfaces::surface_state_props::create_dashes(o, il)`.

14.15.4 Observers [io2d.dashes.observers]

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

1 *Returns:* A reference to the `basic_dashes` object's data object (See 14.15.1).

14.15.5 Operators [io2d.dashes.ops]

1

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

2 *Returns:* `GraphicsSurfaces::surface_state_props::equal(lhs.data(), rhs.data())`.

3

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

4 *Returns:* `GraphicsSurfaces::surface_state_props::not_equal(lhs.data(), rhs.data())`.

15 Surfaces [io2d.surfaces]

15.1 Enum class `refresh_style` [io2d.refreshstyle]

15.1.1 `refresh_style` summary [io2d.refreshstyle.summary]

1 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 [io2d.refreshstyle.synopsis]

```
namespace std::experimental::io2d::v1 {
  enum class refresh_style {
    as_needed,
    as_fast_as_possible,
    fixed
  };
}
```

15.1.3 `refresh_style` enumerators [io2d.refreshstyle.enumerators]

Table 31 — `refresh_style` value meanings

Enumerator	Meaning
`as_needed`	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*]
`as_fast_as_possible`	The draw callback shall be called as frequently as possible, subject to any limits of the execution environment.

Table 31 — `refresh_style` value meanings (continued)

Enumerator	Meaning
`fixed`	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*]

15.2 Enum class `image_file_format` [io2d.imagefileformat]

15.2.1 `image_file_format` summary [io2d.imagefileformat.summary]

1 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 [io2d.imagefileformat.synopsis]

```
namespace std::experimental::io2d::v1 {
  enum class image_file_format {
    unknown,
    png,
    jpeg,
    tiff
  };
}
```

15.2.3 `image_file_format` enumerators [io2d.imagefileformat.enumerators]

Table 32 — `imagefileformat` enumerator meanings

Enumerator	Meaning
`unknown`	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.
`png`	The PNG format.

Table 32 — `imagefileformat` enumerator meanings (continued)

Enumerator	Meaning
`jpeg`	The JPEG format.
`tiff`	The TIFF format.

15.3 Overview of surface classes [io2d.surface]

15.3.1 Surface class templates description [io2d.surface.intro]

1 There are three *surface class templates*:

(1.1) — `basic_image_surface`

(1.2) — `basic_output_surface`

(1.3) — `basic_unmanaged_output_surface`

2 For ease of description, an instantiation of a surface class template will be called a *surface*.

3 A surface contains visual data and provides an interface for managing and manipulating that visual data.

4 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*]

5 The surface class templates manipulate visual data through rendering and composing operations.

6 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 [io2d.surface.rendering]

15.3.2.1 Operations [io2d.surface.rendering.ops]

1 The surface classes provide four fundamental rendering and composing operations:

Table 33 — surface rendering and composing operations

Operation	Function(s)
Painting	`paint`
Filling	`fill`
Stroking	`stroke`
Masking	`mask`

2 All composing operations shall happen as-if in a linear color space, regardless of the color space of the visual data that is involved.

3 [*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 [io2d.surface.rendering.brushes]

1 All rendering and composing operations use a *source brush* of type `basic_brush`.

2 The masking operation uses a *mask brush* of type `basic_brush`.

15.3.2.3 Rendering and composing source path [io2d.surface.rendering.sourcepath]

1 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 [io2d.surface.rendering.commonstate]

1 All rendering and composing operations use the following state data:

Table 34 — `surface` rendering and composing common state data

Name	Type
Brush properties	`brush_props`
Surface properties	`render_props`
Clip properties	`clip_props`

15.3.2.5 Specific state data [io2d.surface.rendering.specificstate]

1 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

Operation	Name	Type
Stroking	Stroke properties	`stroke_props`
Stroking	Dashes	`dashes`
Masking	Mask properties	`mask_props`

15.3.2.6 State data default values [io2d.surface.rendering.statedefaults]

1 For all rendering and composing operations, the state data objects named above are provided using `optional<T>` class template arguments.

2 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 [io2d.surface.coordinatespaces]

1 There are four standard coordinate spaces relevant to the rendering and composing operations (15.3.2):

(1.1) — the brush coordinate space;

(1.2) — the mask coordinate space;

(1.3) — the user coordinate space; and

(1.4) — the surface coordinate space.

2 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).

3 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).

4 The *user coordinate space* is the standard coordinate space of `basic_interpreted_path` objects. Its transformation matrix is a default-constructed `basic_matrix_2d`.

5 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).

6 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

From	To	Transform
brush coordinate space	mask coordinate space	`mcsm.transform_-pt(bcsm.invert().transform_-pt(pt)).`
brush coordinate space	user coordinate space	`bcsm.invert().transform_pt(pt).`
brush coordinate space	surface coordinate space	`scsm.transform_-pt(bcsm.invert().transform_-pt(pt)).`
user coordinate space	brush coordinate space	`bcsm.transform_pt(pt).`
user coordinate space	mask coordinate space	`mcsm.transform_pt(pt).`
user coordinate space	surface coordinate space	`scsm.transform_pt(pt).`
surface coordinate space	brush coordinate space	`bcsm.transform_-pt(scsm.invert().transform_-pt(pt)).`
surface coordinate space	mask coordinate space	`mcsm.transform_-pt(scsm.invert().transform_-pt(pt)).`
surface coordinate space	user coordinate space	`scsm.invert().transform_pt(pt).`

15.3.4 surface painting [io2d.surface.painting]

1 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 [io2d.surface.filling]

1 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 [io2d.surface.stroking]

1 When a stroking operation is initiated on a surface, it is carried out for each figure in the source path (15.3.2).

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.

3 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*]

4 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 [io2d.surface.masking]

1 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 [io2d.outputsurface.misc]

1 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*]

2 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 [io2d.outputsurface.state]

1 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

Name	Type	Function	Default value
Letterbox brush	`brush`	This is the brush that shall be used as specified by `scaling::letterbox` (Table 30)	`brush{ { rgba_color::black } }`
Letterbox brush props	`brush_props`	This is the brush properties for the letterbox brush	`brush_props{ }`
Scaling type	`scaling`	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	`scaling::letterbox`
Draw width	`int`	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	*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*]

Table 37 — Output surface observable state (continued)

Name	Type	Function	Default value
Draw height	`int`	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	*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*]
Draw format	`format`	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	*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*]

Table 37 — Output surface observable state (continued)

Name	Type	Function	Default value
Display width	`int`	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	*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*]

Table 37 — Output surface observable state (continued)

Name	Type	Function	Default value
Display height	`int`	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 otherwise 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	*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*]
Auto clear	`bool`	If `true` the implementation shall call `clear`, which shall clear the back buffer, immediately before it executes the draw callback	`false`
Refresh style	`refresh_style`	The `refresh_style` value that determines when the draw callback shall be called while `basic_output_surface<T>::begin_show` is being executed	`refresh_style::as_fast_as_possible`
Desired frame rate	`float`	This value is the number of times the draw callback shall be called per second while `basic_output_surface<T>::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).	`30.0f`

15.4 Class basic_image_surface [io2d.imagesurface]

15.4.1 basic_image_surface summary [io2d.imagesurface.summary]

1 The class `basic_image_surface` provides an interface to raster graphics data.

2 It has a *pixel format* of type `format`, a *width* of type `int`, and a *height* of type `int`.

3 [*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 [io2d.imagesurface.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_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);
    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);
  };
}
```

15.4.3 `basic_image_surface` constructors and assignment operators [io2d.imagesurface.cons]

```
basic_image_surface(io2d::format fmt, int w, int h);
```

1 *Requires:* `w` is greater than `0` and not greater than `basic_image_surface::max_width()`.

2 `h` is greater than `0` and not greater than `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 `basic_image_surface::max_width()` or if the height of the uncompressed raster graphics data would be less than `1` or greater than `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).

17 *Error conditions:* Any error that could result from trying to access `f`, open `f` for reading, or reading data from `f`.

18 `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.

19 `errc::invalid_argument` if `fmt` is `io2d::format::invalid`.

20 `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.mofifiers]

```
void clear();
```

1 *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);
```

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).

5 *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);
```

6 *Effects:* Performs the stroking rendering and composing operation as specified by 15.3.6.

7 The meanings of the parameters are specified by 15.3.2.

8 *Throws:* As specified in Error reporting (Clause 4).

9 *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);
```

10 *Effects:* Performs the filling rendering and composing operation as specified by 15.3.5.

11 The meanings of the parameters are specified by 15.3.2.

12 *Throws:* As specified in Error reporting (Clause 4).

13 *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);
```

14 *Effects:* Performs the masking rendering and composing operation as specified by 15.3.7.

15 The meanings of the parameters are specified by 15.3.2.

16 *Throws:* As specified in Error reporting (Clause 4).

17 *Error conditions:*

The errors, if any, produced by this function are implementation-defined.

15.5 Class `basic_output_surface` [io2d.outputsurface]

15.5.1 `basic_output_surface` summary [io2d.outputsurface.summary]

1 <TODO>

15.5.2 `basic_output_surface` synopsis [io2d.outputsurface.synopsis]

```
namespace std::experimental::io2d::v1 {
  template <class GraphicsSurfaces>
```

```
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);
  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_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;
    optional<basic_brush<GraphicsSurfaces>> letterbox_brush() const noexcept;
    optional<basic_brush_props<GraphicsSurfaces>> letterbox_brush_props() const noexcept;
    bool auto_clear() const noexcept;
  };
}
```

15.5.3 basic_output_surface constructors [io2d.outputsurface.cons]

```
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;
```

1 <TODO>

```
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;
```

2 <TODO>

15.5.4 `basic_output_surface` modifiers [io2d.outputsurface.mofifiers]

`int begin_show();`

1 *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.

2 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.

3 No later than when this function returns, the output device shall cease to display the contents of the display buffer.

4 What the output device shall display when it is not displaying the contents of the display buffer is unspecified .

5 *Returns:* The possible values and meanings of the possible values returned are implementation-defined.

6 *Throws:* As specified in Error reporting (Clause 4).

7 *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.

8 *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.

9 Other errors, if any, produced by this function are implementation-defined.

`void end_show();`

10 *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.

11 Otherwise, the implementation initiates the process of exiting the `basic_output_surface::begin_show` function's continuous loop.

12 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.

13 The `basic_output_surface::begin_show` function's loop continues execution until it returns.

`void clear();`

14 *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);
```

15 *Effects:* Performs the painting rendering and composing operation as specified by 15.3.4.

16 The meanings of the parameters are specified by 15.3.2.

17 *Throws:* As specified in Error reporting (Clause 4).

18 *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);
```

19 *Effects:* Performs the stroking rendering and composing operation as specified by 15.3.6.

20 The meanings of the parameters are specified by 15.3.2.

21 *Throws:* As specified in Error reporting (Clause 4).

22 *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);
```

23 *Effects:* Performs the filling rendering and composing operation as specified by 15.3.5.

24 The meanings of the parameters are specified by 15.3.2.

25 *Throws:* As specified in Error reporting (Clause 4).

26 *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);
```

27 *Effects:* Performs the masking rendering and composing operation as specified by 15.3.7.

28 The meanings of the parameters are specified by 15.3.2.

29 *Throws:* As specified in Error reporting (Clause 4).

30 *Error conditions:*

The errors, if any, produced by this function are implementation-defined.

```
void draw_callback(const function<void(basic_output_surface& sfc)>& fn);
```

31 *Effects:* <TODO>

```
void size_change_callback(const function<void(basic_output_surface& sfc)>& fn);
```

32 *Effects:* <TODO>

```
void dimensions(basic_display_point<graphics_math_type> dp);
void dimensions(basic_display_point<graphics_math_type> dp, error_code& ec) noexcept;
```

33 *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;
```

34 *Effects:* <TODO>

```
void scaling(io2d::scaling scl) noexcept;
```

35 *Effects:* <TODO>

```
void user_scaling_callback(const
  function<basic_bounding_box<graphics_math_type>(const basic_output_surface&, bool&)>& fn);
```

36 *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;
```

37 *Effects:* <TODO>

```
void auto_clear(bool val) noexcept;
```

38 *Effects:* <TODO>

```
void redraw_required(bool val = true) noexcept;
```

39 *Effects:* <TODO>

15.5.5 `basic_output_surface` observers [io2d.outputsurface.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.

```
basic_display_point<graphics_math_type> max_dimensions() const noexcept;
```

3 *Returns:* <TODO>The maximum available height and width of a `basic_output_surcace` for the device.

```
basic_display_point<graphics_math_type> display_dimensions() const noexcept;
```

4 *Returns:* <TODO>

```
basic_display_point<graphics_math_type> max_output_dimensions() const noexcept;
```

5 *Returns:* <TODO>

```
io2d::scaling scaling() const noexcept;
```

6 *Returns:* The scaling type.

```
optional<basic_brush<GraphicsSurfaces>> letterbox_brush() const noexcept;
```

7 *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;
```

8 *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_clear() const noexcept;
```

9 *Returns:* The value of auto clear.

15.6 Class basic_unmanaged_output_surface [io2d.unmanagedoutputsurface]

15.6.1 basic_unmanaged_output_surface summary [io2d.unmanagedoutputsurface.summary]

1 <TODO>

15.6.2 basic_unmanaged_output_surface synopsis [io2d.unmanagedoutputsurface.synopsis]

```
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_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);
    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>

```
basic_display_point<graphics_math_type> max_display_dimensions() const noexcept;
```

7 *Returns:* <TODO>

```
io2d::scaling scaling() const noexcept;
```

8 *Returns:* <TODO>

```
optional<basic_brush<GraphicsSurfaces>> letterbox_brush() const noexcept;
```

9 *Returns:* <TODO>

```
optional<basic_brush_props<GraphicsSurfaces>> letterbox_brush_props() const
  noexcept;
```

10 *Returns:* <TODO>

```
bool auto_clear() const noexcept;
```

11 *Returns:* <TODO>

15.6.5 `basic_unmanaged_output_surface` modifiers [io2d.unmanagedoutputsurface.mofifiers]

```
void clear();
```

1 *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);
```

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).

5 *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);
```

6 *Effects:* Performs the stroking rendering and composing operation as specified by 15.3.6.

7 The meanings of the parameters are specified by 15.3.2.

8 *Throws:* As specified in Error reporting (Clause 4).

9 *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);
```

10 *Effects:* Performs the filling rendering and composing operation as specified by 15.3.5.

11 The meanings of the parameters are specified by 15.3.2.

12 *Throws:* As specified in Error reporting (Clause 4).

13 *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);
```

14 *Effects:* Performs the masking rendering and composing operation as specified by 15.3.7.

15 The meanings of the parameters are specified by 15.3.2.

16 *Throws:* As specified in Error reporting (Clause 4).

17 *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);
```

18 *Effects:* <TODO>

```
void size_change_callback(const function<void(basic_unmanaged_output_surface& sfc)>& fn);
```

19 *Effects:* <TODO>

```
void dimensions(basic_display_point<graphics_math_type> dp);
void dimensions(basic_display_point<graphics_math_type> dp, error_code& ec) noexcept;
```

20 *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;
```

21 *Effects:* <TODO>

```
void scaling(io2d::scaling scl) noexcept;
```

22 *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;
```

23 *Effects:* <TODO>

```
void auto_clear(bool val) noexcept;
```

24 *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 [io2d.standalone]

17.1 Standalone functions synopsis [io2d.standalone.synopsis]

```
namespace std::experimental::io2d::v1 {
  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;
}
```

17.2 `copy_surface` [io2d.standalone.copysurface]

```
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;
```

1 *Returns:* `GraphicsSurfaces::surfaces::copy_surface(sfc)`.

17.3 `degrees_to_radians` [io2d.standalone.degtorad]

```
template <class T>
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` [io2d.standalone.radtodeg]

```
template <class T>
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` [io2d.standalone.angleforpoint]

```
float angle_for_point(point_2d ctr, point_2d pt) noexcept;
```

1 *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`.

17.6 `point_for_angle` [io2d.standalone.pointforangle]

```
point_2d point_for_angle(float ang, float rad = 1.0f) noexcept;
point_2d point_for_angle(float ang, point_2d rad) noexcept;
```

1 *Requires:* If it is a `float`, `rad` is greater than `0.0f`. If it is a `point_2d`, `rad.x` or `rad.y` is greater than `0.0f` and neither is less than `0.0f`.

2 *Returns:* The result of rotating the point `point_2d{ 1.0f, 0.0f }`, around an origin of `point_2d{ 0.0f, 0.0f }` by `ang` radians, with a positive value of `ang` meaning counterclockwise rotation and a negative value meaning clockwise rotation, with the result being multiplied by `rad`.

17.7 `arc_start` [io2d.standalone.arcstart]

```
point_2d arc_start(point_2d ctr, float sang, point_2d rad,
  const matrix_2d& m = matrix_2d{}) noexcept;
```

1 *Requires:* `rad.x` and `rad.y` are both greater than `0.0f`.

2 *Returns:* As-if:

```
auto lmtx = m;
lmtx.m20 = 0.0f; lmtx.m21 = 0.0f;
auto pt = point_for_angle(sang, rad);
return ctr + pt * lmtx;
```

3 [*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 `arc_center` [io2d.standalone.arccenter]

```
point_2d arc_center(point_2d cpt, float sang, point_2d rad,
  const matrix_2d& m = matrix_2d{}) noexcept;
```

1 *Requires:* `rad.x` and `rad.y` are both greater than `0.0f`.

2 *Returns:* As-if:

```
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;
```

17.9 `arc_end` [io2d.standalone.arcend]

```
point_2d arc_end(point_2d cpt, float eang, point_2d rad,
  const matrix_2d& m = matrix_2d{}) noexcept;
```

1 *Requires:* `rad.x` and `rad.y` are both greater than `0.0f`.

2 *Returns:* As-if:

```
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;
```

Annex A (informative) Bibliography [bibliography]

1 The following is a list of informative resources intended to assist in the understanding or use of this Technical Specification .

(1.1) — Porter, Thomas and Duff, Tom, 1984, Compositing digital images. ACM SIGGRAPH Computer Graphics. 1984. Vol. 18, no. 3, p. 253-259. DOI 10.1145/964965.808606. Association for Computing Machinery (ACM)

(1.2) — Foley, James D. et al., *Computer graphics: principles and practice.* 2nd ed. Reading, Massachusetts : Addison-Wesley, 1996.

Index

Index of library names

Index of implementation-defined behavior

The entries in this section are rough descriptions; exact specifications are at the indicated page in the general text.